Nonne & Schneider
Informationssysteme GmbH

Applikationsgenerator FIX
Release 3.1.0

Handbuch für Anwendungsentwickler

Handbuchversion: Edition 3.1.0, 05.06.2008

Nonne & Schneider Informationssysteme GmbH
Friedrich-List-Str. 31
35398 Gießen
Tel.: 0641/97477-0
Fax: 0641/97477-77

Im Zuge der Weiterentwicklung des Produkts *FIX* können Leistungsmerkmale hinzukommen, verändert werden oder entfallen.

Vorwort

Dieses Handbuch beschreibt *FIX* in der Version 3.1.0. *FIX* ist ein Applikationsgenerator für UNIX®-Systeme[1], der den Anwendungsentwickler bei der Erstellung von in der Regel datenbankgestützten Dialogprogrammen in der Sprache C unterstützt.

Das bislang am häufigsten mit *FIX* eingesetzte Datenbanksystem ist IBM Informix®[2]. Dieses Handbuch bezieht sich an einigen Stellen direkt auf IBM Informix und verwendet dessen Terminologie; die Unterschiede zu anderen relationalen Datenbanksystemen, die SQL verwenden, sind jedoch relativ gering. Welche Produkte Sie - neben einem C-Compiler, der auf vielen UNIX-Systemen bereits vorhanden ist - benötigen, um mit *FIX* Programme zu entwickeln, können Sie im Anhang *Softwarevoraussetzungen* nachlesen.

FIX wird entwickelt und vertrieben von

Nonne & Schneider Informationssysteme GmbH
Friedrich-List-Str. 31
35398 Gießen
Tel.: 0641/97477-0
Fax: 0641/97477-77

Wie lernen Sie FIX kennen?

Mit jedem *FIX*-Entwicklungssystem wird eine kleine Demo-Anwendung ausgeliefert, das “Software-Pröbchen”. Um es sich anzusehen, wechseln Sie in das Verzeichnis, in dem Sie *FIX* installiert haben (üblicherweise `fix`), und führen Sie dort das Script `fd` aus: Dieses wechselt in das Unterverzeichnis `demo`, liest die Datei `.profile` und verzweigt in das Programm.

So erhalten Sie zunächst einmal einen Eindruck davon, wie ein mit *FIX* erzeugtes Programm “aussieht”, wie also auch Ihre Anwendungsprogramme aussehen können, wenn Sie sich der vielfältigen Möglichkeiten von *FIX* bedienen.

Zum Aufbau des Handbuchs

Die Kapitel *1* bis *8* geben einen Überblick über Elemente und Bedienung von mit *FIX* erstellten Anwendungen.

Kapitel *9* beschreibt die Struktur von *FIX* und *FIX*-Anwendung.

Kapitel *10* stellt die Entwickler-Oberfläche von *FIX* vor.

Weitere *FIX*-Tools werden im Kapitel *11* vorgestellt.

Die Erstellung der deskriptiven Teile einer Anwendung beschreiben die Kapitel *12* bis *21*.

In den Kapiteln *22* bis *37* werden die zentralen Datenstrukturen und Operationen, die Anwendungen verwenden, vorgestellt.

Kapitel *38* beschreibt spezielle Features von FIX.

Kapitel *39* beschreibt Features von *FIX*, die nur bei Einsatz eines Grafik-Frontends (*FIX/Win* oder *FIX/Web*) zum Tragen kommen.

Kapitel *40* schildert, wie der Zeichensatz definiert wird.

---

1. UNIX ist ein eingetragenes Warenzeichen der Open Group.
2. IBM Informix ist ein eingetragenes Warenzeichen der International Business Machines Corporation.

Kapitel *41* beschreibt die Semigrafik-Möglichkeiten.

Das Ein-/Ausgabe-Konzept von *FIX* erläutert Kapitel *42*.

Kapitel *43* beschreibt alle dem Entwickler zur Verfügung stehenden Library-Routinen.

Globale Variablen, die der Entwickler abfragen oder setzen kann, um das Verhalten von *FIX*-Routinen zu beeinflussen, beschreibt Kapitel *44*.

Kapitel *45* stellt dar, wie *FIX* Objektbeschreibungen generiert.

Kapitel *46* erläutert die von *FIX* aus Masken generierten Programmtext-Muster sowie die darin verwendeten “Methoden” der Datenextraktion und -manipulation.

Kapitel *47* erläutert die von *FIX* aus Menübeschreibungen generierten Programmtext-Muster.

Kapitel *48* gibt Hinweise, wie Sie Schwierigkeiten mit 7- bzw. 8Bit-Plattformen vermeiden können.

Wie Sie *FIX* an Terminals anpassen, erfahren Sie in Kapitel *49*.

In Kapitel *50* wird das Lizenzkonzept von *FIX* erläutert.

Anhang A zeigt, wie Sie die Darstellung von Werten an länderspezifische Gepflogenheiten anpassen können.

Anhang B beschreibt die aus früheren *FIX*-Versionen übernommenen C-ISAM®[1] gestützten Selos.

Anhang C beschreibt seit Version 2.9.x enthaltene, aber nicht voll unterstützte Features.

Anhang D nennt die Komponenten des Runtime-Systems.

Anhang E nennt die IBM Informix-Produkte, die Sie zum Einsatz von *FIX* benötigen.

Anhang F gibt den Zeichensatz ISO 8859:15 wieder.

Anhang G weist auf Unterschiede der *FIX*-Version 3.1.0 gegenüber der Vorversion hin.

Eine Bitte an Sie

Sie kennen das: Wenn man das eigene Produkt beschreibt, wird man leicht blind für die Probleme, die ein unbefangener Leser mit dieser Beschreibung hat. Lassen Sie uns wissen, was Ihnen Verständnisschwierigkeiten bereitet hat, wo wir uns unklar, missverständlich oder sogar falsch ausgedrückt haben.

Und sollten Sie Unzulänglichkeiten in *FIX* selbst entdecken, so teilen Sie es uns ebenfalls mit. Für Anregungen sind wir stets offen. Schließlich ist auch *FIX* mit seinen Anwendungen und durch seine Anwender gewachsen.

1. C-ISAM ist ein eingetragenes Warenzeichen der International Business Machines Corporation.

Inhaltsverzeichnis

1 *FIX* im Überblick

Bei *FIX* existieren zwei grundlegende Elemente: *Objekte* und *Ereignisse*. Das verbindende Glied zwischen beiden, das ausführende Element sozusagen, sind die *Routinen der FIX-Library.*

1 Objekte

FIX unterstützt den Entwickler beim Erstellen einer Anwendung durch vordefinierte Objekte wie

- *Menüs*
  Auswahl einer von mehreren vorgegebenen Handlungsalternativen
- *Masken*
  Einzelsatz-Masken: Bearbeitung eines Datensatzes,
  Mehrsatz-Masken: “in memory”-Bearbeitung einer Menge von gleich strukturierten Datensätzen
- *Selos*
  “Fenster in die Datenbank” zur Auswahl eines Datensatzes
- *Choices*
  Auswahl einer Teilmenge von evtl. strukturierten Werten
- *Hilfetexte*

Jedes Objekt wird durch eine eigene Beschreibungs*datei* definiert.

Grundsätzlich sind alle Objekte eigenständig verwendbar. Komplexe Anwendungen basieren allerdings auf Beziehungen zwischen Objekten. Typische Beziehungen unterstützt *FIX* hierbei schon auf Beschreibungsebene.

Hierarchische Beziehungen

> Hierunter fällt beispielsweise die Verwendung von Masken als Elemente anderer Masken (Einbettung), die sich hervorragend zur Modellierung von 1:n-Beziehungen eignet, oder die Verwendung von Menüs als Untermenüs.

Explizite Beziehungen

> Hier wird die Beziehung durch eine *Bindung* von Objekten an Elemente anderer Objekte hergestellt. Im Gegensatz zu hierarchischen Beziehungen können solche Beziehungen nicht nur auf Beschreibungsebene, sondern auch per Programm erzeugt werden. Beispielsweise können Felder mit Choices oder Selos zur Auswahl kombiniert werden.

Implizite Beziehungen

> Hier wird die Verbindung über den Namen des Objekts hergestellt. So assoziiert *FIX* die Hilfetext-Datei `file` mit dem gleichnamigen Feld oder Menüpunkt.

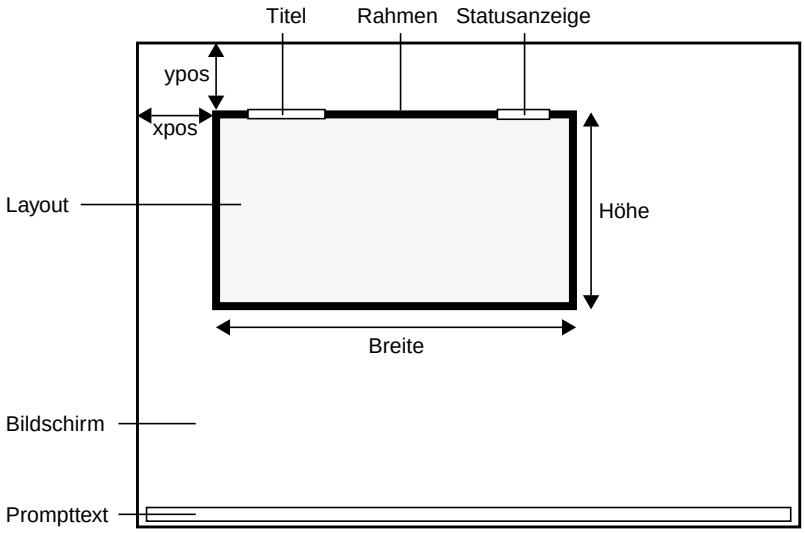
Am Bildschirm können zur gleichen Zeit beliebig viele Objekte dargestellt werden, jedes in einem eigenen *Window.* Jedoch ist höchstens eines dieser Objekte *aktiv,* d.h. für Eingabe aufnahmebereit.

Windows muss man sich in Schichten übereinander liegend vorstellen: wo sich zwei überschneiden, wird das darunterliegende verdeckt. Im Verlauf der Programmausführung kann sich die Schichtabfolge und Position der Windows ändern; neue Windows können hinzukommen und vorhandene wieder verschwinden.

Mit Ausnahme von Hilfetexten und Selos besitzen alle Objekte eine feste, vom Entwickler vorgegebene Größe und Lage auf dem Bildschirm. Sie können wahlweise mit einem Rahmen umgeben und mit einem Titel und einem Hintergrund-Layout versehen werden, das auch Semigrafik enthalten kann. Einem Objekt kann ein einzeiliger *Prompt*text zugeordnet werden, der am unteren Bildschirmrand erscheint, wenn das Objekt aktiv ist. Zu den Elementen von Objekten, wie z.B. zu Feldern, können außerdem ausführliche, auf Tastendruck abrufbare Hilfetexte hinterlegt werden.

Größe und Position von Selos kann *FIX* abhängig von Inhalt und Bildschirmgröße auch dynamisch festlegen.

Abb. 1 Schematische Darstellung eines Objekt-Windows



Um die Verwaltung der Windows braucht sich der Entwickler in der Regel nicht zu kümmern. *FIX* sorgt selbst dafür, dass sich das Window des aktiven Objekts jeweils im Vordergrund befindet. Will der Entwickler dennoch Einfluss nehmen, kann er die Funktionen des Window-Managers auch explizit benutzen.

2 Interaktion mit dem Anwender

FIX-Programme werden über die Tastatur bedient.[1] Da Tastaturen sich je nach Rechnersystem hinsichtlich der Anzahl und der Beschriftung der Tasten stark unterscheiden, liegt *FIX* eine *abstrakte Tastatur* zugrunde, die neben den mit Zeichen markierten Tasten etliche zusätzliche *Sondertasten* umfasst. Sondertasten sind solche Tasten, die nicht zur Dateneingabe, sondern zur Ablaufsteuerung dienen.

Jeder Sondertaste entspricht eine einzelne bzw. eine Folge von Tasten, die sich auf Ihrer Terminal-Tastatur befinden. Zur Bezeichnung der Sondertasten dienen aus zwei Zeichen bestehende Benennungen.[2]

Die folgende Tabelle gibt einen Überblick über die Sondertasten (exemplarisch sind hier Belegungen für zwei weitverbreitete Tastaturen aufgeführt, die im Übrigen frei konfigurierbar sind).

Kürzel	Mnemo	Fujitsu Siemens 97801	PC-102
ku	CursorUp	↑	↑
kd	CursorDown	↓	↓
kl	CursorLeft	←	←
kr	CursorRight	→	→

1. Die Frontends *FIX/Win* und *FIX/Web* unterstützen auch eine Bedienung mit der Maus.

2. In früheren Releases wurden die Tasten im termcap-File definiert, das zur Bezeichnung von Capabilities nur Identifier der Länge 2 erlaubt.

TB	Tab	→|	→|
BT	Backtab	|←	|←
kh	Home	↖	Pos1
PU	PageUp	⇑	Bild ↑
PD	PageDown	⇓	Bild ↓
PL	PageLeft	↶	- (numerischer Block)
PR	PageRight	↷	Esc P R
RT	Return	↵	↵
EN	End	END	Ende
HP	Help	HELP	F1
hc	Hardcopy	ESC h c	Esc h c
DL	Backspace	⌫	⟵
CE	Clear-Entry	C_E	Strg-F8
CI	Character Insert	INSERT_CHAR.	Einfg
CD	Character Delete	DELETE_CHAR.	Strg-F5
WI	Word Insert	INSERT_WORD	Strg-F3
WD	Word Delete	DELETE_WORD	Strg-F6
LI	Line Insert	INSERT_LINE	Strg-F4
LD	Line Delete	DELETE_LINE	Strg-F7
ST	Start	START	Strg-F9
MD	Mode	MODE	F12
PT	Print	PRINT	Strg-F11
sh	Shell	Esc s h	Esc s h
f1	(reserviert)		
f2	CPYFIELD	F2	F2
f3	JMPFIELD	F3	F3
f4	INSERT	F4	F4
f5	DETAIL	F5	F5
f6	ADD	F6	F6
f7	DELETE	F7	F7
f8	KEYFIRST	F8	F8
f9	KEYSELECTION	F9	F9
f0	KEYLAST	F10	F10
g1	REBUILD	F11	F11
g2	(reserviert)		
g3	(reserviert)		
g4	(reserviert)		
g5	T_HELP	F15	Strg-F1
g6	CHELP	F16	Shift-F1
g7	UNDO	F17	Shift-F7
g8	EDITHELP	F18	Shift-F8

Jede dieser Sondertasten löst ein *Event* aus, für das eine Standard-Bedeutung definiert ist, die vom aktiven Objekt abhängt; eine abweichende Reaktion kann der Entwickler *objektbezogen* mittels einer selbst erstellten Funktion festlegen, in der er seine eigene Logik formuliert.

Als reserviert bezeichnete Tasten sind für zukünftige Erweiterungen vorgesehen und sollten daher von Anwendungen nicht benutzt werden.

Daneben unterstützt *FIX* zwölf weitere Sondertasten (g9, g0, h1, …, h0), deren Bedeutung ausschließlich dem Anwendungsprogramm überlassen bleibt. Einige dieser Tasten werden auch von *FIX*-Tools wie dem Layout-Editor **led** verwendet.

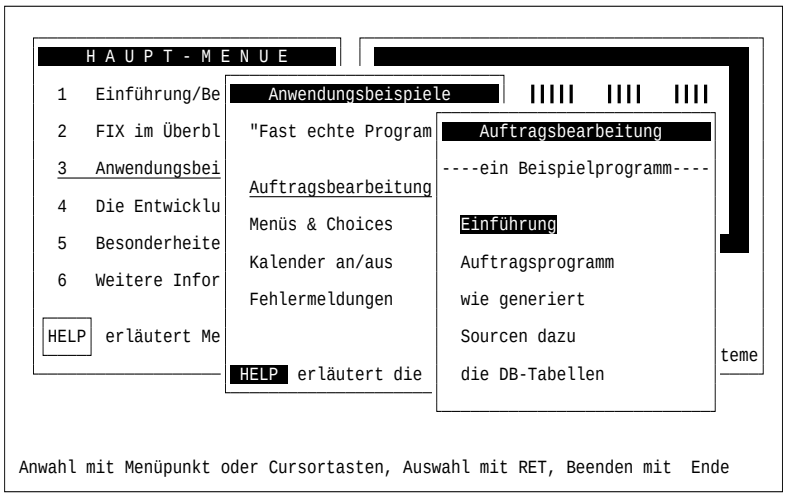
Welche Taste(n) Sie bei Ihrem Terminal für eine Sondertaste drücken müssen, zeigt Ihnen ein Hilfetext, den sie in jedem *FIX*-Programm durch zweimaliges Drücken der Help-Taste aufblenden können.

Wie Sie *FIX* an eine Tastatur anpassen, finden Sie im Kapitel “Terminal-Anpassung” auf Seite 511.

2 Menüs

Menüs dienen im Allgemeinen dazu, den Anwender eine Auswahl bezüglich des weiteren Programmablaufs treffen zu lassen. Aus den durch die *Menüpunkte* vorgegebenen Alternativen kann eine gewählt und anschließend ausgeführt werden.

Abb. 2 Beispiel für ein Menü



Wenn ein Menü aktiv wird, bringt *FIX* es automatisch am Bildschirm in den Vordergrund und zeigt am unteren Bildschirmrand den zugehörigen Prompttext. Menüs besitzen stets einen aktuellen (= angewählten) Menüpunkt, der optisch hervorgehoben ist (invers); beim Betreten ist dies in der Regel der erste Menüpunkt. Die letzte Zeile des Menüs kann optional eine kurze Erläuterung zum gerade angewählten Menüpunkt enthalten.

Bei der Bearbeitung des Menüs kann der Anwender einen anderen Menüpunkt anwählen oder den aktuellen Menüpunkt auslösen, d.h. die dahinterstehende Aktion starten. Diese kann programmiert sein, aber auch einfach im Aufblenden eines weiteren (Unter-)Menüs bestehen.

Die Anwahl kann durch schrittweises Hinbewegen zu dem gewünschten Menüpunkt mittels Sondertasten oder durch Angabe des/der Anfangsbuchstaben des Menüpunktes erfolgen. Ausgelöst wird die Aktion erst durch Drücken der Taste RT. Während der Ausführung ändert sich die Art der Hervorhebung (unterstrichen).

Menüs reagieren - sofern vom Entwickler kein abweichendes Verhalten spezifiziert wird - auf die Sondertasten

kh	Home	zum ersten Menüpunkt
kd	CursorDown	zum nachfolgenden Menüpunkt
ku	CursorUp	zum vorausgehenden Menüpunkt
kr	CursorRight	zum nachfolgenden Menüpunkt auf der gleichen Zeile
kl	CursorLeft	zum vorausgehenden Menüpunkt auf der gleichen Zeile
PU	PageUp	zum ersten Menüpunkt
PD	PageDown	zum letzten Menüpunkt
RT	Return	Aktion auslösen
EN	End	Menü beenden/abbrechen
ST	Start	zurück zum Ausgangsmenü

g5	T_HELP	allgemeine Tastenerläuterung anzeigen
HP	Help	Hilfetext zum angewählten Menüpunkt anzeigen
g8	EDITHELP	nur im Entwicklermodus: Hilfetext zum Menüpunkt erstellen bzw. ändern
g1	REBUILD	Bildschirm neu aufbauen
sh	Shell	in eine Shell verzweigen

Statt kd kann auch die Leertaste benutzt werden.

Alphanumerische Eingaben werden i.d.R. zu einer Zeichenfolge zusammengefasst und bewirken die Anwahl des nächstfolgenden Menüpunktes, der mit dieser Zeichenfolge beginnt. Bei einer nicht alphanumerischen Eingabe - außer DL (Backspace) - oder wenn kein Menüpunkt mit diesem Präfix existiert, bricht die Anwahl ab.

Alternativ unterstützt *FIX* auch Menüs, bei denen eine alphanumerische Eingabe unmittelbar die Auslösung des mit ihr angewählten Menüpunktes bewirkt. In diesem Fall durchsucht *FIX*, wenn das aktive Menü keinen passenden Menüpunkt enthält, auch übergeordnete Menüs in umgekehrter Reihenfolge ihres Betretens. Wird ein passender Menüpunkt gefunden, wird in dieses Menü zurückgekehrt, wobei alle dazwischen liegenden Untermenüs regulär beendet werden, und weiterverfahren, als sei zum Zeitpunkt des Tastendrucks dies das aktive Menü gewesen.

FIX/Win, FIX/Web

Bei Einsatz von *FIX/Win* oder *FIX/Web* kann ein Menüpunkt auch durch Anklicken mit der Maus angewählt bzw. ausgelöst werden.

3 Masken

Masken erlauben es, Daten kommentiert und in übersichtlicher Form zu präsentieren oder zu bearbeiten. Dazu enthalten sie *Felder*. Aus Sicht des Anwenders ist ein Feld ein Bereich innerhalb der Maske, in dem ihm ein Wert angezeigt wird oder in dem er einen Wert eingeben bzw. verändern kann. Ein solches Feld hat stets einen Typ, der die Art der Werte bestimmt (Zeichenkette, Zahl, Datum etc.). In der Praxis kommen häufig weitere Einschränkungen hinzu: der Wert eines Feldes muss zu denen anderer Felder in Beziehung stehen, aus einer vorgegebenen Menge stammen oder sonstige Bedingungen erfüllen.

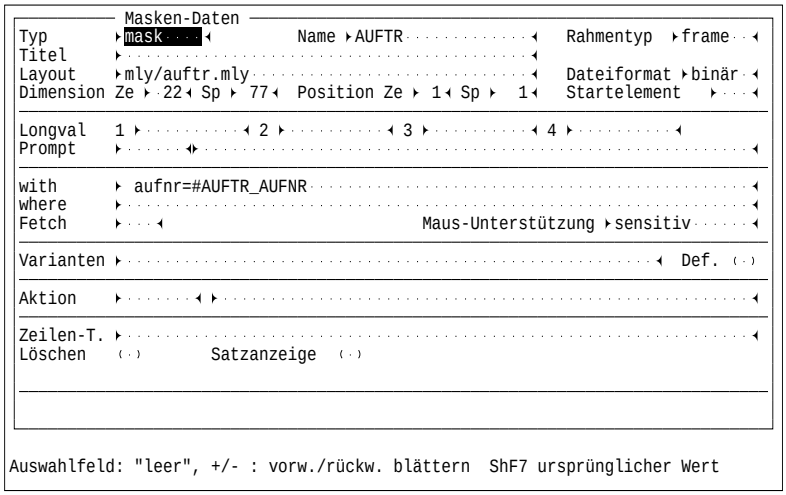
Eine Maske im Sinne von *FIX* kann als Elemente neben Feldern auch *eingebettete Masken* enthalten, was vor allem bei der Modellierung von 1:n-Beziehungen hilfreich ist.

FIX unterscheidet zwei Klassen von Masken: *Einzel-* und *Mehrsatz-Masken*.

Einzelsatz-Masken

Hier handelt es sich um die klassische Form einer Datenerfassungsmaske, wie sie allgemein bekannt ist. Abb. 3 vermittelt einen Eindruck von einer Maske. Die Felder sind hier von ‘▸’ und ‘◂’ bzw. ‘(’ und ‘)’ eingeschlossen.

Abb. 3 Beispiel für eine Maske



Mehrsatz-Masken

Mehrsatz-Masken können beliebig viele Datensätze speichern und gestatten es dem Anwender, zwischen den Sätzen zu blättern. Der jeweils aktuelle Satz kann dabei in der gleichen Weise wie bei einer Einzelsatz-Maske bearbeitet werden. Der Vergleich mit einem Cursor im Sinne des Datenbanksystems bietet sich an. *FIX* kennt drei Arten von Mehrsatz-Masken: *Sub-*, *Roll-* und *Tabellenmasken*.

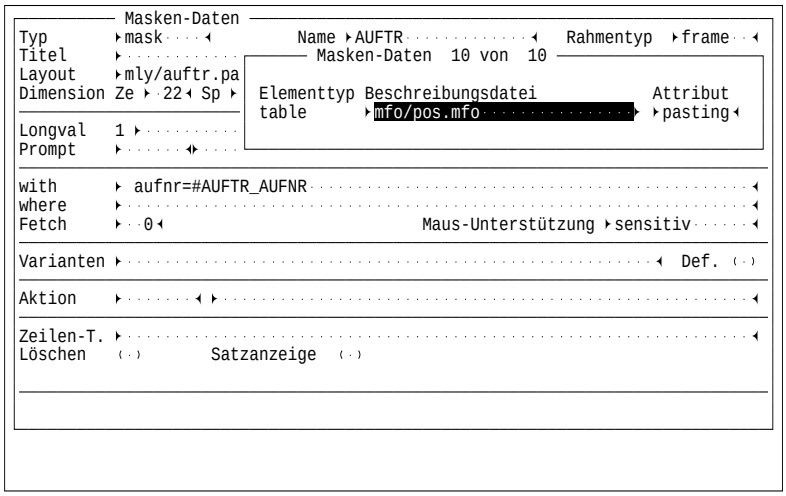
- Sub- und Rollmasken

  Sub- und Rollmasken können zwar beliebig viele Datensätze aufnehmen, zeigen zu einem Zeitpunkt jedoch nur einen Satz an. Der Anwender kann mittels Tasten zwischen den verschiedenen Sätzen wechseln. Eine optionale

Satzanzeige am oberen Maskenrand informiert ihn dabei über die Gesamtanzahl der Sätze und die Nummer des aktuellen Satzes.

Zwischen Sub- und Rollmasken gibt es lediglich geringfügige Unterschiede in der Bedienung, wobei aus Komforterwägungen meist die Rollmaske bevorzugt wird.

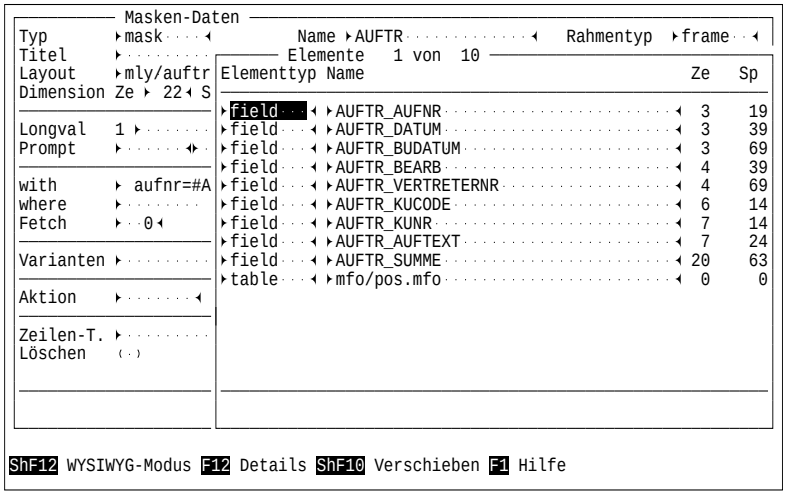
Abb. 4 Beispiel für eine Rollmaske



- Tabellenmasken

  Mit einer Tabellenmaske können im Gegensatz zu einer Rollmaske mehrere Sätze gleichzeitig angezeigt werden, und zwar so viele, wie in das zugehörige Masken-Window hineinpassen. Meist wird der Entwickler die Tabellenmaske so gestalten, dass pro Bildschirmzeile ein Datensatz dargestellt wird, ein Satz kann aber auch über mehrere Bildschirmzeilen verteilt werden. Zusätzlich zum “Scrollen” um jeweils einen Satz ist hier auch ein seitenweises Blättern möglich.

Abb. 5 Beispiel für eine Tabellenmaske



Ergänzend werden *Tabellenmasken mit Zeilentypen* angeboten, mit denen verschiedene Darstellungen für Sätze innerhalb einer Tabellenmaske realisiert werden können. Näheres hierzu finden Sie auf Seite 99.

Varianten

Eine *Variante* zu einer Maske ist eine alternative Maske mit gleichartigen Elementen, die eine andere Sicht der Daten beschreibt. Insbesondere können Varianten unterschiedliche Elemente (Felder) ausblenden, oder andersherum betrach-

tet, anzeigen. Am besten zu illustrieren ist dies an folgendem - häufig verwendeten - Fall: Eine Tabellenmaske zeigt, in jeweils einer Zeile, wenige Felder eines Datensatzes an und stellt damit eine Übersicht über viele Datensätze zur Verfügung. Der Wechsel in die Variante, eine Rollmaske, zeigt nur noch einen Datensatz an, jedoch mit allen Feldern des Satzes - also eine Detailsicht.

Wichtig ist dabei, dass der Entwickler nur eine Sicht in seiner Ablauflogik zu formulieren und den Variantenwechsel im Programm nicht zu berücksichtigen braucht (wohl aber kann); das übernimmt *FIX*, so dass Varianten auch noch nachträglich einem Maskensystem hinzugefügt werden können.

Wenn eine Maske aktiv wird, bringt *FIX* sie automatisch am Bildschirm in den Vordergrund und zeigt am unteren Bildschirmrand den zugehörigen Prompttext. Beim Bearbeiten “besucht” *FIX* ihre Elemente in einer von der Programmlogik und der Anwendereingabe abhängigen Reihenfolge, beginnend mit dem in der Maskenbeschreibung festgelegten *Startelement* des aktuellen Satzes.

Wird ein Feld besucht, wird es optisch hervorgehoben (invers). Die Schreibmarke steht je nach Feldeigenschaften entweder am linken Rand des Feldes oder hinter dem letzten Zeichen. Im Verlauf der Erfassung zeigt sie jeweils die Position an, an der das nächste Zeichen[1] eingefügt werden kann. Unzulässige *Zeichen*eingaben lösen ein akustisches Signal aus und werden abgewiesen.

Die Felderfassung interpretiert neben Zeichen eine Reihe von Sondertasten zum

- Bewegen der Schreibmarke über den Feldinhalt:

kl	CursorLeft	Schreibmarke nach links
kr	CursorRight	Schreibmarke nach rechts
TB	Tab	Schreibmarke hinter das letzte Zeichen im Feld
BT	Backtab	Schreibmarke zum Feldanfang

- Einfügen und Löschen von Zeichen:

CI	Character Insert	Zeichen unter Schreibmarke einfügen[2]
CD	Character Delete	Zeichen unter Schreibmarke löschen
DL	Backspace	Zeichen links neben Schreibmarke löschen
WD	Word Delete	Zeichen ab Schreibmarke löschen
CE	Clear Entry	alle Zeichen löschen

- Restaurieren des Feldinhaltes bei der vorherigen Bearbeitung:

f2	CPYFIELD	vorherigen Feldinhalt rekonstruieren

- Restaurieren des ursprünglichen Inhaltes:

g7	UNDO	ursprünglichen Feldinhalt restaurieren

- Anzeigen des vom Entwickler vorgesehenen Hilfetextes:

HP	Help	Hilfetext zum Feld anzeigen

- Erfassen und Anzeigen eines eigenen Hilfetextes:

g6	CHELP	Text anzeigen
g8	EDITHELP	Text erstellen oder ändern nur im Entwicklermodus: Hilfetext zum Feld erstellen oder ändern

- Erläutern der Tastenbedeutung:

g5	T_HELP

- Neuaufbau des Bildschirms:

g1	REBUILD

- Verzweigen in die Shell:

1. In diesem Zusammenhang wollen wir nicht darstellbare Kontrollzeichen nicht als “Zeichen” ansehen.
2. Vgl. aber “Bedienungsmodi” auf Seite 34.

sh	Shell	

- *regulären* Abschließen der Felderfassung:

RT	Return	Feldprüfungen ausführen und Feld verlassen
f3	JMPFIELD	wie RT

Aufgrund der Möglichkeit, Feldinhalte zu editieren, können diese kurzfristig nicht typgerecht sein, z.B. zu viele Nachkommastellen oder unsinnige Datumsbestandteile enthalten. Ein Feld kann allerdings erst verlassen werden, wenn sein Inhalt typgerecht ist, also als Wert interpretiert werden kann. Auf weitere Besonderheiten bei der Felderfassung weist das Kapitel “Felder” auf Seite 33 hin.

Alle oben nicht aufgeführten Sondertasten bewirken ein Verlassen des Feldes (mit dem augenblicklichen Wert) und bestimmen - sofern der Entwickler keine eigene Bedeutung vorgesehen hat - das nächste zu besuchende Feld oder Objekt.

Ohne Einflussnahme des Entwicklers reagiert *FIX* folgendermaßen:

f9	F9	Selo oder Choice zur Auswahl aktivieren, danach ins Feld zurückkehren und, sofern ein Satz ausgewählt wurde, RT simulieren
MD	Mode	Umschalten in (nächste) Maskenvariante, danach ins Feld zurückkehren
EN	End	Maske (vorwärts) verlassen

Die Bedeutung weiterer Tasten unterscheidet sich geringfügig je nach Maskenart:

Einzelsatz-Maske:

RT	Return	zum nachfolgenden Element wenn im letzten Element: Maske (vorwärts) verlassen
f3	JMPFIELD	Turbo-Taste: zum nächsten Stop-Feld oder eingebetteten Maske wenn nicht vorhanden: Maske (vorwärts) verlassen
PL	PageLeft	zum vorausgehenden Element wenn im Startelement: Maske (rückwärts) verlassen
kh	Home	zum Startelement wenn im Startelement: unzulässig

Mehrsatz-Maske:

RT	Return	zum nachfolgenden Element wenn im letzten Element: zum nachfolgenden Satz, der ggf. neu angelegt wird
f3	JMPFIELD	Turbo-Taste: zum nächsten Stop-Feld oder eingebetteten Maske wenn nicht vorhanden: zum nachfolgenden Satz, der ggf. neu angelegt wird
PL	PageLeft	zum vorausgehenden Element wenn im Startelement: Maske (rückwärts) verlassen
kh	Home	zum Startelement wenn im Startelement: zum obersten Satz wenn im Startelement des obersten Satzes: Maske (rückwärts) verlassen

Bei einer Submaske wird im Gegensatz zu einer Roll- oder Tabellenmaske bei RT und f3 nicht automatisch zum nächsten Satz übergegangen, sondern in einen speziellen Modus verzweigt (vgl. Seite 201).

Zum Blättern durch die Sätze kommen hinzu:

f8	KEYFIRST	zum ersten Satz
f0	KEYLAST	zum letzten Satz
f4	INSERT	Satz einfügen
f6	ADD	Satz anfügen

f7	DELETE	Satz löschen
PD	PageDown	zum nachfolgenden Satz
PU	PageUp	zum vorausgehenden Satz

Bei Tabellenmasken erfolgt das Blättern durch die Sätze etwas unterschiedlich:

kd	CursorDown	zum nachfolgenden Satz
ku	CursorUp	zum vorausgehenden Satz
PD	PageDown	nachfolgende Seite
PU	PageUp	vorausgehende Seite

Beim Blättern durch die Sätze wird möglichst wieder das gleiche Feld betreten.

Auf Tasten, die nicht interpretiert werden können, reagiert *FIX* mit einem Warnton und kehrt in das vormals aktuelle Feld zurück.

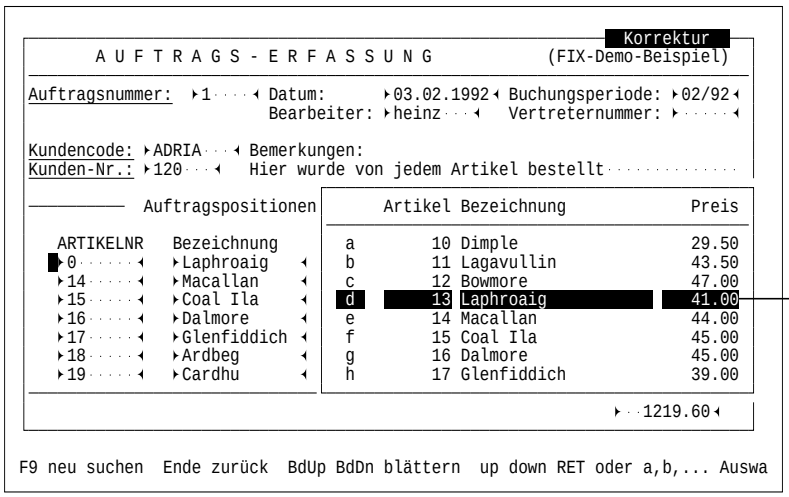
FIX/Win, FIX/Web

Bei Einsatz von *FIX/Win* oder *FIX/Web* kann unter bestimmten Voraussetzungen das nächste Feld auch durch Anklicken mit der Maus bestimmt werden.

4 Selos

Zweckmäßig für ein Programm, das mit einer Datenbank zusammenarbeitet, ist die Fähigkeit, dass Daten aus der Datenbank angezeigt, ausgewählt und in Felder übernommen werden können. Diese Aufgabe übernehmen Selos (der Name “Selo” leitet sich ab von “Select Logic”). Ein Programm wird durch den intensiven Einsatz von Selos recht anwenderfreundlich. Obgleich Selos eigenständig existieren können, macht ihr Einsatz meist nur Sinn in Verbindung mit einem Feld.

Abb. 6 Beispiel für ein Selo



FIX kennt zwei Arten von Selos, die sich hinsichtlich der Art der Datenbeschaffung unterscheiden: SQL gestützte und C-ISAM gestützte Selos. Letztere sind in *Anhang B* beschrieben.

SQL gestützte Selos definieren die Datenmenge entweder durch eine **select**-Anweisung (Mehr zu deren Aufbau finden Sie auf Seite 105) oder eine **execute procedure**-Anweisung.

select-Anweisung

Beispiel:

```
selo
select aufnr, datum, bearb, auftr.kunr nr, name from auftr, kunde
where auftr.kunr = kunde.kunr and bearb > " " ;          Grundanweisung
restrict aufnr >= ?AUFTR.AUFNR ;                         interaktive Zusatzbedingung
aufnr … #AUFTR.AUFNR,
datum … ,                                                Angaben zur Darstellung und
bearb … ,                                                Übernahme von Spaltenwerten in
nr … ,                                                   Maskenfelder
name …
end
```

Durch die **restrict**-Angabe wird *FIX* veranlasst, vor der Datenbeschaffung im Feld AUFNR der Maske AUFTR nach einer unteren Schranke bzgl. der zu suchenden Auftragsnummern zu fragen.

Wenn ein Selo aktiv wird, bringt *FIX* es automatisch am Bildschirm in den Vordergrund. Das Window enthält zu diesem Zeitpunkt in der Regel noch keine Sätze, da der Anwender (optional) vor der Datenbanksuche Gelegenheit erhält, die vom Entwickler durch die Grundanweisung vorgegebene Datenmenge situationsabhängig interaktiv einzuschränken.

Die Interaktion wird gesteuert durch Zusatzbedingungen, die der Entwickler in der **restrict**-Klausel festlegt. Wie die Grundanweisung ist auch diese Zusatzklausel parametrisierbar insofern, als in einfacher Weise auf die Werte von Maskenfeldern Bezug genommen werden kann. In der **restrict**-Klausel unterstützt *FIX* allerdings neben gewöhnlichen Feld-Referenzen, bei denen der in der Maske vorhandene Wert als Parameter übernommen wird, auch solche, bei denen vor der Auswertung der Suchanfrage Maskenfelder[1] nacheinander zur Eingabe angeboten und die abgefragten Werte als Parameter verwendet werden. Diese Referenzen werden zur Unterscheidung statt mit # mit ? eingeleitet.

Bei der Abfrage von Parametern führt die Taste

- RT ins nachfolgende Abfrage-Feld, bzw. startet beim letzten Feld die Suche
- PL zurück ins vorherige Feld, bzw. bricht beim ersten Feld die Seloauswahl ab
- kh ins erste Feld.

Wird ein Abfrage-Feld mit einer anderen Taste verlassen (z.B. f8, f0, EN, f9), wird die Bearbeitung der **restrict**-Klausel abgebrochen und die Grundanweisung benutzt.

Selos basieren auf dynamisch erzeugten Scrollcursorn. Beim jedem Neuaufsetzen der Suche (dies ist auch während des Blätterns auf Tastendruck möglich) stellt *FIX* einen entweder für die Grundanweisung oder für die um die Zusatzbedingungen erweiterte Anweisung geeigneten Cursor bereit.

Beispiel:

Wird ein Anfangswert eingegeben, so führt *FIX* die Anweisung

```
select aufnr, datum, bearb, auftr.kunr nr, name from auftr, kunde
where aufnr >= ? and auftr.kunr = kunde.kunr and bearb > " "
```

aus, wobei für ? der im Feld AUFNR eingegebene Wert eingesetzt wird.

execute procedure-Anweisung

Alternativ kann die Datenmenge auch mit Hilfe einer cursorartigen SPL-Prozedur beschafft werden.[2]

Beispiel:

```
selo
execute procedure auftraege(#, ?AUFTR_AUFNR);     Grundanweisung
                                                   mit interaktivem Parameter
_1[3] … #AUFTR.AUFNR,
_2 … ,                                             Angaben zur Darstellung und
_3 … ,                                             Übernahme von Spaltenwerten in
_4 … ,                                             Maskenfelder
_5 …
end
```

1. *FIX* verwendet in diesem Fall vereinfachte Kopien der betreffenden Felder, wodurch der tatsächliche Feldwert unverändert bleibt und in diesem Kontext irrelevante Feldeigenschaften unterdrückt werden.
2. Prozeduren werden erst ab IBM Informix ESQL/C 5.x unterstützt.
3. Da die Spalten der Ergebnismenge von SPL-Prozeduren unbenannt sind, generiert *FIX* als Namen *_spaltennummer*.

Die Interaktion wird hier gesteuert durch die Form der Argumente, die für die SPL-Prozedur angegeben werden. Analog zur select-Anweisung kann in der Argumentliste auf die Werte von Maskenfeldern Bezug genommen werden. Bei Feld-Referenzen, die mit # eingeleitet sind, werden die in der Maske vorhandenen Werte als Argument übernommen, für mit ? eingeleitete Referenzen werden vor dem Aufruf der Prozedur die Maskenfelder[1] nacheinander zur Eingabe angeboten und die abgefragten Werte als Argument verwendet.

Die spezielle Referenz '#' ersetzt *FIX* durch einen ganzzahligen Wert, den die Prozedur auswerten kann, um zu erkennen, welche Daten zu beschaffen sind:

- 1 - "interaktive Parameter sind nicht relevant" (entspricht Grundanweisung bei select):
  FIX übergibt für alle mit ? eingeleiteten Referenzen NULL als Argument;
- 2 - "interaktiv erfasste Parameter sind signifikant" (entspricht erweiterter Anweisung bei select):
  FIX übergibt die erfassten Werte als Argument.

Wird das Selo zur Satzsuche benutzt (vgl. Seite 214), übergibt *FIX* für # den Wert 3 und für mit ? eingeleitete Referenzen den Wert NULL.[2]

Ist die Datenmenge bestimmt, werden die gelesenen Sätze entsprechend den Formatangaben in der Beschreibungsdatei im Selo-Window dargestellt. Passt die Ergebnismenge nicht in das Window, wird sie in mehrere Seiten aufgeteilt.[3] Die Einträge einer Seite sind mit Kleinbuchstaben markiert. Markierung und Werte des angewählten Satzes werden optisch hervorgehoben (invers).

In Selos sind folgende Sondertasten zulässig:

kd	CursorDown	zum nachfolgenden Satz
ku	CursorUp	zum vorausgehenden Satz
kh	Home	zum ersten sichtbaren Satz
f8	KEYFIRST	zum ersten Satz
f0	KEYLAST	zum letzten Satz
PU	PageUp	vorwärts blättern
PD	PageDown	rückwärts blättern
f9	KEYSELECTION	Auswahl neu starten
RT	Return	aktuellen Satz auswählen
EN	End	Auswahl abbrechen
HP	Help	Hilfetext des Feldes anzeigen, zu dem das Selo gehört
g5	T_HELP	
g1	REBUILD	
sh	Shell	

Ein Satz kann auch durch Eingabe des Buchstabens ausgewählt werden, mit dem er markiert ist; dabei wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Werden keine Sätze gefunden, erfolgt eine Meldung und die Bearbeitung des Selos wird automatisch beendet, als ob der Anwender die Taste EN gedrückt hätte.

Zum nicht interaktiven Einsatz von Selos vgl. Seite 214.

1. vgl. Fußnote auf Seite 26.
2. Für die Satzsuche wird i. Allg. ein weiterer Parameter zur Übergabe des Suchwertes benötigt.
3. Die Pufferung der dargestellten Sätze überlässt *FIX* der Datenbankschnittstelle, so dass beim Blättern durch die Sätze stets neue Leseoperationen erfolgen.

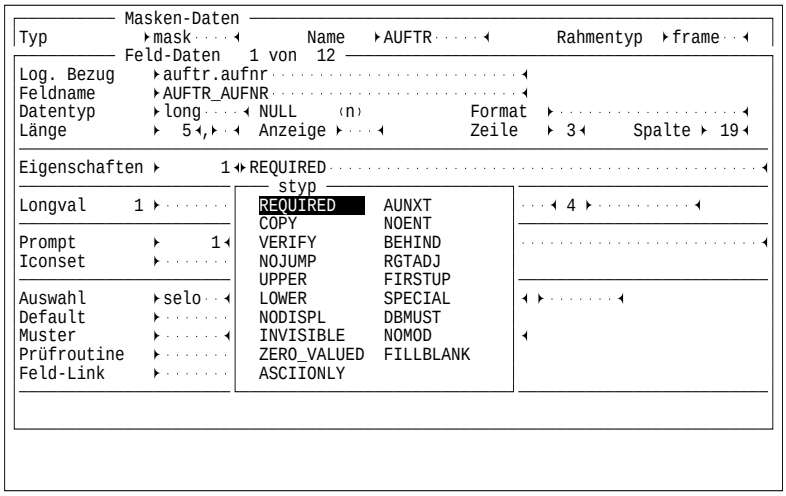
FIX/Win, FIX/Web

Bei Einsatz von *FIX/Win* oder *FIX/Web* kann ein Satz auch durch Anklicken mit der Maus ausgewählt werden.

5 Choices

Choices sind Objekte, mit denen der Anwender eine *Teilmenge* aus vorgegebenen, beliebig strukturierten Werten auszuwählen kann. Der Einsatz von Choices macht wie der von Selos meist nur Sinn in Verbindung mit einem Feld.

Abb. 7 Beispiel für eine Choice



Wenn eine Choice aktiv wird, verwendet *FIX* je nach Vorgabe des Entwicklers die vorgefundene Datenmenge oder bestimmt die Datenmenge neu. *FIX* bringt die Choice automatisch am Bildschirm in den Vordergrund und zeigt am unteren Bildschirmrand den zugehörigen Prompttext.

Die Elemente werden matrixförmig angeordnet dargestellt. Passen nicht alle Elemente in das Window, zerlegt *FIX* sie selbständig in mehrere Seiten (sofern der Platz ausreicht, erscheint dann eine Satzanzeige ähnlich der bei Mehrsatz-Masken). Ausgewählte Elemente werden am Bildschirm optisch hervorgehoben (invers).

Sofern die Anwendung nicht ein anderes Verhalten erzwingt, wird auf dem ersten Element positioniert.

Choices reagieren auf die Sondertasten

kd	CursorDown	zum (vertikal) nachfolgenden Element
ku	CursorUp	zum (vertikal) vorausgehenden Element
kh	Home	zum ersten sichtbaren Element
f8	KEYFIRST	zum ersten Element
f0	KEYLAST	zum letzten Element
PU	PageUp	vorwärts blättern
PD	PageDown	rückwärts blättern
kr	CursorRight	zum nachfolgenden Element der gleichen Zeile
kl	CursorLeft	zum vorausgehenden Element der gleichen Zeile
TB	Tab	zum letzten Element der gleichen Zeile
BT	Backtab	zum ersten Element der gleichen Zeile
MD	Mode	zwischen vertikalem und horizontalem Blättern umschalten
RT	Return	aktuelles Element (de)selektieren
ST	Start	alle Elemente (de)selektieren

CE	Clear Entry	Grundzustand herstellen
EN	End	Auswahl beenden
PL	PageLeft	Auswahl abbrechen
HP	Help	Hilfetext des Feldes anzeigen, zu dem die Choice gehört
g5	T_HELP	
g1	REBUILD	
sh	Shell	

Mit den Tasten ‘>’ und ‘<’ kann ein Block von Elementen selektiert werden. Durch die Eingabe von alphanumerischen Zeichen gelangt der Anwender zum nächsten Element, das mit diesem Zeichen (oder dieser Zeichenfolge) beginnt; hierbei kann der Entwickler bestimmen, ob Groß- und Kleinschreibung unterschieden wird.

FIX/Win, FIX/Web

Bei Einsatz von *FIX/Win* oder *FIX/Web* kann ein Element auch durch Anklicken mit der Maus (de)selektiert werden.

Anbindung an Felder

Die Anbindung einer Choice an ein Feld kann auf dreierlei Art erfolgen:

- *konventionell*: der Anwender gelangt, wie bei Selos, nur durch Drücken der Taste f9 in die Choice,
- *bevorzugt*: jeder Besuch des Feldes führt, sofern es leer ist, aber einen Wert erfordert, automatisch zunächst zu einer Aktivierung der Choice; bei einem Abbruch der Bearbeitung oder einer missglückten Übernahme eines Wertes in das Feld kann der Feldinhalt vom Anwender manuell bearbeitet werden,
- *obligatorisch*: *FIX* behandelt das Feld als “nicht betretbar”, d.h. der Anwender kann den Feldwert nur mittels der Choice manipulieren.

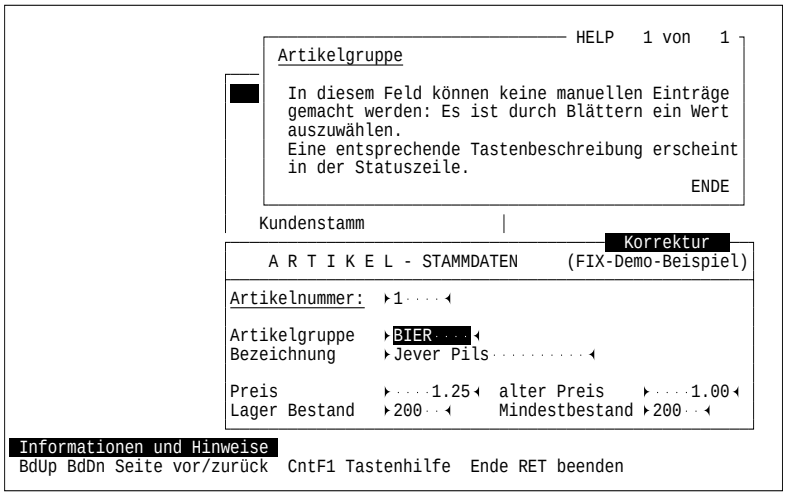
Näheres hierzu finden Sie in “Besuch eines Feldes” auf Seite 208.

6 Hilfetexte

FIX ermöglicht dem Entwickler - und teils auch dem Anwender -, auf einfache Weise zu jedem Menüpunkt oder Feld einen Hilfetext zu erstellen, der in der Anwendung auf Tastendruck abgerufen werden kann.

Ein Hilfetext kann aus nahezu beliebig vielen Seiten bestehen. Er erscheint in einem Window, das so hoch wie die längste Seite und so breit wie die breiteste Seite ist, und wird nach Möglichkeit so am Bildschirm platziert, dass das aktuelle Element des aktiven Objekts nicht überdeckt wird.

Abb. 8 Beispiel für einen Hilfetext



Der Anwender kann seitenweise vorwärts und rückwärts blättern; dabei werden ihm die Nummer der aktuellen Seite und die Gesamtzahl der Seiten angezeigt. Mit den Tasten RT oder EN verschwindet der Text wieder vom Bildschirm.

In Hilfetexten können folgende Tasten benutzt werden:

kh	Home	zur ersten Seite
PD	PageDown	nachfolgende Seite
PU	PageUp	vorausgehende Seite
RT	Return	Hilfetext verlassen
EN	End	Hilfetext verlassen
HP	Help	Tastenbelegung anzeigen
g5	T_HELP	
g1	REBUILD	
sh	Shell	

Vor PD und PU kann ein *Wiederholungsfaktor* angegeben werden, etwa 5 PD für “fünf Seiten vorwärts”.

FIX/Win, FIX/Web

Hilfetexte reagieren nicht auf Klicken.

7 Felder

Felder können bei *FIX* nur als Elemente von Masken definiert werden. *FIX* hat von einem Feld im Wesentlichen folgende Vorstellung:

Ein Feld besitzt

- einen *Namen*, der es - in Verbindung mit dem der Maske - über die ganze Anwendung hinweg eindeutig identifiziert;
- einen *Datentyp*, der bestimmt, welche Art von Werten es aufnehmen kann. Die Feldtypen sind angelehnt an die von relationalen Datenbanken unterstützten Basistypen;
- ein Format bzw. eine *Länge* ($1 \leq l < 2048$), die angibt, wie viele Zeichenpositionen zur Darstellung und Eingabe eines Feldwertes maximal zur Verfügung stehen. Format bzw. Länge müssen hinreichend sein, alle vorkommenden Werte für das Feld aufnehmen zu können;
- einen *Wert*, der in einer *Feld-Hostvariable* abgelegt wird;
- eine Reihe von *Eigenschaften*, z.B. “rechtsbündig” oder “nicht veränderbar”. Eine Übersicht der Eigenschaften finden Sie in “Feldeigenschaften” auf Seite 44.

FIX kennt

- CHARACTER-Felder
- Wahrheitswert-Felder
- Semigrafik-Felder
- Numerische Felder
- Datum-Felder
- Zeitpunkt-Felder
- Zeitspannen-Felder

Bei allen Feldern unterstützt *FIX* den Wert NULL (“undefiniert”). Relationale Datenbanken sehen NULL als Spaltenwert vor, wenn keine logisch sinnvolle Angabe möglich ist oder der tatsächliche Wert zum Zeitpunkt der Anlage des Datensatzes nicht zur Verfügung steht. Hierbei übernimmt *FIX* das Konzept von IBM Informix ESQL/C, einen bzgl. des Datentyps der Hostvariable[1] zulässigen, in der Praxis aber unwahrscheinlichen Wert als Kodierung für NULL zu verwenden, wodurch NULL ohne Einsatz von Indikatorvariablen[2] unterstützt werden kann. Mehr hierzu auf Seite 40.

Die folgenden Abschnitte beschreiben die Feldtypen im Detail. Bei der Darstellung werden Grundkenntnisse von IBM Informix ESQL/C vorausgesetzt.

1. Hostvariablen sind Programmvariablen, über die die Anwendung mit dem Datenbanksystem kommuniziert; es liefert die Spaltenwerte dort ab und übernimmt sie von dort.
2. Indikatorvariablen sind Programmvariablen, die gemäß dem SQL-Standard zusätzlich jeder Hostvariablen, die den Wert NULL annehmen kann, zugeordnet werden müssen; vor dem Zugriff auf den Wert der Hostvariable muss dann jeweils die Indikatorvariable inspiziert werden.

1 CHARACTER-Felder (FXCHARTYPE)

FXCHARTYPE-Felder entsprechen dem Spaltentyp CHAR(*n*), wobei *FIX* die Länge auf maximal 2047 Zeichen beschränkt. Die Felder lassen als Werte Zeichenfolgen fester Länge über dem eingestellten Zeichenvorrat zu (vgl. Kapitel *40*). Im Gegensatz zur Datenbank, bei der CHAR-Spalten sämtliche Zeichen enthalten können, erlaubt *FIX* in den Werten von CHARACTER-Feldern ausschließlich *darstellbare* Zeichen. Das Null-Zeichen (‘\0’) beendet, sofern es vorkommt, den signifikanten Teil einer Zeichenfolge; es und alle nachfolgenden Zeichen werden vom Datenbanksystem als Leerzeichen interpretiert. Folglich behandelt auch *FIX* Leerzeichen am Ende einer Zeichenfolge als *Füllzeichen*, d.h. die Zeichenfolgen “abc ” und “abc” sind für *FIX* äquivalent.

Der Wert NULL wird nur optional unterstützt.

FIX benutzt zur Speicherung der Werte von FXCHARTYPE-Feldern der Länge *n* einen Bereich von *n*+1 Bytes, wobei das letzte Byte ‘\0’ enthält. Ein abzulegender Wert wird - mit Ausnahme von NULL - ggf. mit Leerzeichen auf *n* Zeichen aufgefüllt oder auf *n* Zeichen verkürzt. Beim Vergleich von Werten sind nachgestellte Leerzeichen nicht signifikant.

Eine besondere Bedeutung hat ‘\0’ im ersten Byte. IBM Informix ESQL/C wie *FIX* verwenden dies als Kodierung von NULL. Die (C-)Strings " " und "" ergeben also verschiedene Werte: bei Ersterem handelt es sich um einen Leertext (*n* Leerzeichen), bei Letzterem um NULL. Eine leere Zeichenfolge muss also immer mindestens aus einem Leerzeichen bestehen.

FIX benutzt für die Darstellung der nicht signifikanten *Füll-Leerzeichen* in der Regel ein spezielles Zeichen des alternativen Zeichensatzes (vgl. Seite 271, Feldeigenschaft “FILLBLANK” auf Seite 45). Wenn das Feld NULL erlaubt, wird jedoch mindestens ein Leerzeichen dargestellt, um optisch zwischen einer leeren Zeichenfolge und NULL unterscheiden zu können.

Enthält der Wert eines Feldes ein unzulässiges Zeichen, wird er durch eine Folge von ‘∗’ wiedergegeben.

Bedienungsmodi

FIX verwendet bedingt durch seine Entstehungsgeschichte standardmäßig einen *Overwrite-Modus*, d.h. Zeicheneingabe manipuliert das Zeichen unter der Schreibmarke. Da dies bei grafischen Oberflächen untypisch ist, unterstützt *FIX* seit Version 3.1.0 optional einen *Insert-Modus*. Wenn er aktiv ist, wird ein eingegebenes Zeichen vor der Position der Schreibmarke eingefügt. Ist kein Platz im Eingabepuffer vorhanden, ertönt ein akustisches Signal und der Feldinhalt bleibt unverändert.

Wie der Insert-Modus aktiviert oder im Programm abgeschaltet wird, ist auf Seite 449 bzw. auf Seite 444 beschrieben.

Ist bei der Felderfassung der Insert-Modus aktiviert, besitzt die Taste CI eine andere Bedeutung: mit ihr kann dann zwischen Insert- und Overwrite-Modus hin und her geschaltet werden (vgl. aber die Ressource AllowToggleFieldInsertMode auf Seite 447).

2 Wahrheitswert-Felder (FXTRUTHTYPE)

FXTRUTHTYPE-Felder korrespondieren zum Spaltentyp CHAR(1). Es sind Felder mit optionaler Unterstützung von NULL, die, sofern sie nicht leer sind, nur die in den Variablen S_no und S_yes hinterlegten Zeichenwerte aufnehmen können (S_no ist mit ‘n’, S_yes mit ‘j’ vorbelegt). Sie dienen zur Erfassung von Wahrheitswerten: Der Wert S_no steht für “Nein”, der Wert S_yes für “Ja”, ein leeres Feld für “undefiniert”. Diese Art von Feld wird häufig als Schalter verwendet.

Die Darstellung der Werte S_no und S_yes ist mittels der Umgebungsvariable FXTRUTH konfigurierbar (vgl. Abschnitt *3* in *Anhang A*); voreingestellt sind ‘n’ und ‘j’. Besteht der Wert des Feldes aus einem unzulässigen Zeichen, wird er durch einen ‘∗’ wiedergegeben.

3 Semigrafik-Felder (FXGRAPHICSTYPE)

FXGRAPHICSTYPE-Felder korrespondieren wie FXCHARTYPE-Felder zum Spaltentyp CHAR(n), wobei die Länge ebenfalls auf maximal 2047 Zeichen beschränkt ist. Die Felder lassen als Werte Zeichencode-Folgen der Länge n über den Zeichencodes 32 bis 255 zu. Jeder Code steht für das entsprechende *FIX*-Semigrafikzeichen (vgl. Kapitel *41*). Insbesondere in Verbindung mit einem Frontend wie *FIX/Win*, das frei definierbare Grafikzeichen erlaubt, können mit ihnen “Symbol”-Felder realisiert werden, bei denen der Anwender zwar keine Eingabe vornehmen, wohl aber aus vorgegebenen Symbolen wählen kann.

FIX benutzt zur Speicherung der Werte den C-Datentyp **char[n]**, d.h. die Werte sind nicht mit ‘\0’ terminiert.

Der Wert NULL, der wie bei CHARACTER-Feldern kodiert ist, ist stets zulässig und wird durch ein vollständig mit Füll-Leerzeichen besetztes Feld wiedergegeben.

Ein unzulässiger Wert wird durch eine Folge von ‘∗’ wiedergegeben.

4 Numerische Felder

Numerische Felder können als Werte Zahlen aufnehmen. Alle numerischen Felder unterstützen den Wert NULL optional. NULL wird durch ein vollständig mit Füll-Leerzeichen besetztes Feld wiedergegeben.

Felder mit Formatangabe

Für numerische Felder können Formate aus den Formatzeichen ‘+’, ‘-’, ‘(‘, ‘)’, ‘∗’, ‘,’, ‘#’, ‘&’, ‘.’ angegeben werden (zur Bedeutung der Formatzeichen vgl. IBM Informix ESQL/C-Dokumentation). Feldlänge und Genauigkeit ergeben sich aus dem Format nach folgenden Regeln:

> Vorkommastellen = Anzahl der Formatzeichen ‘+’, ‘-’, ‘(’, ‘∗’, ‘#’, ‘&’ links vom Dezimalpunkt, vermindert um 1, sofern ‘+’, ‘-’, oder ‘(’im Format vorkommen
>
> Nachkommastellen = Anzahl der Formatzeichen ‘+’, ‘-’, ‘(’, ‘∗’, ‘#’, ‘&’ rechts vom Dezimalpunkt

Bei Formaten ohne Dezimalpunkt (Formatzeichen ‘.’) wird hierbei von einem fiktiven Dezimalpunkt am Ende des Formats ausgegangen.

Um die Position der Schreibmarke bei der Eingabe in ein Feld erkennbar zu halten und den auch für die interne Funktionsweise wichtigen Grundsatz zu gewährleisten, dass die formatierte Zahlendarstellung nie weniger Zeichen als die C-typische Notation benötigt, sind allerdings einige *zusätzliche* Einschränkungen zu beachten:

- das Format muss eine Zahlendarstellung fester Länge liefern; dies ist automatisch sichergestellt, da *FIX* das Formatzeichen ‘<’nicht unterstützt. Mindestens ein Formatzeichen muss von ‘∗’ verschieden sein.
- Die Anzahl der Vorkommastellen muss mindestens 1 betragen.
- Enthält das Format einen Dezimalpunkt (Formatzeichen ‘.’), so muss die Anzahl der Nachkommastellen mindestens 1 betragen.
- Die Summe von Vor- und Nachkommastellen darf 32 nicht übersteigen.

Formatgerecht dargestellt werden können neben NULL nur solche Werte, die sich innerhalb des durch das Format bestimmten Zahlenbereichs bewegen:

$$\text{ABS(ROUND(}\mathit{Wert}, \mathit{Nachkommastellen}\text{))} < 10^{\mathit{Vorkommastellen}}$$

Ist der Wert eines Feldes aufgrund des unzureichenden Formats nicht darstellbar, wird er durch eine Folge von ‘∗’ wiedergegeben.

Hinweis

Das Formatzeichen ‘.’ (Dezimalpunkt) wird durch das in der Umgebungsvariablen FXRADIXCHAR angegebene Zeichen (Default ‘.’) wiedergegeben, das für das Formatierzeichen ‘,’ (Tausender-Trennzeichen) verwendete Zeichen hängt von dem als Dezimalpunkt verwendeten Zeichen ab (vgl. *Anhang A*, Seite 521).

Bei Anwendungen, die auch nicht formatierte numerische Felder enthalten, die den Dezimalpunkt stets durch '.' wiedergeben, wird empfohlen, FXRADIXCHAR nicht abweichend vom Default zu setzen, um den Anwender nicht mit unterschiedlichen Dezimalpunkt-Darstellungen zu verwirren.

Felder ohne Formatangabe

Hier werden Zahlen in C-typischer Notation dargestellt. Ältere *FIX*-Versionen stellten den Wert 0 generell nicht dar, da 0 in der Praxis häufig mit "keine Angabe" gleichgesetzt wurde. Aus Kompatibilitätserwägungen wurde dieses Verhalten beibehalten, jedoch mit zwei Ausnahmen:

- Wenn ein Feld den Wert NULL unterstützt, wird der Wert 0 stets durch Ziffern dargestellt; anderenfalls wären 0 und NULL am Bildschirm nicht zu unterscheiden.
- Bei einem Feld, das den Wert NULL nicht zulässt, kann der Entwickler durch eine besondere Feldeigenschaft (vgl. "ZERO_VALUED" auf Seite 45) ebenfalls erreichen, dass 0 durch Ziffern dargestellt wird.

Bei der Darstellung werden Vor- und Nachkommastellen durch einen *Punkt* getrennt; dem Punkt geht mindestens eine Ziffer voraus. Wenn die Anzahl der Nachkommastellen nicht vorgegeben ist, werden bei linksbündiger Darstellung Nullen am Ende ab der zweiten Stelle hinter dem Dezimalpunkt abgeschnitten, anderenfalls bleiben sie erhalten.

4.1 FXSHORTTYPE

FXSHORTTYPE-Felder entsprechen dem IBM Informix-Spaltentyp SMALLINT. Der Wertebereich umfasst die ganzen Zahlen von -32767 bis +32767 und optional NULL. *FIX* benutzt zur Speicherung der Werte den C-Datentyp **short**.

Felder ohne Formatangabe

Ist der Wert eines Feldes aufgrund der zu geringen Feldlänge nicht darstellbar, wird er durch eine Folge von '*' wiedergegeben.

4.2 FXLONGTYPE

FXLONGTYPE-Felder entsprechen dem IBM Informix-Spaltentyp INTEGER oder SERIAL. Der Wertebereich umfasst die ganzen Zahlen von -2147483647 bis +2147483647 und optional NULL. *FIX* benutzt zur Speicherung der Werte den C-Datentyp **long**.

Felder ohne Formatangabe

Ist der Wert eines Feldes aufgrund der zu geringen Feldlänge nicht darstellbar, wird er durch eine Folge von '*' wiedergegeben.

4.3 FXFLOATTYPE

FXFLOATTYPE-Felder entsprechen dem IBM Informix-Spaltentyp SMALLFLOAT. Der Wertebereich umfasst einfach genaue Gleitkommazahlen (ca. 7 exakte Ziffern, Intervall plattformabhängig) und optional NULL. *FIX* benutzt zur Speicherung der Werte den C-Datentyp **float**.

Felder ohne Formatangabe

Es werden 0 bis 9 Nachkommastellen (Voreinstellung: 3) unterstützt. Ist der Wert eines Feldes aufgrund der zu geringen Feldlänge nicht darstellbar, wird zunächst versucht, (ggf. bis zur nächsten ganzen Zahl) zu runden, reicht die Feldlänge auch dafür nicht aus, wird er durch eine Folge von '*' wiedergegeben.

4.4 FXDOUBLETYPE

FXDOUBLETYPE-Felder entsprechen dem IBM Informix-Spaltentyp FLOAT. Der Wertebereich umfasst doppelt genaue Gleitkommazahlen (ca. 14 exakte Ziffern, Intervall plattformabhängig) und optional NULL. *FIX* benutzt zur Speicherung der Werte den C-Datentyp **double**.

Felder ohne Formatangabe

Es werden 0 bis 9 Nachkommastellen (Voreinstellung: 3) unterstützt. Ist der Wert eines Feldes aufgrund der zu geringen Feldlänge nicht darstellbar, wird zunächst versucht, (ggf. bis zur nächsten ganzen Zahl) zu runden, reicht die Feldlänge auch dafür nicht aus, wird er durch eine Folge von ‘∗’ wiedergegeben.

4.5 FXMONEYTYPE

FXMONEYTYPE-Felder benutzt *FIX* für den IBM Informix-Spaltentyp MONEY(*p*) mit $p \leq 32$. Der Wertebereich umfasst doppelt genaue Gleitkommazahlen (ca. 14 exakte Ziffern, Intervall plattformabhängig) und optional NULL. Die Verwendung des C-Datentyps **double** dient zur Wahrung der Kompatibilität mit früheren Versionen.

Felder ohne Formatangabe

Die Darstellung beinhaltet kein Währungszeichen und erfolgt rechtsbündig mit genau 2 Nachkommastellen. Ist der Wert eines Feldes aufgrund der zu geringen Feldlänge nicht darstellbar, wird er durch eine Folge von ‘∗’ wiedergegeben.

4.6 FXDECIMALTYPE

FXDECIMALTYPE-Felder entsprechen dem IBM Informix-Spaltentyp DECIMAL(*p*) mit $p \leq 32$. Der Wertebereich umfasst Gleitkommazahlen mit max. 32 signifikanten Ziffern und Absolutbetrag zwischen 10^{-128} und 10^{126}. *FIX* benutzt zur Speicherung der Werte den C-Datentyp **dec_t**.

Felder ohne Formatangabe

Es werden 0 bis 9 Nachkommastellen (Voreinstellung: 3) unterstützt. Die Darstellung erfolgt rechtsbündig. Ist der Wert eines Feldes aufgrund der zu geringen Feldlänge nicht darstellbar, wird zunächst versucht, (ggf. bis zur nächsten ganzen Zahl) zu runden, reicht die Feldlänge auch dafür nicht aus, wird er durch eine Folge von ‘∗’ wiedergegeben.

5 Datum-Felder (FXDATETYPE)

FXDATETYPE-Felder entsprechen dem IBM Informix-Spaltentyp DATE, d.h. sie können einen bestimmten Tag im Zeitraum 01.01.0001 bis 31.12.9999 beschreiben. Der Wert NULL ist stets zulässig und wird durch ein vollständig mit Füll-Leerzeichen besetztes Feld wiedergegeben.

Analog zu IBM Informix wird ein Datum im C-Datentyp **long** gespeichert als Abstand in Tagen zum 31.12.1899. So sind, neben der Kodierung des Wertes NULL (-2147483648), lediglich Werte zwischen -693594 und 2958464 sinnvoll.

FIX unterstützt verschiedene Darstellungen für Datumswerte, die - in Verbindung mit dem Wert der Umgebungsvariable FXDATE (vgl. Abschnitt “Datum (FXDATETYPE)” auf Seite 521) - aus der Feldlänge abgeleitet werden:

- Länge 10 - Tag, Monat und vierstelliges Jahr mit Trennzeichen (gewöhnlich “dd.mm.yyyy”)
  Diese Felder können alle legalen Datumswerte darstellen.

- Länge 7 - Monat und vierstelliges Jahr mit Trennzeichen (gewöhnlich “mm/yyyy”)
  Diese Felder können nur solche Datumswerte darstellen, die dem ersten (bzw. in Verbindung mit der Feldeigenschaft UPPER, dem letzten) Tag eines Monats entsprechen (Periode).
- Länge 8 - Tag, Monat und zweistelliges Jahr mit Trennzeichen (gewöhnlich “dd.mm.yy”)
  Diese Felder können nur solche Datumswerte darstellen, die im aktuellen 100-Jahre-Zeitraum[1] liegen.
- Länge 6 - Tag, Monat und zweistelliges Jahr ohne Trennzeichen (gewöhnlich “ddmmyy”)
  Diese Felder können nur solche Datumswerte darstellen, die im aktuellen 100-Jahre-Zeitraum[1] liegen.
- Länge 5 - Monat und zweistelliges Jahr mit Trennzeichen (gewöhnlich “mm/yy”)
  Diese Felder können nur solche Datumswerte darstellen, die dem ersten (bzw. in Verbindung mit der Feldeigenschaft UPPER, dem letzten) Tag eines Monats entsprechen und im aktuellen 100-Jahre-Zeitraum[1] liegen (Periode).

Ein Wert, der den obigen Bedingungen nicht genügt, wird durch eine Folge von ‘*’ wiedergegeben.

Bei der Konversion von der textuellen Darstellung in einen Wert sollte beachtet werden, dass

- für dd, mm, yy 1 oder 2 Ziffern, für yyyy 1 bis 4 Ziffern angegeben werden können,
- eine zweistellige Jahresangabe auf den aktuellen 100-Jahre-Zeitraum bezogen wird,
- bei Formaten ohne Tageskomponente (Längen 5 und 7) das Datum des ersten (bzw. in Verbindung mit der Feldeigenschaft UPPER, des letzten) Tages des angegebenen Monats und Jahres hinterlegt wird,
- der Wert NULL hinterlegt wird, wenn die Darstellung ausschließlich Leerzeichen enthält.

Übersicht

Feldlänge	kleinster Wert	größter Wert	
10	01.01.0001	31.12.9999	
7	01.01.0001	01.12.9999	sofern erster Tag des Monats
7, UPPER	31.01.0001	31.12.9999	sofern letzter Tag des Monats
8	01.01.1969	31.12.2068	[2]
6	01.01.1969	31.12.2068	
5	01.01.1969	01.12.2068	sofern erster Tag des Monats
5, UPPER	31.01.1969	31.12.2068	sofern letzter Tag des Monats

Beispiel

Feldlänge	Wert = 21069	Wert = 33662	Wert = 36525
10	04.10.1957	29.02.1992	01.01.2000
7	*******	*******	01/2000
7, UPPER	*******	02/1992	*******
8	********	29.02.92	01.01.00
6	******	290292	010100
5	*****	*****	01/00
5, UPPER	*****	02/92	*****

1. Der aktuelle 100-Jahre-Zeitraum ist, wenn nicht anders definiert, der Zeitraum vom 01.01.1969 bis 31.12.2068. Siehe hierzu auch Seite 446.
2. bei Standardwert für den 100-Jahre-Zeitraum

6 Zeitpunkt-Felder (FXDTIMETYPE)

FXDTIMETYPE-Felder entsprechen dem IBM Informix-Spaltentyp DATETIME … TO …. Der Wertebereich sind Zeitpunkte. Ein Zeitpunkt wird durch Angabe von Jahr, Monat, Tag, Stunde, Minute, Sekunde und Sekundenbruchteil bzw. eines zusammenhängenden Ausschnitts dieser Komponenten beschrieben. Zur Speicherung von DATETIME-Werten benutzt *FIX* den C-Datentyp **dtime_t**. Dieser beinhaltet einen *Qualifier*, der die zum Wert beitragenden Komponenten spezifiziert.

Ein FXDTIMETYPE-Feld kann allerdings nur solche Zeitpunkte als Wert annehmen, die ausschließlich jene Komponenten umfassen, die in seinem *Format* enthalten sind. Das Format setzt sich aus Formatsequenzen für die einzelnen Komponenten zusammen. Zwischen den Formatsequenzen kann maximal ein Trennzeichen angegeben werden.

Komponente	Formatsequenz	mögliche Werte
Jahr	yyyy bzw. yy	0001 - 9999 bzw. zweistellige Jahreszahl im aktuellen 100-Jahre-Zeitraum
Monat	mm	01 - 12
Tag	dd	01 - 31
Stunde	hh	00 - 23
Minute	nn	00 - 59
Sekunde	ss	00 - 59

Sekundenbruchteile werden nicht unterstützt.

Zu je zwei enthaltenen Komponenten müssen auch alle dazwischenliegenden Komponenten vorkommen, und jede Komponente darf höchstens einmal vorkommen.

Wird eines der Formatzeichen statt in Klein- in Großschrift angegeben, so wird die betreffende Komponente mit führenden Nullen dargestellt, anderenfalls nicht. Als Trennzeichen kann jedes Zeichen mit Ausnahme von ‘y’, ‘m’, ‘d’, ‘h’, ‘n’ und ‘s’ (in Groß- oder Kleinschrift) verwendet werden.

Der Wert NULL ist stets zulässig und wird durch ein vollständig mit Füll-Leerzeichen besetztes Feld wiedergegeben.

Ein unzulässiger Wert wird durch eine Folge von ‘∗’ wiedergegeben.

7 Zeitspannen-Felder (FXINVTYPE)

FXINVTYPE-Felder entsprechen dem IBM Informix-Spaltentyp INTERVAL … TO …. Der Wertebereich umfasst vorzeichenbehaftete Zeitspannen. Eine Zeitspanne wird entweder durch eine Anzahl von Jahren und Monaten oder durch eine Anzahl von Tagen, Stunden, Minuten, Sekunden und Sekundenbruchteilen bzw. durch einen zusammenhängenden Ausschnitt der jeweiligen Komponenten beschrieben. Zur Speicherung von INTERVAL-Werten benutzt *FIX* den C-Datentyp **intrvl_t**. Dieser beinhaltet einen *Qualifier*, der die zum Wert beitragenden Komponenten spezifiziert.

Ein FXINVTYPE-Feld kann allerdings nur solche Zeitspannen als Wert annehmen, die ausschließlich jene Komponenten umfassen, die in seinem *Format* enthalten sind. Das Format setzt sich aus Formatsequenzen für die einzelnen Komponenten zusammen, denen optional ein ‘+’ oder ‘-’ vorausgehen kann. Die Sequenzen müssen absteigend geordnet sein. Während die Sequenz an erster Stelle aus ein bis neun Formatzeichen bestehen kann, müssen alle weiteren aus exakt zwei Formatzeichen bestehen. Zwischen den Sequenzen kann optional ein Trennzeichen angegeben werden.

Komponenten	Formatsequenzen		mögliche Werte,
(absteigend geordnet)	an 1. Pos.	an folg. Pos.	wenn nicht erste Komponente
Anzahl Jahre	y…y		
Anzahl Monate	m…m	mm	00 - 11

Anzahl Tage	d…d		
Anzahl Stunden	h…h	hh	00 - 23
Anzahl Minute	n…n	nn	00 - 59
Anzahl Sekunden	s…s	ss	00 - 59

Sekundenbruchteile werden nicht unterstützt.

Zu je zwei enthaltenen Komponenten müssen auch alle dazwischenliegenden Komponenten vorkommen, und jede Komponente darf höchstens einmal vorkommen.

Wird eines der Formatzeichen statt in Klein- in Großschrift angegeben, so wird die betreffende Komponente mit führenden Nullen dargestellt, anderenfalls nicht. Als Trennzeichen kann jedes Zeichen mit Ausnahme von ‘y’, ‘m’, ‘d’, ‘h’, ‘n’ und ‘s’ (in Groß- oder Kleinschrift) verwendet werden.

Beginnt das Format mit dem Formatzeichen ‘+’, wird an dieser Position stets das Vorzeichen des Wertes (‘+’, ‘-’) angezeigt, bei dem Formatzeichen ‘-’ wird bei einem positiven Wert statt ‘+’ ein ‘ ’ verwendet. Ist keines der beiden Formatzeichen angegeben, unterstützt das Feld nur vorzeichenlose Werte.

Der Wert NULL ist stets zulässig und wird durch ein vollständig mit Füll-Leerzeichen besetztes Feld wiedergegeben.

Ein unzulässiger Wert wird durch eine Folge von ‘∗’ wiedergegeben.

8 NULL-Werte

8.1 NULL-Werte bei IBM Informix ESQL/C

IBM Informix unterstützte bis Version 5.x keine expliziten Defaultwerte für Tabellenspalten und belegte beim Anlegen von Sätzen Spalten, für die kein Wert angegeben wurde, mit NULL; wenn eine Spalte den Wert NULL nicht zuließ, war die Angabe eines Wertes zwingend erforderlich. Letzteres hatte zur Folge, dass die Eigenschaft ‘NOT NULL’ von Entwicklern nur ungern vergeben wurde.

IBM Informix ESQL/C kann den Wert NULL nicht nur - dem SQL-Standard entsprechend - durch eine zusätzliche Indikatorvariable ausdrücken, sondern auch in der eigentlichen Hostvariablen; hierzu ist ein hinsichtlich der Wirtssprache C zulässiger Wert der Variablen als Darstellung von NULL festgelegt (vgl. Seite 33). Mittels zweier Funktionen der ESQL/C-Library (risnull() bzw. rsetnull()) kann eine Hostvariable auf NULL abgefragt oder ihr NULL zugewiesen werden.

Das Lesen von Spalten, die NULL enthalten können, ist auch ohne Indikatorvariablen möglich. Die Hostvariable enthält, ist der Spaltenwert NULL, anschließend einen Wert, für den risnull() das Ergebnis 1 (d.h. ‘gleich NULL’) liefert. Irgendwelche Operationen auf dem Wert sind in diesem Fall nicht sinnvoll.

Allerdings wird diese Kodierung von NULL im Bedingungsteil von SQL-Anweisungen (where-Klausel) nicht als NULL behandelt. Bei der Suche nach NULL müssen die Formulierungen ‘*column* IS NULL’ oder ‘*column* IS NOT NULL’ verwendet werden, ‘*column* = :*var*’ mit einer Hostvariable, die den Wert NULL enthält, funktioniert nicht.

8.2 NULL-Werte bei *FIX*

Bei älteren Versionen erlaubte *FIX* den Wert NULL nur bei Datum-Feldern und setzte bei allen anderen Feldtypen stillschweigend voraus, dass eine Anwendung nicht zwischen NULL und der Zahl 0 bzw. einem Leerstring differenzieren wollte, genauer, Letztere wurden in der logischen Bedeutung von NULL behandelt. Beispielsweise wurden numerische Felder generell mit 0 vorbelegt und der Wert 0 wurde, da als “nicht gehaltvoll” betrachtet, nicht angezeigt. Aus der Datenbank gelesene NULL-Werte wurden beim Ablauf des Programms u.U. in 0 bzw. “ ” umgesetzt.

Obwohl diese Praxis von Anwendungen wohl weiterhin häufig beibehalten werden wird, unterstützt *FIX* NULL seit längerem in einer Weise, die einen guten Kompromiss hinsichtlich der Kompatibilität zu älteren Anwendungen und den Anforderungen neuerer Anwendungen bildet. Der Bedeutung wegen sind in den folgenden Abschnitten die mit NULL verbundenen Probleme noch einmal kompakt zusammengestellt.

CHARACTER-, Wahrheitswert- und numerische Felder können optional die Eigenschaft “Wert kann NULL sein” erhalten.

Durch die nur optionale NULL-Unterstützung bei Feldern wird bezweckt, dass bestehende Anwendungen weitgehend ohne Modifikationen auch mit neueren *FIX*-Versionen ablaufen können.

Bei Feldern, die NULL nicht unterstützen, bleibt ein aus der Datenbank in die Hostvariable gelesener NULL-Wert wie bisher, solange ihn *FIX* nicht als Feldwert bearbeitet, erhalten; spätestens beim ersten wrktoinfo() wird er konvertiert.

Wird andererseits NULL als Wert zugelassen, hat das folgende Konsequenzen:

- Ein Löschen des Feldes (interaktiv oder per Programm) wird als Zuweisung von NULL behandelt.
- Das Feld gilt nur dann als leer, wenn es den Wert NULL enthält.
- Bei der Konversion Zeichenkette → Wert ergibt ein Leerstring (“”) NULL. Insbesondere werden Felder ohne expliziten Defaultwert mit NULL vorbelegt. Bei FXTRUTHTYPE-Feldern wird “ ” (ein Leerzeichen), bei FXDATETYPE-, FXDTIMETYPE und FXINVTYPE-Feldern eine nur aus Leerzeichen bestehende Zeichenkette wie “” behandelt.
- Bei numerischen Feldern wird NULL durch ein vollständig mit Füllzeichen besetztes Feld dargestellt, 0 immer durch Ziffern.

  Bei CHARACTER- und Wahrheitswert-Feldern wird NULL durch ein vollständig mit Füllzeichen besetztes Feld dargestellt, eine leere Zeichenfolge durch mindestens ein Leerzeichen.
- Ein Feld, das NULL-Werte zulässt, verliert diese Eigenschaft, solange es zur Erfassung von Parametern für die Selosuche benutzt wird; dies gilt nicht für Datum- und Zeitpunkt-Felder.[1]

Bei der Selo-Auswahl bleiben NULL-Werte im ausgewählten Satz im Selo-eigenen Puffer erhalten.

- Erfolgt die Übernahme eines Spaltenwertes in ein Feld, das NULL-Werte nicht zulässt, durch *FIX* (d.h. aufgrund einer Angabe in der Beschreibungsdatei, durch Aufruf von fputselo()), so wird ein angelieferter NULL-Wert automatisch nach “ ” bzw. 0 konvertiert.

  Die Routine fx_selo_fld_adr() hingegen gibt ggf. einen Zeiger auf eine Speicherfläche zurück, an der der Wert NULL steht; hier muss die Anwendung die Übernahme in ein Feld selbst geeignet gestalten.
- NULL-Werte werden im Selo durch eine formatabhängige Anzahl von Leerzeichen dargestellt.

Beim Vergleich von Werten werden NULL-Werte als zueinander gleich und kleiner als alle anderen Werte behandelt.

Beim Erstellen von Satzmengen bleiben NULL-Werte zunächst erhalten.

Benutzt eine Anwendung Felder mit NULL-Unterstützung, so ist zu beachten:

- Arithmetische oder stringverarbeitende Operationen können nur dann auf Feld- bzw. Hostvariablenwerte angewendet werden, wenn diese ungleich NULL sind.
- Felder oder Hostvariablen mit Wert NULL sind in where-Bedingungen von SQL-Anweisungen nicht zulässig.

1. Die würde genauso für FXGRAPHICSTYPE-Felder gelten, die jedoch sowieso nicht als Parameter bei der Selosuche in Frage kommen.

Zum Test, ob der Wert eines Feldes bzw. einer Variablen NULL ist, bietet *FIX* die Funktionen f_isnull() und fxisnull() an.

9 Besonderheiten bei der Felderfassung

Felder mit obligatorischer Choice

Hier kann der Feldwert nur durch eine Auswahl mittels einer Choice verändert werden, das Feld selbst wird als nicht betretbar behandelt. Mehr hierzu auf Seite 208.

Felder mit Wertvorgabe

Hier kann keine direkte Eingabe erfolgen. Vielmehr kann der Anwender mit der Leertaste und der Taste '+' vorwärts, mit der Taste '-' rückwärts durch vom Entwickler vorgegebene Werte blättern. Buchstaben oder Ziffern blättern - mit Ausnahme von Feldern mit Formatangabe und Semigrafik-Feldern - zum nächsten Wert, der mit diesem Zeichen beginnt, wobei nicht zwischen Groß- und Kleinbuchstaben unterschieden wird.

CHARACTER-Felder

Unterstützt das Feld den Wert NULL nicht, wird ein leeres Feld behandelt wie eines, das ein Leerzeichen enthält.

Wahrheitswert-Felder

Unterstützt das Feld den Wert NULL nicht, wird ein leeres Feld behandelt wie eines, das ein Leerzeichen enthält.

Der Anwender kann nur die Zeichen 'n' und 'j' (bzw. die in der Umgebungsvariable FXTRUTH spezifizierten Kleinbuchstaben), die dazu korrespondierenden Großbuchstaben (sie werden automatisch in Kleinbuchstaben konvertiert) und ' ' eingeben. 'n' ergibt den in S_no, 'j' den in S_yes hinterlegten Wert, ' ' entspricht einem Löschen des Feldes.

Semigrafik-Felder

Hier kann keine direkte Eingabe erfolgen. Existiert eine Wertvorgabe, verhalten sich die Felder wie oben beschrieben, ansonsten wie ein Feld mit der Eigenschaft NOMOD (vgl. "Feldeigenschaften" auf Seite 44).

Numerische Felder

Unterstützt das Feld den Wert NULL nicht, behandelt *FIX* ein leeres Feld wie eines mit dem Wert 0.

Felder mit Formatangabe

Für die Bearbeitung wird der numerische Wert in eine interne Repräsentation "entpackt", die eine einfache Manipulation und formatierte Darstellung ermöglicht. Es werden zwei Erfassungsmodi unterschieden.

Im ersten, der bei FXMONEYTYPE-Feldern zum Einsatz kommt (vgl. aber B_numeric_logic auf Seite 440), beginnt die Felderfassung am rechten Rand (Schreibmarke rechts neben dem Feld). Bei Eingabe einer Ziffer werden die bereits vorhandenen - evtl. über den Dezimalpunkt - nach links geschoben und die Ziffer wird an der freiwerdenden Position eingefügt. Beim Drücken der Taste '.' (oder ',') werden alle Ziffern vor den Dezimalpunkt geschoben und die Schreibmarke wechselt zur ersten Nachkommastelle. Befindet sich die Schreibmarke nicht rechts vom Feld, so wird bei Eingabe einer Ziffer das Zeichen unter der Schreibmarke überschrieben und die Schreibmarke nach rechts versetzt.

Im zweiten Modus, der bei allen anderen Feldtypen benutzt wird (vgl. aber B_numeric_logic auf Seite 440), beginnt die Felderfassung auf der Einer-Position. Bei Eingabe einer Ziffer werden die Ziffern links neben und an der aktuellen Position nach links geschoben und die Ziffer an der aktuellen Position eingefügt; die Ziffern zur Rechten bleiben erhalten. Beim Drücken der Taste '.' oder ',' wechselt die Schreibmarke zur ersten Nachkommastelle (nicht bei FXSHORTTYPE- und FXLONGTYPE-Feldern). Befindet sich die Schreibmarke über einer Nachkommastelle, so wird bei Eingabe einer Ziffer das Zeichen unter der Schreibmarke überschrieben und die Schreibmarke nach rechts versetzt.

Ein Vorzeichenwechsel durch Drücken der Taste '-' ist nur möglich, wenn das Format des Feldes ein Vorzeichen beinhaltet (vgl. aber auch "UPPER" auf Seite 44).

DL löscht das Zeichen links neben, CD das Zeichen unter der Schreibmarke. Die Tasten WD, CI, BT und TB werden nicht unterstützt und lösen lediglich ein akustisches Signal aus.

Felder ohne Formatangabe

Hier können nur Ziffern und, bei Feldern mit Typ FXFLOATTYPE, FXDOUBLETYPE, FXMONEYTYPE und FXDECIMALTYPE, ein Dezimalpunkt eingegeben werden. Zur Eingabe des Dezimalpunktes können die Tasten '.' und ',' benutzt werden (dargestellt wird aber in jedem Fall ein Punkt). Die Taste '-' vertauscht das Vorzeichen.

Enthält ein FXMONEYTYPE-Feld keinen Dezimalpunkt, fasst *FIX* den Wert als Pfennigbetrag auf und setzt beim Verlassen des Feldes den Dezimalpunkt selbständig (siehe aber Programmschalter -DM).

Führende Nullen werden bei der Darstellung nicht unterstützt.

FXDATETYPE-Felder

Bei der Eingabe können zur Abgrenzung der Bestandteile außer '.' alle Satzzeichen des Zeichensatzes ISO 646:1983 verwendet werden; dargestellt werden abhängig von der Feldlänge '.' und
'/' (bzw. die in der Umgebungsvariable FXDATE spezifizierten Trennzeichen).

Eine nur aus Trennzeichen bestehende Eingabe ersetzt *FIX* bei Verlassen des Feldes durch das aktuelle Datum; nur bei der Komponentenabfolge Tag-Monat-Jahr wird ein unvollständiges Datum mit dem aktuellen Monat und/oder Jahr aufgefüllt.

FXDATETYPE-Felder unterstützen stets den Wert NULL.

FXDTIMETYPE-Felder

Die Eingabe in ein FXDTIMETYPE-Feld erfolgt komponentenweise. Bei Betreten des Feldes steht die Schreibmarke am linken Rand der ersten Komponente. Der Inhalt der aktuellen Komponente kann (von links nach rechts fortschreitend) erfasst oder manipuliert werden.

Ein Wechsel zur nächsten Komponente findet automatisch statt, wenn die letzte Position einer Komponente beschrieben wird. Daneben kann durch Eingabe eines im Format enthaltenen Trennzeichens in die unmittelbar rechts von diesem Trennzeichen liegende Komponente gewechselt werden. Voraussetzung für einem Komponentenwechsel ist in beiden Fällen, dass die aktuelle Komponente einen gültigen Inhalt hat.

Wie bei FXDATETYPE-Feldern wird eine zweistellige Angabe bei einer vierstelligen Jahres-Formatsequenz auf den aktuellen 100-Jahre-Zeitraum bezogen.

FXDTIMETYPE-Felder unterstützen stets den Wert NULL.

FXINVTYPE-Felder

Die Eingabe in ein FXINVTYPE-Feld erfolgt komponentenweise. Bei Betreten des Feldes steht die Schreibmarke am rechten Rand der ersten Komponente. Der Inhalt der aktuellen Komponente kann (bei der ersten Komponente nach links schiebend, sonst von links nach rechts fortschreitend) erfasst oder manipuliert werden.

Ein Wechsel zur nächsten Komponente findet automatisch statt, wenn, bei der ersten Komponente, die Komponente voll ist, bzw. wenn, bei den weiteren Komponenten, die letzte Position der Komponente beschrieben wird. Daneben kann durch Eingabe eines im Format enthaltenen Trennzeichens in die unmittelbar rechts von diesem Trennzeichen lie-

gende Komponente gewechselt werden. Voraussetzung für einem Komponentenwechsel ist in beiden Fällen, dass die aktuelle Komponente einen gültigen Inhalt hat.

Die Taste '-' bewirkt nur dann einen Vorzeichenwechsel, wenn rechts von der aktuellen Komponente als Trennzeichen kein Bindestrich mehr vorkommt. Voraussetzung ist ferner, dass das Format des Feldes ein Vorzeichen beinhaltet (vgl. aber auch "UPPER" auf Seite 44).

FXINVTYPE-Felder unterstützen stets den Wert NULL.

10 Feldeigenschaften

FIX kennt folgende Feldeigenschaften:

NODISPL

Das Feld wird nicht dargestellt und kann nicht betreten werden.

SPECIAL

Die Feldbegrenzer werden nicht dargestellt.

INVISIBLE

Der Feldinhalt wird durch Leerzeichen dargestellt und bleibt so verborgen.

RGTADJ

Der Feldinhalt wird rechtsbündig ausgerichtet.

Bei Feldern mit Formatangabe ist diese Eigenschaft unzulässig.

NOENT

Das Feld kann nicht betreten werden.

NOMOD

Der Feldwert kann nicht verändert werden.

BEHIND

Bei Betreten des Feldes wird die Schreibmarke hinter das letzte Nicht-Leerzeichen platziert.

Bei Feldern mit Formatangabe ist diese Eigenschaft unzulässig.

ASCIIONLY

CHARACTER-Felder: bei der Eingabe werden nur ASCII-Zeichen akzeptiert.

UPPER

typabhängig:
CHARACTER-Felder: bei der Eingabe werden Buchstaben in Großschreibung umgewandelt.
Numerische Felder, FXINVTYPE-Felder: das Vorzeichen kann nicht gewechselt werden.[1]
FXDATETYPE-Felder ohne Tag-Bestandteil: der Wert repräsentiert statt des Ersten den Letzten des Monats.

FIRSTUP

CHARACTER-Felder: bei der Eingabe wird ein Buchstabe am Anfang des Feldinhalts in Großschreibung umgewandelt.

LOWER

CHARACTER-Felder: bei der Eingabe werden Buchstaben in Kleinschreibung umgewandelt.

FILLBLANK

CHARACTER-Felder: nicht signifikante Leerzeichen am Ende werden gewöhnlich durch Füll-Leerzeichen wiedergegeben. Mit der Eigenschaft FILLBLANK kann erreicht werden, dass stattdessen ein gewöhnliches Leerzeichen benutzt wird. Dies ist eine rein optische Eigenschaft; eine solche Stelle im Feld ist nach wie vor nicht signifikant.

Auf die Darstellung des Wertes NULL hat diese Eigenschaft keinen Einfluss: hier wird die Defaultdarstellung benutzt, da anderenfalls NULL nicht von einem Leertext zu unterscheiden wäre.

ZERO_VALUED

Numerische Felder: Bei Feldern ohne Formatangabe, die NULL nicht unterstützen, wird durch die Feldeigenschaft ZERO_VALUED erreicht, dass, wenn die C-Variable eines Feldes den Wert 0 besitzt, als dessen textuelle Darstellung “0” (typgerecht formatiert) produziert wird.[1] Enthält ein solches Feld bei Abschluss der Felderfassung nur Füllzeichen, so wird, *bevor* weitere Prüfungen stattfinden, “0” (typgerecht formatiert) hineingeschrieben. Daher ist die Eigenschaft REQUIRED stets erfüllt.

Bei Feldern mit Formatangabe ist diese Eigenschaft unzulässig.

AUNXT

Das Feld wird beim Beschreiben der letzten Position automatisch verlassen.

Bei Feldern mit Formatangabe ist diese Eigenschaft unzulässig.

VERIFY

Beim Versuch, das Feld zu verlassen, wird der Feldinhalt gelöscht und muss, damit das Feld verlassen wird, ein zweites Mal eingegeben werden. Dies ist allerdings nur dann nötig, wenn sich irgendwann zwischen Betreten und Verlassen des Feldes aufgrund einer Eingabe des Anwenders der Inhalt des Feldes verändert hat.

REQUIRED

Das Feld kann nicht leer verlassen werden.

Ein numerisches Feld mit Formatangabe gilt als leer, wenn - bei NULL-Unterstützung - die Darstellung nur aus Füll-Leerzeichen besteht bzw. wenn - ohne NULL-Unterstützung - keine Ziffernposition eine von 0 verschiedene Ziffer enthält.

DBMUST

Der Feldinhalt muss in der Datenmenge des zugeordneten Auswahl-Objektes (Selo, Choice) enthalten sein. Mehr hierzu auf Seite 214.

1. Diese Eigenschaft wird manchmal genutzt, um die Eingabe negativer Werte in leere Felder zu verhindern.

1. Gegenwärtig “erben” Feld-Links sowie die korrespondierenden Felder aus Varianten diese Eigenschaft, da sie nur eine gemeinsame info-Komponente besitzen.

COPY

Der vormalige Feldinhalt wird in neu angelegte Sätze übernommen.

NOJUMP

Das Feld kann im Turbomodus nicht übersprungen werden.

Einige Eigenschaften werden nicht bei allen Feldtypen unterstützt bzw. werden von *FIX* automatisch vergeben; bei Feldern mit Formatangabe sind in der nachfolgenden Übersicht aufgeführten Eigenschaften nicht zulässig bzw. ergeben sich aus dem Format (UPPER):

	CHAR	TRUTH	GRAPH-ICS	SHORT/ LONG	FLOAT/ DOU-BLE	MONEY	DECI-MAL	DATE
AUNXT		auto	nein			nein	nein	
BEHIND		nein	nein			nein	nein	
RGTADJ	nein	nein	nein			auto	auto	nein
ASCIIONLY		nein	nein	nein	nein	nein	nein	nein
LOWER		auto	nein	nein	nein	nein	nein	nein
UPPER		nein	nein					nur Peri-ode
FIRSTUP		nein	nein	nein	nein	nein	nein	nein
FILLBLANK			nein	nein	nein	nein	nein	nein
ZERO_VALUED	nein	nein	nein	NOT NULL	NOT NULL	NOT NULL	NOT NULL	nein

Die Eigenschaften SPECIAL, INVISIBLE, NOMOD und COPY werden bei allen Feldtypen unterstützt. Einzige Ausnahme bildet die Benutzung eines Feldes zur Parametererfassung bei der Selosuche (vgl. Seite 26), bei der *FIX* folgende Feldeigenschaften ausblendet: NODISPL, NOENT, NOMOD, NOJUMP, INVISIBLE, AUNXT, DBMUST, VERIFY, COPY, REQUIRED.[1]

Bei der Vergabe von Feldeigenschaften sind folgende Regeln zu beachten:

- Ein Feld mit der Eigenschaft NODISPL erhält beim Laden der Maske automatisch auch die Eigenschaft NOENT.[2]
- Sofern ein Feld die Eigenschaft NOENT besitzt, werden die Eigenschaften NOMOD, NOJUMP, REQUIRED, DBMUST sowie UPPER (Ausnahme: DATE), LOWER, FIRSTUP, VERIFY und AUNXT ignoriert.
- Sofern ein Feld die Eigenschaft NOMOD besitzt, werden die erfassungsbezogenen Eigenschaften UPPER (Ausnahme: FXDATETYPE-Felder), LOWER, FIRSTUP, VERIFY und AUNXT ignoriert.
- Die Eigenschaften UPPER, LOWER und FIRSTUP schließen sich paarweise aus.
- Die Eigenschaft RGTADJ ist nicht mit den Eigenschaften AUNXT und BEHIND kombinierbar.
- Bei einem numerischen Feld mit der Eigenschaft ZERO_VALUED macht die Eigenschaft REQUIRED keinen Sinn.
- Die Feldeigenschaften NOMOD, UPPER, LOWER, FIRSTUP sowie AUTONXT und VERIFY berücksichtigt *FIX* nur bei Einzelzeichen-Eingabe in ein Feld, nicht bei der Übernahme vollständiger Werte (Ausnahme: bei Feldern mit Wertvorgabe verhindert die Eigenschaft NOMOD die Wertauswahl durch Blättern).

1. Außer bei den Typen FXDATETYPE und FXDTIMETYPE verlieren solche Felder außerdem die Eigenschaft, NULL zu unterstützen; numerische Felder erhalten dabei die Eigenschaft ZERO_VALUED.

2. Wird einem Feld *programmgesteuert* die Eigenschaft NODISPL gegeben, so muss dies zusammen mit der Eigenschaft NOENT geschehen.

8 Paintareas

Eine Paintarea ist ein rechteckiger Bereich im Layout eines *FIX*-Objektes, der von *FIX/Win* oder der Benutzerbibliothek von *FIX/Win* durch eigene Zeichenoperationen gefüllt wird. Je nach Typ der Paintarea stellt sie ein Textlabel, einen Tabellenkopf, eine Bitmap oder einen Button dar. Paintareas sind nur in Verbindung mit *FIX/Win* nutzbar. Startet ein *FIX*-Programm im Terminal, so sind definierte Paintareas nicht sichtbar bzw. nutzbar. Paintareas können verschiedene Typen haben, um unterschiedliche Darstellungsformen oder Funktionalitäten zu erzielen.

FIX stellt hierbei folgende Paintarea-Typen zur Verfügung:

- TEXTLABEL - ein Textlabel.
- TABLEHEADER - ein Text als Kopf einer Tabellenspalte.
- BITMAP - eine Bitmap. Dieser Wert gilt als reserviert. Bitmaps werden von *FIX/Win* zur Zeit noch nicht unterstützt.
- BUTTON - ein Button.
- VARBUTTON- ein Button zur Umschaltung von Varianten.
- PA_USERDEFINED - benutzerdefiniert. Alle Paintareas, die diesen Wert oder einen größeren Wert benutzen, werden von *FIX/Win* nicht dargestellt. Die Darstellung bleibt der Benutzerbibliothek von *FIX/Win* überlassen.

Paintareas werden von *FIX/Win* oder von Funktionen der anwendungsspezifischen Bibliothek gezeichnet. Daten der Paintarea werden dazu an die Funktionen übermittelt. Es ist der anwendungsspezifischen Funktion vorbehalten, eine Paintarea selbst zu malen oder die *FIX/Win*-interne Funktion diese Aufgabe erledigen zu lassen.

Für alle Paintarea-Typen stehen spezielle Farbattribute in der *FIX/Win*-Farbtabelle zur Verfügung, mit denen das Aussehen angepasst werden kann.

Während der Definition oder der Änderung einer Paintarea wird von *FIX* eine Übersetzungsfunktion aufgerufen, wenn eine solche für die Anwendung definiert wurde, bzw. der übersetzte Text der Funktion nicht als Parameter übergeben wird. Sie kann zu dem Text einer Paintarea einen Text zur Darstellung definieren. Da in einer Paintarea oft mit Proportionalschrift gearbeitet wird, darf der dargestellte Text wesentlich länger sein. Wenn nicht mit einer Übersetzungsfunktion gearbeitet wird, dann kann der übersetzte Text auch als Parameter von außen mitgegeben werden. Ist das nicht der Fall, so wird der nicht übersetzte Text zur Anzeige verwendet.

Alle Paintareas können mit der Maus angeklickt werden. Bei der Definition kann angegeben werden, auf Klicks welcher Maustaste eine Paintareas reagieren soll. Ist die Paintarea für den entsprechenden Klick aktiviert, so wird in der Anwendungslogik ein spezielles Event generiert. *FIX* stellt Funktionen zur Verfügung, mit denen in diesem Fall ermittelt werden kann, auf welche Paintarea geklickt wurde.

Paintareas können nicht im Sinne der Anwendungslogik betreten werden. Eine Paintarea kann niemals aktives Feld/Objekt sein. Beim Durchlaufen der Maske sind Paintareas für die Anwendungslogik und ihre Events nicht existent.

Alle Paintareas haben die Möglichkeit, einen Tooltip zu besitzen. Dieser Tooltip kann schon während der Definition oder zu einem späteren Zeitpunkt an der Paintarea gesetzt werden. Paintarea-Tooltips werden direkt an *FIX/Win* gesendet und von *FIX/Win* ohne Rückfrage an *FIX* angezeigt.

1 TEXTLABEL

Paintareas vom Typ TEXTLABEL werden zur Darstellung von beliebigen Texten im Layout benutzt. Das können Führungstexte von Feldern, aber auch Rahmen oder andere grafische Bereiche sein. Da die Zeichenroutinen in der anwendungsspezifischen Bibliothek einen grafischen Bereich zur Verfügung gestellt bekommen, in dem sie die Darstellung des Textlabels hinterlegen müssen, kann dessen Umsetzung auch in eine Bitmap-Grafik erfolgen. Ein häufiger Anwendungsfall ist die Umsetzung der Labeltexte in Proportionalschrift.

2 TABLEHEADER

Paintareas vom Typ TABLEHEADER sind spezielle Textlabels, die dazu dienen, die Überschriftzeilen von Tabellenobjekten darzustellen. Dazu können TABLEHEADER mit einem entsprechenden Rahmen versehen werden, damit ein passendes Aussehen erzielt wird.

Da bei einer Tabelle die Überschrift der ersten und letzten Spalte anders umrahmt wird als die der mittleren Spalten, müssen diese Spalten entsprechend gekennzeichnet werden.

3 BITMAP

Dieser Typ wird von *FIX/Win* noch nicht unterstützt. Der Bezeichner PA_BITMAP ist als Platzhalter für zukünftige Erweiterungen reserviert

4 BUTTON

Dieser Typ wird von *FIX/Win* als Button gezeichnet. Er hat einen Maus-Hover-Effekt.

5 VARBUTTON

Reiter für Varianten dienen zum Umschalten auf eine beliebige Variante durch einen Mausklick. Die Reiter werden durch Paintareas des Typs PA_VARBUTTON realisiert. Als Bezeichnung für diese Paintareas wird der Begriff *Varbuttons* in der Dokumentation verwendet. *FIX* bietet ein API an, das die Definition einer Zeile mit mehreren Paintareas vom Typ PA_VARBUTTON erlaubt. *FIX/Win* enthält Zeichenfunktionen, die Varbuttons entsprechend darstellen, so dass die von anderen Programmen bekannte Optik entsteht. *FIX/Win* übernimmt dabei folgende Aufgaben, ohne dass eine explizite Programmierung notwendig ist:

- Darstellung der Zeile mit Varbuttons in der aktuell sichtbaren Variante.
- Umschaltung der Variante, wenn eine der Paintareas angeklickt wurde.
- Automatische Darstellung von Buttons zum Verschieben der Paintareas, wenn die Gesamtbreite größer wird als der verfügbare Platz.
- Verschieben der Paintareas beim Anklicken der Buttons zum Verschieben.

- Automatische Steuerung des Aktivierungszustands der Buttons zum Verschieben:

  Wenn kein weiteres Verschieben möglich ist, wird der entsprechende Button grau dargestellt und ist damit nicht anklickbar.

Voraussetzung dafür ist allerdings, dass die betroffene Variante zu diesem Zeitpunkt das aktive Objekt ist. Ist das nicht der Fall, dann besteht die Möglichkeit, durch Aufruf von set_activeobj_list() das aktuelle Objekt zusätzlich als aktiv zu markieren. Allerdings muss dann die Behandlung der Events beim Anklicken der Reiterleiste (im Wesentlichen BT_LEFT_PA) in der Anwendungslogik der aktiven Maske programmiert werden.

6 USERDEFINED

Alle Paintareas, die diesen Wert oder einen größeren Wert benutzen, werden von *FIX/Win* nicht dargestellt. Die Darstellung bleibt der Benutzerbibliothek von *FIX/Win* überlassen.

9 Die Entwicklungsumgebung

Programme laufen gewöhnlich in einer mehr oder weniger komplexen *Umgebung* ab. Dazu zählen bei UNIX u.a. die Position innerhalb des Dateibaumes, aus der heraus das Programm gestartet wird oder zu der es wechselt - das “working directory” -, und die Umgebungsvariablen, auf die das Programm zugreifen kann - das “environment”. Meist benötigt ein Programm zum Ablauf aber weitere Ressourcen wie bestimmte Verzeichnisse und Dateien, in denen es Hilfsprogramme und Informationen zu Geräten, Schnittstellen, Datenbanksystem etc. findet: Man denke an `/etc/termcap`, die *terminal capability database*, die die spezifischen Eigenschaften von Terminals beschreibt.

So muss ein *FIX*-Programm zwei spezielle Verzeichnisse vorfinden, um ablaufen zu können. Deren Pfade erwartet es in den Umgebungsvariablen FXDIR und FXHOME.

FXDIR enthält den Pfadnamen des *FIX-Verzeichnisses*, das bei der Installation von *FIX* an beliebiger Stelle im Dateisystem angelegt wird und alle Komponenten beinhaltet, die mit *FIX* ausgeliefert werden.

Der Umfang des *FIX*-Verzeichnisses unterscheidet sich bei *Runtime-System* und *Entwicklungssystem*. Im Runtime-System sind lediglich die Komponenten vorhanden, die eine fertige Anwendung zum Ablauf benötigt.[1] Im Entwicklungssystem sind alle *FIX*-Komponenten enthalten; es umfasst also immer ein Runtime-System. Darüberhinaus ist dem Entwicklungssystem bei Auslieferung zusätzlich auch eine *FIX*-Anwendung beigefügt, das so genannte *Softwarepröbchen* (Unterverzeichnis `demo`). Es besteht aus einigen Demo-Programmen, die dem Entwickler zur Anschauung dienen sollen.

FXHOME enthält den Pfadnamen des Verzeichnisses, das die Anwendung enthält. Eine mit *FIX* erstellte Anwendung sieht alle *FIX*-spezifischen Dateien, die nicht mit absolutem Pfadnamen angegeben sind, relativ zu diesem Verzeichnis, auch wenn der Prozess mit einem anderen Arbeitsverzeichnis abläuft.

Üblicherweise wird der Dateibaum `$FXHOME` von einem Entwickler angelegt, indem er ein entsprechendes Dienstprogramm (**mkfixenv**) aufruft; dieses erzeugt einige Unterverzeichnisse und Dateien, die die Basis der Anwendung bilden. Die so erzeugte Struktur von `$FXHOME` ist zwar nicht bindend, es empfiehlt sich jedoch, sie nicht ohne Grund zu ändern, da einige *FIX*-Tools sich daran orientieren, insbesondere das Entwicklermenü **fxm** und seine Komponenten.

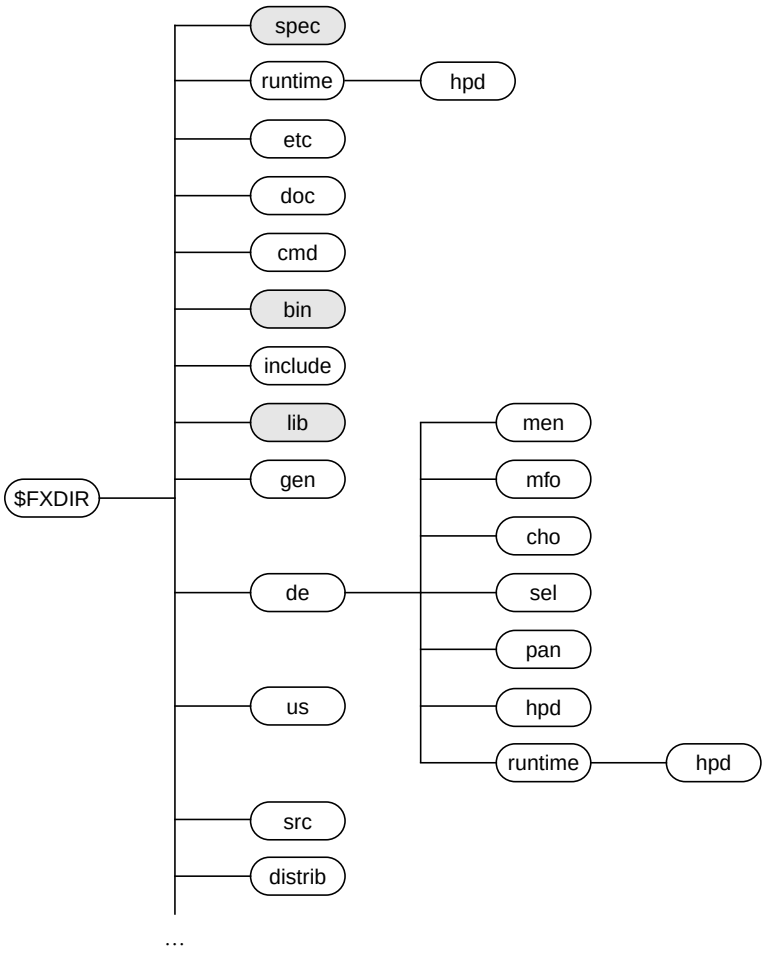
Eine detaillierte Beschreibung des *FIX*-Verzeichnisses erfolgt im nächsten Abschnitt, die des Verzeichnisses `$FXHOME` in Abschnitt 2 auf Seite 56.

1 Die Struktur des *FIX*-Verzeichnisses

Das *FIX*-Verzeichnis hat (bei einem Entwicklungssystem) folgende Struktur:

1. Teile dieser Laufzeitumgebung können vom Entwickler durch gleichartig aufgebaute Verzeichnisse und Dateien überlagert werden, die an system- oder anwendungsspezifische Gegebenheiten angepasst sind.

Abb. 9 Struktur des *FIX*-Verzeichnisses



Die einzelnen Verzeichnisse sind bestimmt für

`spec`	Installationsbeschreibung und Lizenzinformation
`runtime`	Meldungstexte und Standard-Hilfetexte
`etc`	
`fxkeycap`	Tastenbeschreibungen
`fxtermcap`	Bildschirmbeschreibungen
`ISO-15`	Zeichensatz-Verzeichnisse
`ISO-2`	
`IBM437`	
`GERMAN7`	
`doc`	Dokumentation
`man`	Manual-Seiten
`cmd`	Scripts (d.h. ausführbare Textdateien)
`bin`	Binärprogramme
`include`	Header-Dateien
`fix`	Standard Header-Dateien
`lib`	Libraries
`gen`	Steuerdateien für Source-Generierung
`de`	Dateiressourcen für *FIX*-Tools
`men`	Menübeschreibungen
`mfo`	Maskenbeschreibungen
`cho`	Choicebeschreibungen
`sel`	Selobeschreibungen
`pan`	Layoutbeschreibungen
`hpd`	Hilfetexte
`runtime`	Standard-Hilfetexte
`us`	Dateiressourcen für *FIX*-Tools (englisch)

`src`	Sourcen offen gelegter Funktionen
`distrib`	Release-Informationen

In vielen Verzeichnissen befinden sich Dateien mit dem Namen `READ_ME`, die einen Kommentar zur Bedeutung des jeweiligen Verzeichnisses enthalten.

Bei einer Mehrfach-Installation von *FIX* für unterschiedliche Umgebungen (z.B. IBM Informix ESQL/C-Version, Systemplattform) können, damit möglichst viele Teile gemeinsam genutzt werden können, die Unterverzeichnisse `spec`, `lib` und `bin` auch an anderer Stelle installiert werden. Diese Verzeichnisse werden von *FIX* nicht im Verzeichnis `$FXDIR`, sondern im Verzeichnis `$FXDIRSYS` gesucht. Die Umgebungsvariable FXDIRSYS wird, sofern sie nicht gesetzt ist, bei der Ausführung der .profile-Datei mit dem Wert von FXDIR belegt, so dass im Regelfall die beiden Verzeichnisse miteinander identifiziert werden.

$FXDIRSYS/spec

Das Verzeichnis `$FXDIRSYS/spec` enthält system- und installationsspezifische Angaben.

`p.default`, `m.default`, `m.port`

Eine Installationsbeschreibung besteht bei einem Runtime-System aus einer Datei `p.default`, beim Entwicklungssystem kommen die Dateien `m.default` und `m.port` hinzu. Die Namen der Dateien sind “entwicklungsgeschichtlich” bedingt; im Folgenden werden die Dateien meist abstrakter als p-Datei bzw. m-Datei bezeichnet.

Bei Ausführung der .profile-Datei der Anwendung wird (über `$FXDIR/etc/stdprofile`) die p-Datei der Beschreibung eingelesen.

Die für den Entwickler relevanten Angaben einer Installationsbeschreibung sind auf Seite 57ff. ausführlich erläutert. Hierzu zählen insbesondere die Pfade für Header-Dateien, Programme und Libraries.

`fxinstfile`

In dieser Datei erwartet *FIX* die bei der Registrierung erfassten Lizenzdaten, sofern nicht in der Umgebungsvariablen FXINSTFILE eine andere Datei angegeben wird.

`fxserial`

In dieser Datei werden die bei der Installation angegebene Seriennummer und der Aktivierungsschlüssel vermerkt.

`PortID`

Diese Datei enthält eine (i.d.R. zehnstellige) Nummer, die die Portierung (und damit den Patch-Stand) eindeutig kennzeichnet.

$FXDIR/fix.rc

Die Datei `$FXDIR/fix.rc` enthält Ressourcedefinitionen, mit denen das Verhalten von *FIX*-Tools beeinflusst werden kann.

$FXDIR/runtime

Das Verzeichnis `$FXDIR/runtime` enthält die sprachabhängigen Teile der Laufzeitumgebung.

`fx_texte`, `messages`

In die mit Anwendungen gebundenen *FIX*-Bibliotheken sind Meldungen nicht “eingebrannt”, sondern werden aus der (Binär-)Datei `messages` gelesen. Dies verringert zum einen die Programmgröße und erlaubt es zum anderen, solche Texte zentral zu ändern. `messages` ist mit dem *FIX*-Tool **msgprep** aus der Textdatei `fx_texte` erzeugt und enthält die Meldungstexte in einer Form, die dem Programm einen effizienten Zugriff auf den einzelnen Text gestattet.

`hpd`

Dieses Unterverzeichnis enthält Standard-Hilfetexte - wie die Tastenhilfe -, auf die ein *FIX*-Programm in gewissen Situationen zugreift.

$FXDIR/etc

Der Inhalt des Verzeichnisses `etc` ist releaseabhängig und besteht vorwiegend aus Hilfsdateien für verschiedene *FIX*-Komponenten. Einige, die auch für einen Entwickler/Benutzer interessant sein könnten, werden unten näher erläutert.

`stdprofile`

Diese Datei wird von den .profile-Dateien der einzelnen Anwendungen eingeschleust, wodurch zentrale Änderungen der Umgebung möglich sind. Sie integriert die installationsspezifischen Definitionen aus der Datei `p.default` im Verzeichnis `$FXDIRSYS/spec` (vgl. Seite 57).

`stdmakefile`

Diese Datei wird von den makefile-Dateien der einzelnen Anwendungen eingeschleust. Sie integriert die installationsspezifischen Vereinbarungen aus der Datei `m.default` im Verzeichnis `$FXDIRSYS/spec` und definiert Makros für die Übersetzung von Programmen (vgl. Seite 59).

`fxtermcap`[1]

FIX erwartet in diesem Verzeichnis eine Datei mit Namen $FXTERM (ersatzweise $TERM), die den benutzten Bildschirm beschreibt.

`fxkeycap`[1]

FIX erwartet in diesem Verzeichnis eine Datei mit Namen $FXKEYBOARD (ersatzweise $TERM), die die gewünschte Tastaturbelegung beschreibt.

`ISO-15`, `ISO-2`, `IBM437`, `GERMAN7`

Diese Verzeichnisse enthaltenen jeweils eine Zeichensatzdefinition `charset` und Unterverzeichnisse `fxkeymap` und `fxcharmap` zur Unterstützung der Ein- und Ausgabe von Zeichen dieses Zeichensatzes an den verschiedenen Geräten.[2]

$FXDIR/doc

Das Verzeichnis `doc` und seine Unterverzeichnisse (`man`) enthalten die Texte für die Online-Hilfen **fxman** und **fxindex**.

$FXDIR/cmd

Das Verzeichnis `cmd` enthält diejenigen *FIX*-Tools, die Script-Form haben. Bei Ausführung der .profile-Datei der Anwendung wird (über `etc/stdprofile`) dieses Verzeichnis in die Umgebungsvariable PATH aufgenommen.

In Unterverzeichnissen von `cmd` enthaltene Scripts sind nicht zum direkten Gebrauch durch den Entwickler/Anwender bestimmt.

---

1. Die Verzeichnisse `fxtermcap` und `fxkeycap` müssen nicht unbedingt in `$FXDIR/etc` liegen. Als Wert der Umgebungsvariable FXTERMPATH kann optional eine durch ‘:’ getrennte Folge von Verzeichnissen angegeben werden, die *FIX* nach den Dateien `fxtermcap/$FXTERM` bzw. `fxkeycap/$FXKEYBOARD` durchsuchen soll.

2. Derartige Verzeichnisse müssen nicht unbedingt in `$FXDIR/etc` liegen. Als Wert der Umgebungsvariable FXCHARSETPATH kann optional eine durch ‘:’ getrennte Folge von Verzeichnissen angegeben werden, die *FIX* nach den Dateien `$FXCHARSET/charset`, `$FXCHARSET/fxcharmap/$FXTERM` bzw. `$FXCHARSET/fxkeymap/$FXKEYBOARD` durchsuchen soll.

$FXDIRSYS/bin

Das Verzeichnis `bin` enthält diejenigen *FIX*-Tools, die Binärprogramme sind. Bei Ausführung der .profile-Datei der Anwendung wird (über `etc/stdprofile`) dieses Verzeichnis in die Umgebungsvariable PATH aufgenommen.

$FXDIR/include

Das Verzeichnis `include` und seine Unterverzeichnisse enthalten die zur Übersetzung von *FIX*-Programmen benötigten Header-Dateien. In der makefile-Datei der Anwendung wird durch die Einschleusung von `etc/stdmakefile` dafür gesorgt, dass dieses Verzeichnis beim Aufruf des C-Compilers nach Header-Dateien durchsucht wird (Makro FXCFLAGS).

$FXDIRSYS/lib

Das Verzeichnis `lib` enthält die zur Übersetzung von *FIX*-Programmen benötigten *FIX*-spezifischen Libraries, in der Regel `libfix.a` und `libipc.a`. In der makefile-Datei der Anwendung wird durch das Einschleusen von `etc/stdmakefile` dafür gesorgt, dass beim Binden von Programmen diese Libraries mit eingebunden werden (Makro FXLIBS).

$FXDIR/gen

Das Verzeichnis `gen` und seine Unterverzeichnisse enthalten Dateien, die *FIX* zur Erzeugung von C-Sourcen benötigt.

$FXDIR/de

Dieses Verzeichnis und seine Unterverzeichnisse enthalten die von *FIX*-Tools benutzten Meldungsdateien, Menüs, Masken, Selos und Choices sowie deren Layoutbeschreibungen und Hilfetexte .

Auch die wesentlichen *FIX*-Binärprogramme entnehmen Meldungen einer (Binär-)Datei. `messages` ist mit dem *FIX*-Tool **msgprep** aus der Textdatei `fix_msg` erzeugt. `fix_msg` beinhaltet eine Obermenge der Meldungen von `$FXDIR/runtime/fx_texte`.

Hinweis:

Durch Setzen der Umgebungsvariable FXTOOLHOME auf einen absoluten Verzeichnispfad kann *FIX* veranlasst werden, die Dateiressourcen für seine Tools aus diesem alternativen Verzeichnis zu lesen, das die gleiche Struktur wie `de` aufweisen muss (vgl. etwa `$FXDIR/us` unten).

$FXDIR/us

Dieses Verzeichnis und seine Unterverzeichnisse enthalten für einige der *FIX*-Tools die benutzten Meldungsdateien, Menüs, Masken, Selos und Choices sowie deren Layoutbeschreibungen und Hilfetexte in englischer Sprache.

$FXDIR/src

Dieses Verzeichnis enthält die Sourcen offen gelegter *FIX*-Funktionen, die als Vorlagen für eigene Routinen gedacht sind.

$FXDIR/distrib

Dieses Verzeichnis enthält eine Datei `ReleaseNotes.html` und zu jedem installierten Paket eine oder zwei Dateilisten.

Hinweis:

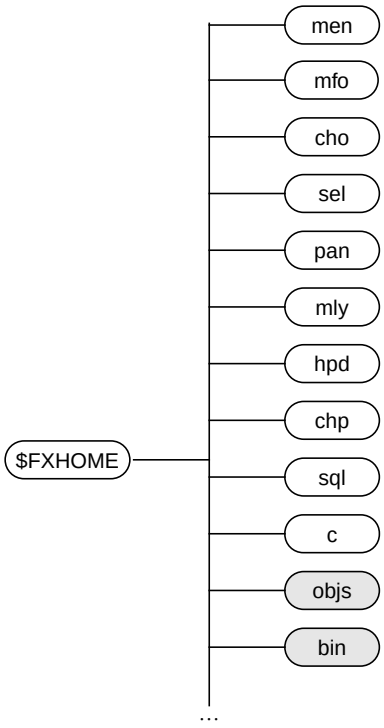
Änderungen an den Dateien des *FIX*-Verzeichnisses sollten nur bei Notwendigkeit und mit Bedacht erfolgen. Vorgenommene Änderungen müssen Sie unbedingt sichern, damit sie nach einer Neuinstallation von *FIX* ggf. nachvollzogen werden können!

2 Die Struktur der Anwendung

2.1 Verzeichnisse

Um eine Anwendung mit den von *FIX* angebotenen Werkzeugen bequem bearbeiten zu können, muss die Entwicklungsumgebung `$FXHOME` eine bestimmte Verzeichnisstruktur besitzen.

Abb. 10 Struktur einer Anwendung



Die einzelnen Verzeichnisse sind bestimmt für

`men`	Menüs
`mfo`	Masken
`cho`	Choices
`sel`	Selos
`pan`	Layoutbeschreibungen
`mly`	binäre Layouts
`hpd`	Hilfetexte der Anwendung
`chp`	“persönliche” Hilfetexte des Entwicklers/Anwenders
`sql`	SQL-Anweisungen
`c`	C-Sourcen
`objs`	Objektmodule
`bin`	Programme

Auch hier können die Unterverzeichnisse `objs` und `bin` in einem separaten Verzeichnis abgelegt werden, dessen Pfad in der Umgebungsvariable FXHOMESYS erwartet wird. FXHOMESYS wird, sofern die Variable nicht (in der .profile-Datei) gesetzt wird, bei der Ausführung von `etc/stdprofile` mit dem Wert von FXHOME belegt, so dass die beiden Verzeichnisse miteinander identifiziert werden.

FIX stellt ein Kommando zur Verfügung, das obige Verzeichnisstruktur und Standarddateien für eine Anwendung erzeugt:

<FXDIR>/cmd/mkfixenv [-y] [<directory> [<db>]]

<FXDIR> steht hier für den Pfad des *FIX*-Verzeichnisses. Ist das *FIX*-Verzeichnis der aktuellen Shell noch nicht bekannt, muss - nur dieses eine Mal - das Kommando mit komplettem Pfadnamen angegeben werden.

Sofern beim Aufruf kein Verzeichnis explizit angegeben wird, legt **mkfixenv** Dateien und Unterverzeichnisse im aktuellen Verzeichnis an. Die Bedeutung des Arguments <db> ist abhängig davon, ob Sie mit einer IBM Informix Dynamic Server[1]- oder einer IBM Informix SE[2]-Datenbank arbeiten wollen: neben einem Eintrag in der Datei `.profile` wird im zweiten Fall eine leere Datenbank mit Namen *db* erzeugt, wenn noch keine Datenbank mit diesem Namen existiert.

Abhängig von der aktuellen Umgebung und Aufrufform fragt **mkfixenv** unter Umständen, in welchem Verzeichnis *FIX* installiert ist, ob FXHOME und FXHOMESYS unterschieden werden sollen und mehr.

Bei Verwendung der Option -y werden Ja/Nein-Fragen automatisch bejaht.

mkfixenv erzeugt dann obige Verzeichnisse und legt folgende Dateien an:

`.profile`	Script, dessen Einschleusen die zum Bearbeiten der Anwendung erforderliche Umgebung schafft
`fx_texte`	Textdatei mit Programm-Meldungen (zum Erweitern)
`messages`	kompilierte Programm-Meldungen
`LOGFILE`	Protokolldatei für Datenbankfehler
`c/makefile`	makefile zum Übersetzen generierter C-Sourcen (wird beim Generieren von Source-Programmen erweitert)

Bei der Erzeugung der Datei `.profile` hat der Entwickler die Wahl, ob darin zum Abschluss automatisch das *FIX*-Entwicklermenü **fxm** ausgeführt werden soll.

Anschließend muss der Entwickler in das Verzeichnis der Anwendung wechseln und dort die Datei `.profile` ausführen:

`. ./.profile` [3]

2.2 Die Dateien `.profile` und `stdprofile`

Die Ausführung der Datei `.profile` erzeugt die zur Benutzung von *FIX* notwendige Umgebung. Frisch angelegt hat `.profile` etwa folgendes Aussehen:

```
# .profile

# begin of user definitions
```

1. IBM Informix Dynamic Server ist ein Warenzeichen der International Business Machines Corporation.
2. IBM Informix SE ist ein Warenzeichen der International Business Machines Corporation.
3. Für UNIX-Anfänger: Der alleinstehende Punkt veranlasst die aktuelle Shell, die Datei `.profile` zu lesen, als wäre es Eingabe von der Tastatur, und nicht, wie etwa beim Aufruf eines Shell-Scripts, einen neuen Prozess zu erzeugen; dann wäre das Setzen von Umgebungsvariablen sinnlos.

```
…
# end of user definitions

[ -n "${FXDIR}" ] || FXDIR=<bei Erzeugung von .profile gültiges FIX-Verzeichnis>
export FXDIR
[ -n "${FXDIRSYS}" ] || FXDIRSYS=<bei Erzeugung von .profile gültiges Verzeichnis $FXDIRSYS>
export FXDIRSYS
. ${FXDIR}/etc/stdprofile
```

In Kenntnis der Verzeichnisse `$FXDIR` und `$FXDIRSYS` wird beim Einschleusen der Datei `$FXDIR/etc/stdprofile` die zur Benutzung von *FIX* erforderliche Umgebung erzeugt. Dabei besetzt *FIX* vom ihm benötigte, *noch nicht gesetzte* Variablen mit Defaultwerten.

Ggf. gesetzt und exportiert werden die Umgebungsvariablen (die Klammern enthalten jeweils den Defaultwert):

Angaben zu *FIX*:

FXDIR	das *FIX*-Verzeichnis
FXDIRSYS	das Verzeichnis für die versionsabhängigen Bestandteile von *FIX* (`$FXDIR`)
FXCHARSET	der dem *FIX*-Verzeichnis zugrunde liegende Zeichensatz
FXDBSYSTEM	das Datenbanksystem, mit dem *FIX* zusammenarbeitet (IBM Informix)
FXOS	die Systemplattform
FXINSTFILE	die *FIX*-Lizenzdatei (sofern nicht `$FXDIRSYS/spec/fxinstfile`)
EDITOR	Texteditor (meist vi)

für *FIX* relevante Angaben zur Anwendung:

FXHOME	das Verzeichnis, das die Anwendung enthält (`pwd`)
FXHOMESYS	das Verzeichnis für die versionsabhängigen Bestandteile der Anwendung (`$FXHOME`)
HLPPATH	das Verzeichnis, das die Hilfetexte der Anwendung enthält; einem relativen Pfad stellt *FIX* "$FXHOME/" voran (`$FXHOME/hpd`)
CHLPPATH	das Verzeichnis, das die "persönlichen" Hilfetexte des Entwicklers/Anwenders enthält; einem relativen Pfad stellt *FIX* "$FXHOME/" voran (`$FXHOME/chp`)
INFORMIXDIR	das IBM Informix-Verzeichnis, wenn erforderlich (`/usr/informix`)
DBPATH [SE]	Datenbank-Pfad (`$FXHOMESYS`)
DBNAME	Name der Datenbank (`basename $FXHOME`)

Mit [SE] markierte Angaben sind nur bei Einsatz von IBM Informix SE von Belang.

Treffen die Defaultwerte nicht zu, sollten die entsprechenden Variablen im Abschnitt "user definitions" gesetzt werden oder bereits vorher in der Umgebung gesetzt sein.

FIX-Programme werten weitere Umgebungsvariablen aus, die jedoch entweder als gesetzt vorausgesetzt werden können (SHELL, PATH, TERM) oder optional sind und ggf. durch Defaultwerte ersetzt werden; dazu zählen FXDATE, FXTRUTH, FXRADIXCHAR, FXRUNTIMEDIR, FXCHARSETPATH, FXTERM, FXKEYBOARD, FXTERMPATH und HARDCOPY.

Die Verwendung von `stdprofile` bietet den Vorteil, dass bei einem Release-Wechsel eine Änderung an zentraler Stelle erfolgen kann und nicht die `.profile`-Dateien aller bestehenden Anwendungen modifiziert werden müssen.

Weitere Aufgaben von `stdprofile` sind:

- Der aktuelle Pfad $PATH wird um die Verzeichnisse `$INFORMIXDIR/bin`, `$FXDIR/cmd` und `$FXDIRSYS/bin` erweitert, sofern diese Verzeichnisse existieren.
- Die Maske zum Anlegen von Dateien (umask) wird auf 002 gesetzt.
- Die Existenz der Bildschirm- und Tastenbeschreibung wird überprüft, sofern nicht FXTERMPATH gesetzt ist.

2.3 Die Dateien `fx_texte` und `messages`

Anstatt Meldungen - worunter alle Texte zu verstehen sind, die abgeändert werden können, ohne den Programmablauf zu beeinflussen - als Zeichenketten in *FIX*-Programme “einzubrennen”, kann der Entwickler sie unter Vergabe einer eindeutigen Nummer in einer separaten Datei ablegen. Dies hat mehrere Vorzüge:

- bei Änderung eines Textes muss die Anwendung nicht neu übersetzt werden,
- bei der Ausführung des Programms werden nur die Texte in den Speicher geladen, die benötigt werden.

Eine wichtige Anwendung von Meldungstexten sind *Prompts*: dies sind einzeilige Hilfstexte mit dem Charakter einer Eingabehilfe, die z.B. zu jedem Maskenfeld definiert werden können.

FIX erwartet, dass zu jeder Anwendung eine Datei `messages` im Verzeichnis `$FXHOME` existiert. Diese Datei muss mit der *FIX*-Utility **msgprep** erzeugt sein, da sie neben den eigentlichen Texten eine Art Inhaltsverzeichnis zum schnellen Zugriff auf einen bestimmten Text enthalten muss.

Als Grundlage für **msgprep** dient eine Textdatei. Nach Konvention ist dies die Datei `fx_texte` im Verzeichnis `$FX-HOME`.

2.4 Die Dateien `c/makefile` und `stdmakefile`

Für die Programmerzeugung generiert *FIX* im Verzeichnis `$FXHOME/c` eine Datei `makefile`, die zunächst etwa so aussieht:

```
# c/makefile

include $(FXDIR)/etc/stdmakefile

SRC=$(FXHOME)/c
OBJS=$(FXHOMESYS)/objs
BIN=$(FXHOMESYS)/bin

ESQLCFLAGS = [1]

CFLAGS = $(FXCFLAGS) -I$(SRC)

LIBS = $(FXLIBS) $(SPECLIBS)
```

`makefile` schleust erst die Datei `$FXDIR/etc/stdmakefile` ein, die - nach Lesen der m-Datei in `$FXDIR-SYS/spec` - darauf aufbauend Makro-Vereinbarungen vornimmt. Die m-Datei enthält dabei die für die Programmentwicklung notwendigen installationsspezifischen Angaben (die Klammern enthalten jeweils ein Beispiel):

SHELL	von **make** benutzter Kommandointerpreter (/bin/sh)
O	Dateiendung von Objektdateien (.o)
A	Dateiendung von Libraries (.a)
E	Dateiendung von Binärprogrammen (Leerstring)
FXCPPFLAGS	Schalter für die Übersetzung von *FIX*-Sourcen (-D$(FXDBSYSTEM))
INCLDIRSQL	das Verzeichnis, das die ESQL/C-Header-Dateien enthält (`/usr/informix/incl/esql`)
ESQLC	Aufruf des ESQL/C-Präprozessors (/usr/informix/lib/esql/esqlc)
CC	Aufruf des C-Compilers (cc)

1. Die Vereinbarung von ESQLCFLAGS wurde erst bei *FIX* 3.1.0 eingeführt.

AR	Aufruf des “*Ar*chivers” (ar)
ARFLAGS	Archiver-Schalter (ruv)
LIBSQL	Pfade der ESQL/C-Libraries (/usr/informix/lib/esql/libsql.a …)
LIBISAM *	Pfad der C-ISAM[1]-Library (/usr/lib/libisam.a)
SPECLIBS	Systemspezifische zusätzliche Libraries (-lm)
RANLIB	Aufruf des “*Lib*rary *Ran*domizers” (ranlib)

Mit * markierte Angaben sind nur bei Einsatz von IBM Informix C-ISAM von Belang.

Makro-Vereinbarungen in `stdmakefile`

Das Makro FXCFLAGS beinhaltet den Wert von FXCPPFLAGS sowie die Compiler-Optionen

-I$(FXDIR)/include	um *FIX* spezifische Header-Dateien zu finden
-I$(INCLDIRSQL)	um ESQL/C spezifische Header-Dateien zu finden

Das Makro FXLIBS enthält die Libraries, mit denen das Programm üblicherweise zu binden ist:

$(LIBFIX)	die *FIX*-Library ($(FXDIRSYS)/lib/libfix$A)
$(LIBIPC)	die Library zur Serverkommunikation ($(FXDIRSYS)/lib/libipc$A)
$(LIBSQL)	die ESQL/C-Libraries
$(LIBISAM) *	die C-ISAM-Library

Makro-Vereinbarungen in `makefile`

Bei der im Verlauf der Source-Generierung vorgenommenen Erweiterung von `makefile` werden drei Verzeichnispfade unterschieden:

SRC	das Verzeichnis, das die Quelltexte (.ec-, .h-Dateien) enthält
OBJS	das Verzeichnis, in dem die vom Compiler erzeugten Objektdateien abgelegt werden
BIN	das Verzeichnis, in dem die gebundenen Programme abgelegt werden

1. C-ISAM ist ein eingetragenes Warenzeichen der International Business Machines Corporation.

10 Das Entwicklermenü fxm

Um dem Entwickler den Einstieg in *FIX* zu erleichtern, bietet *FIX* die Menüoberfläche **fxm**[1] an. Der Einsatz von **fxm** setzt voraus, dass

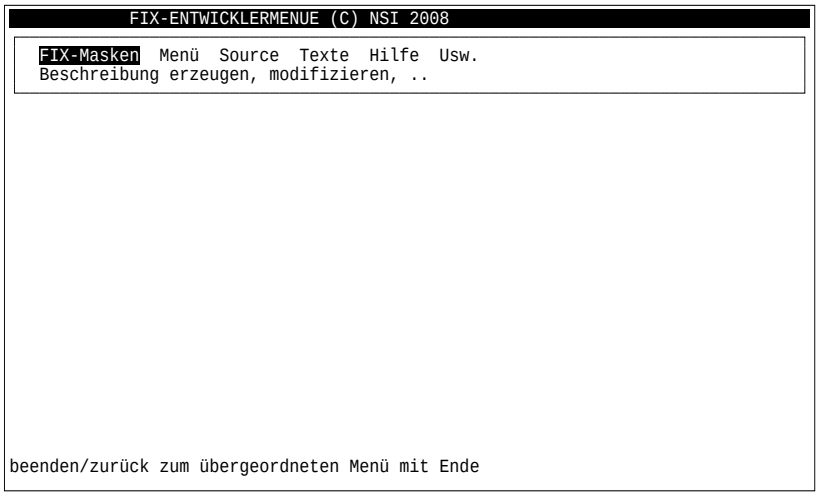
- die Anwendung die auf Seite 56 beschriebene Verzeichnisstruktur besitzt,
- die Konventionen zur Benennung von Dateien eingehalten sind,
- für Namen von Dateien, Datenbanktabellen etc. nur ASCII-Zeichen verwendet werden.

  Hinweis:

  Bei Datei- oder Verzeichnisvorschlägen berücksichtigt **fxm** nur Verzeichniseinträge, die ausschließlich aus den ASCII-Zeichen ! (0x21) bis ~ (0x7E) gebildet sind.

fxm ermöglicht die Ausführung fast aller Tätigkeiten, die zur Erstellung und bei der Pflege von *FIX*-Programmen notwendig sind. Die Bedienung ist so gestaltet, dass sie hinlänglich einfach für den Neuling, aber auch performant für den geübten Nutzer ist. **fxm** wechselt beim Start selbständig ins Verzeichnis `$FXHOME`.

Abb. 11 fxm: Hauptmenü



1 Das Hauptmenü

Das Hauptmenü von **fxm** ist eine Menüleiste, die “Pulldown”-Menüs für die Bearbeitung von *FIX*-Objekten und Quellprogrammen sowie für die Nutzung von Hilfen und Tools beinhaltet (vgl. Abbildung Seite 61).

Menüpunkte kann man auf unterschiedliche Art erreichen. Der einfachste Weg ist die Anwahl über die Tasten kr (CursorRight) und kl (CursorLeft) bzw. kd (CursorDown) und ku (CursorUp). Hierbei verhält sich kd im Hauptmenü wie

1. **fxm** ist selbst ein mit *FIX* erstelltes Programm, das Masken und Menüs aus den entsprechenden Unterverzeichnissen des *FIX*-Verzeichnisses verwendet.

RT, d.h. löst das entsprechende Untermenü aus. In Untermenüs können die Tasten kr und kl benutzt werden, um schnell zum benachbarten Untermenü zu wechseln. Jeder Menüpunkt kann außerdem durch seinen Anfangsbuchstaben ausgelöst werden, da die Anfangsbuchstaben innerhalb eines Untermenüs eindeutig sind.

Ein Menüpunkt kann auch durch Eingabe der Ziffer ausgelöst werden, die der Position des Menüpunktes im Menü entspricht. Die Ziffer 0 hat die gleiche Bedeutung wie die Taste EN, führt also zum Verlassen des Menüs.

Beispiel:

Sie haben augenblicklich den Menüpunkt “FIX-Masken→Layout-Editor” (Abb. 12) angewählt und wollen zum Punkt “Source→generieren aus Maske” (Abb. 14) wechseln.

Geben Sie Folgendes ein:

0 - um das Untermenü “FIX-Masken” zu verlassen

3 - um in das Untermenü “Source” zu gelangen

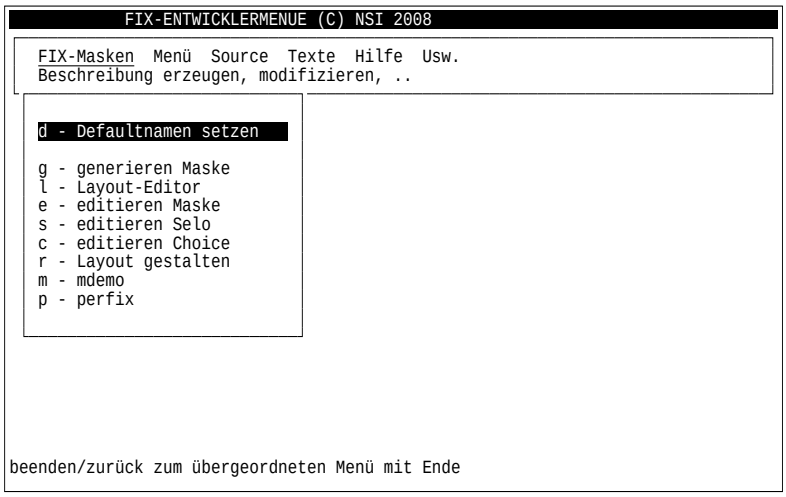
3 - um den Punkt “generieren aus Maske” auszuwählen

Natürlich können Sie die verschiedenen Methoden auch kombinieren. Statt der letzten “3” des Beispiels könnten Sie auch “g” drücken, um den Menüpunkt auszuwählen.

2 Das Untermenü FIX-Masken

In diesem Untermenü werden alle Funktionen zur Verfügung gestellt, die notwendig sind, um Masken zu generieren, zu bearbeiten und zu testen.

Abb. 12 fxm: FIX-Masken



Defaultnamen setzen

Sie werden bei einer Folge von Arbeitsschritten häufig Objekte und Dateien mit gleichen Namen bearbeiten wollen. Unter diesem Punkt kann ein Basisname angegeben werden, der bei den anderen Menüpunkten, jeweils ergänzt durch das passende Unterverzeichnis (hier: `mfo`) und die an dieser Stelle erwartete Endung (hier: .mfo), als Argument vorgeschlagen wird.

generieren Maske

Zu einer von Ihnen erfragten Tabelle der Datenbank $DBNAME wird eine Maske (Objektbeschreibung und Layout) generiert. Die Maske enthält ein Feld zu jeder Spalte der Tabelle. Wahlweise kann ein binäres Layout oder eine Layoutbeschreibung (Layout-Version 295) generiert werden. Mehr hierzu im Kapitel *Generierung von Objektbeschreibungen*.

Layout-Editor

Hier wird der Layout-Editor **led** aufgerufen. Sie werden aufgefordert, den Namen einer Beschreibungsdatei einzugeben. Existiert diese Beschreibungsdatei nicht, so wird sie vor dem Aufruf des Layout-Editors mit Hilfe des Programms **obgen** als Beschreibungsdatei eines Objekts ohne Elemente erstellt; als Layout wird eine Layoutbeschreibung (Layout-Version 295) generiert.

editieren Maske
editieren Selo
editieren Choice

Beschreibungsdateien sind Textdateien, die mit jedem Editor bearbeitet werden können. An dieser Stelle wird der Editor, der in der Umgebungsvariablen EDITOR angegeben ist, mit der Beschreibungsdatei als Argument aufgerufen.

Im Folgenden ist mit Editor immer das in der Umgebungsvariable EDITOR benannte Programm gemeint.

Layout gestalten

Hier wird der eingeschränkte Layout-Editor **rled** aufgerufen. Sie werden aufgefordert, den Namen einer Beschreibungsdatei einzugeben, deren Layout Sie ändern möchten.

mdemo

mdemo ist ein Testprogramm für Objekte. Es finden keine Datenbankzugriffe statt, ausgenommen für Selos und Choice, die auf Datenbankanfragen beruhen.

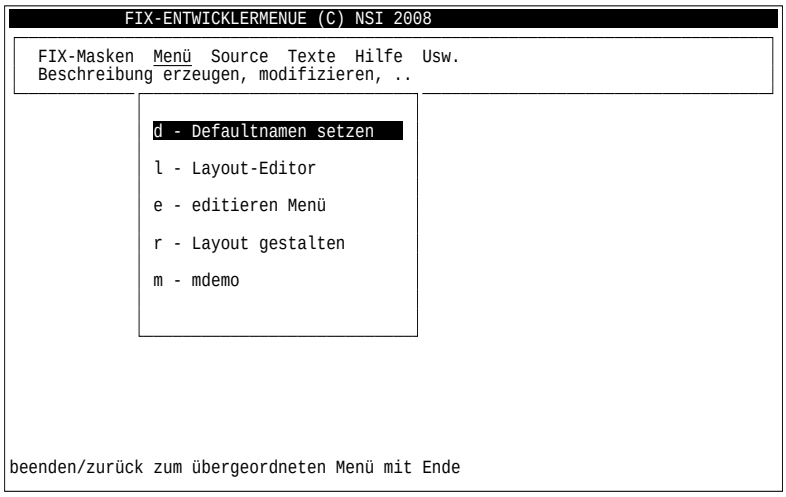
perfix

perfix ist ein Masken-Interpreter. Er beherrscht die grundlegenden Datenbankoperationen Einfügen, Ändern und Löschen.

3 Das Untermenü Menü

In diesem Untermenü werden alle Funktionen zur Verfügung gestellt, die notwendig sind, um Menüs zu generieren, zu bearbeiten und zu testen.

Abb. 13 fxm: Menü



Defaultnamen setzen

siehe Untermenü “FIX-Masken→Defaultnamen setzen”; allerdings wird hier als Verzeichnis `men` und als Dateiendung .men hinzugefügt.

Layout-Editor

siehe Untermenü “FIX-Masken→Layout-Editor”.

editieren Menü

siehe Untermenü “FIX-Masken→editieren Maske”.

Layout gestalten

siehe Untermenü “FIX-Masken→Layout gestalten”.

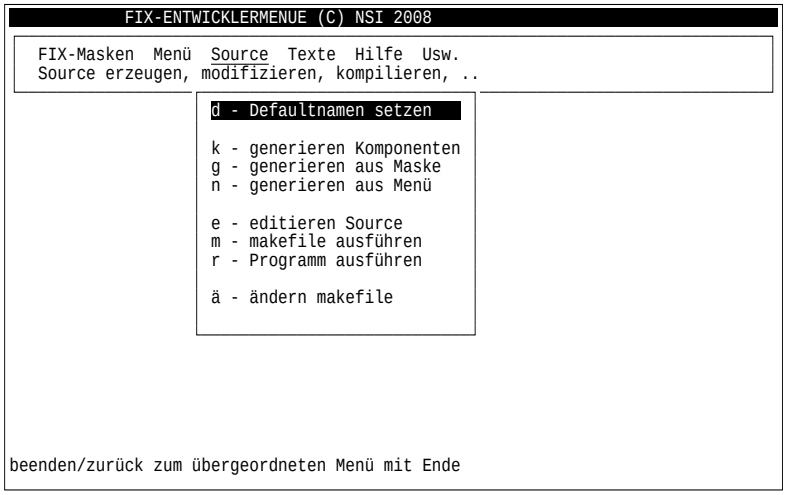
mdemo

siehe Untermenü “FIX-Masken→mdemo”.

4 Das Untermenü Source

In diesem Untermenü sind alle nötigen Hilfsmittel enthalten, um von einer Beschreibungsdatei zu einem ausführbaren Programm zu gelangen.

Abb. 14 fxm: Source



Defaultnamen setzen

siehe Untermenü “FIX-Masken→Defaultnamen setzen”; allerdings wird als Verzeichnis `c` angenommen.

generieren Komponenten

Hier werden zu Beschreibungsdateien von Masken Komponenten des Quellprogramms generiert. Welche Komponenten generiert werden, bestimmen Sie in einer Maske. Einzelheiten hierzu finden Sie im Kapitel *Source-Generierung für Masken*.

generieren aus Maske

Hier erfolgt die Generierung eines kompletten Quellprogramms zu einer Maskenbeschreibungsdatei.

generieren aus Menü

Hier erfolgt die Generierung eines kompletten Quellprogramms zu einer Menübeschreibungsdatei.

editieren Source

Um Änderungen an generierten Quellprogrammen vorzunehmen, wird hier der Editor gestartet. Als Argument werden alle Dateien im Unterverzeichnis `c` mit den Endungen .ec, .c und .h vorgeschlagen.

makefile ausführen

Bei der Generierung wird optional ein Eintrag in `c/makefile` erzeugt. Hier wird **make** ausgeführt und so ein ausführbares Programm oder ein Modul erstellt.

Programm ausführen

Ein übersetztes Programm kann an dieser Stelle ausgeführt und getestet werden.

Hinweis:

Das Feld zur Eingabe der Argumente in der durch “Source→Programm ausführen” aufgeblendeten Maske erlaubt auch Nicht-ASCII-Zeichen.

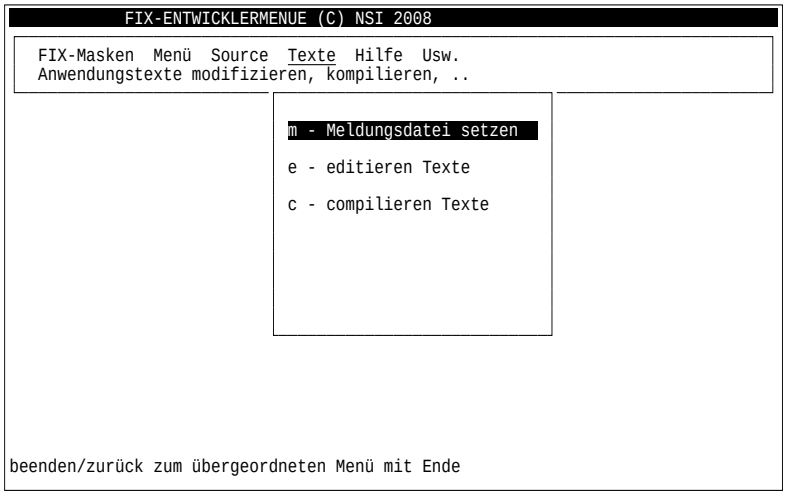
ändern makefile

Standardmäßig wird ein Eintrag in `c/makefile` generiert, der das Quellprogramm in ein ausführbares Programm übersetzt. Sollen mehrere Objekte zu einem Programm gebunden werden, können Sie hier den betreffenden Eintrag mit Hilfe des Editors ändern.

5 Das Untermenü Texte

Dieses Menü umfasst das Editieren und Übersetzen von Meldungsdateien. Als Meldungsdatei ist die in der Umgebungsvariable FXMSG benannte Datei bzw. `fx_texte` voreingestellt.

Abb. 15 fxm: Texte



Meldungsdatei setzen

Mit diesem Menüpunkt kann der Name der Meldungsdatei geändert werden. Der Dateiname wird in der Umgebungsvariablen FXMSG hinterlegt.

editieren Texte

Die Meldungsdatei kann mit Hilfe des Editors bearbeitet werden.

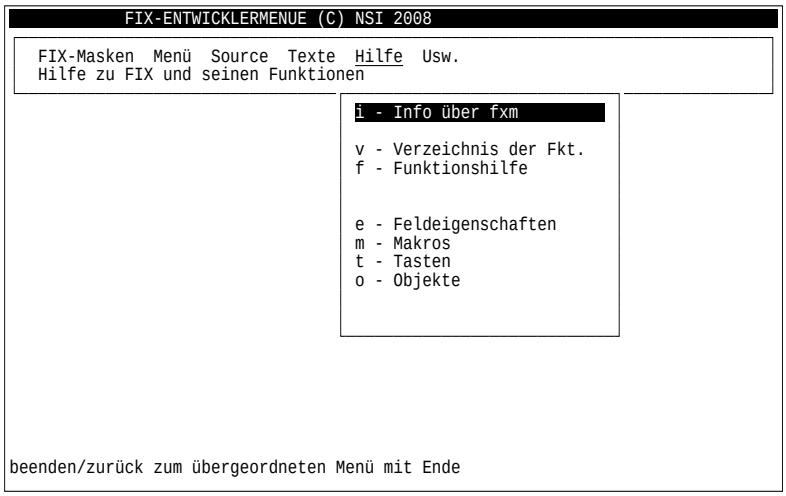
compilieren Texte

Die Auswahl dieses Punktes stößt die Übersetzung der Meldungsdatei an, d.h. die Erzeugung der Datei `messages`.

6 Das Untermenü Hilfe

Dieses Untermenü enthält verschiedene Tools, die dem Entwickler Hilfestellung bieten.

Abb. 16 fxm: Hilfe



Info über fxm

fxm blendet eine Maske ein, der Sie die Versionsnummer Ihres *FIX* entnehmen können. Geben Sie diese Nummer bitte immer an, wenn Sie sich mit einem Problem an uns wenden.

Verzeichnis der Funktionen

Sie erhalten ein Verzeichnis aller dokumentierten *FIX*-Funktionen (→ **fxindex**).

Funktionshilfe

Sie können einen Funktionsnamen oder auch nur einen Teil des Namens angeben und erhalten Informationen über die Verwendung aller Funktionen, die mit diesem Namensteil beginnen (→ **fxman**).

Feldeigenschaften

Hier wird die Header-Datei `fix/accept.h` angezeigt, die u.a. die Makros zu den Feldeigenschaften enthält.

Makros

Hier wird die Header-Datei `fix/macros.h` angezeigt.

Tasten

Hier wird die Header-Datei `fix/keys.h` angezeigt, die u.a. die Makros der Events enthält.

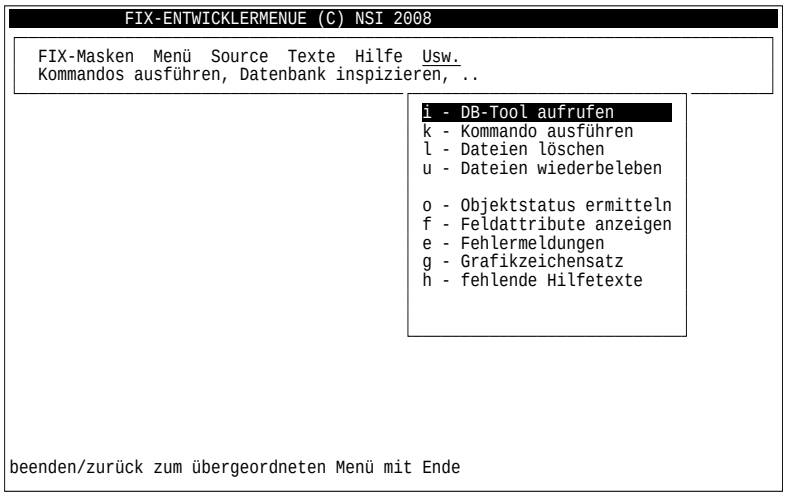
Objekte

Hier wird die Header-Datei `fix/obj.h` angezeigt, die u.a. die Datenstrukturen der Objekte und Wertebereiche ihrer Komponenten enthält.

7 Das Untermenü Usw.

Dieses Untermenü enthält Tools zur Erleichterung der Arbeit.

Abb. 17 fxm: Usw.



DB-Tool aufrufen

Hier erfolgt der Aufruf des IBM Informix-Tools zur interaktiven Arbeit mit einer Datenbank (**dbaccess**) mit dem Wert der Umgebungsvariablen DBNAME als Argument. Vor dem Aufruf wird in das Verzeichnis `$FXHOME/sql` gewechselt.

Kommando ausführen

Das Kommando wird in einer eigenen Shell mit der beim Aufruf von **fxm** vorhandenen Umgebung ausgeführt. Aktuelles Verzeichnis ist das Verzeichnis `$FXHOME`.

Hinweis:

Das Feld zur Eingabe des Kommandos in der durch “Usw.→Kommando ausführen” aufgeblendeten Maske erlaubt auch Nicht-ASCII-Zeichen.

Dateien löschen

Um eine Datei wiederbeleben zu können, wird hier nicht wirklich gelöscht. Vielmehr wird zum “Löschen” das Script **del** verwendet, das die Datei nach `/usr/tmp` verschiebt. Bei einem Neustart des Systems (Reboot) werden die Dateien im Verzeichnis `/usr/tmp` i. Allg. automatisch gelöscht.

Dateien wiederbeleben

Hier können Dateien wiederbelebt werden, die mit dem Menüpunkt Usw.→Dateien löschen (oder mit **del**) gelöscht wurden. Nach einem Neustart des Systems ist ein Wiederbeleben meist nicht mehr möglich, da dabei i. Allg. alle Dateien im Verzeichnis `/usr/tmp` gelöscht werden.

Objektstatus ermitteln

Die numerische Kodierung eines Objektstatus wird entschlüsselt und im Klartext ausgegeben. Sind in der Kodierung nicht erkannte Objektstati enthalten, so wird “unknown” ausgegeben.

Feldattribute anzeigen

Die numerische Kodierung von Feldeigenschaften wird entschlüsselt und im Klartext ausgegeben.

Fehlermeldungen

Zu einer Fehlernummer wird der Fehlertext ausgegeben (→ **err**).

Grafikzeichensatz

Der Zeichensatzes des Bildschirms (→ **graphic**) wird ausgegeben.

fehlende Hilfetexte

Zu allen im Unterverzeichnis `mfo` liegenden Masken werden die Felder ermittelt, die keinen Hilfetext haben.

11 Tools für den Entwickler

Die Verzeichnisse $FXDIR/cmd und $FXDIRSYS/bin enthalten die Tools, die *FIX* dem Entwickler anbietet, und Hilfsprogramme, die *FIX* selbst benutzt. Ersteres enthält nur Scripts, Letzteres Binärprogramme. Der Kommandoumfang ist abhängig von eingesetztem Datenbank- und Betriebssystem.

1 Die Scripts in $FXDIR/cmd

Mit dem Kommando **fxcmd** erhalten Sie eine Beschreibung der Kommandos im Verzeichnis $FXDIR/cmd, die *in Ihrer Installation* zur Verfügung stehen.

fxcmd [<prefix>] bzw. **fxcmd** <file> …

Hierzu gehören:

acr, **ad**, **ae**, **al**, **ap**, **arm**, **arn**
mcr, **md**, **me**, **ml**, **mp**, **mrm**, **mrn**
vcr, **vd**, **ve**, **vp**, **vrm**, **vrn**

Diese Kommandos dienen zum Arbeiten mit *FIX*-Objekten von der Shell aus.

Ihr Gebrauch setzt die Einhaltung der *FIX*-Konventionen für Unterverzeichnisse und Dateiendungen voraus. Die Kommandonamen sind gebildet aus Objekttyp (1. Zeichen)

v	Menü (Verteiler)
m	Maske
a	Choice (Auswahl).

und Aufgabe (2.-3. Zeichen)

cr - “create”:	Erzeugen eines Objekts
e - “edit”:	Editieren der Beschreibungsdatei mittels $EDITOR
l - “layout”:	Editieren mittels **led**
d - “demo”:	Prototyping-Test des Objekts
p - “perform”:	Ausführen von Masken, sonst: wie “d”
rm - “remove”:	Löschen, einschließlich Layout
rn - “rename”:	Umbenennen

So ergeben sich die Kommandos

für Menüs:	vcr, ve, vl, vd, vp, vrm, vrn
für Masken:	mcr, me, ml, md, mp, mrm, mrn
für Choices:	acr, ae, ad, ap, arm, arn (al wird nicht unterstützt)

Wo sinnvoll, können mehrere Argumente angegeben werden, Unterverzeichnis und Endung werden nicht angegeben.

Beispiel:

vl hm	→ led men/hm.men
mrm c1 c2	→ rm mfo/c1.mfo mly/c1.mly mfo/c2.mfo mly/c2.mly
crn c1 c2	→ mv cho/c1.cho cho/c2.cho; mv mly/c1.mly mly/c2.mly

edcmap

Editieren der Umsetzungstabelle für die Zeichenausgabe auf $FXTERM (“Character Map”).

edkc

Editieren der Tastenbeschreibung von $FXKEYBOARD.

edkmap

Editieren der Umsetzungstabelle für die Zeicheneingabe von $FXKEYBOARD (“Key Map”).

edtc

Editieren der Bildschirmbeschreibung von $FXTERM.

fehlend

Durchsucht die auf .mfo endenden Dateien im Verzeichnis `$FXHOME/mfo` nach Feldern, zu denen es keinen Hilfetext gibt.

fgmask [<option> …] <file> [<basename>]

generiert Sourcecode zu einer Maske.

fgmenu [<option> …] <file> [<basename>]

generiert Sourcecode zu einem Menü.

fxcmd [<string> …]

Extrahiert die Dokumentation aus den Scripts in `$FXDIR/cmd`.

Wird genau ein Argument angegeben, werden alle Scripts dokumentiert, deren Namen mit <string> beginnt.

fxindex

Listet alle dokumentierten Funktionen der *FIX*-Library auf.

fxm [-7 | -8]

Aufruf des Entwicklermenüs.

fxman <string>

Gibt die Beschreibung zu einer Funktion der *FIX*-Library aus.

Für <string> ist ein Präfix des Funktionsnamens ausreichend, **fxman** "" liefert also die Beschreibungen aller dokumentierten Funktionen.

hardcopy

Ein Prototyp für ein Hardcopy-Programm. Die Umgebungsvariable HARDCOPY muss den Pfad eines Programms enthalten, das die Hardcopy von stdin entgegennimmt und weiterverarbeitet, z.B. den Namen dieses Scripts.

lconv [-v] [-o <layoutbeschreibungsdatei>] <beschreibungsdatei>

Mit Hilfe dieses Scripts können zu Objekten mit binären Layouts zu diesen äquivalente Layoutbeschreibungen erzeugt werden. Der Aufruf

lconv -h

erklärt den Gebrauch.

Das Script benutzt Hilfsprogramme **mlycheck**, **mly2pan**, **pan2mly** aus `$FXDIRSYS/bin`.

led [-7 | -8] [<beschreibungsdatei>]

Aufruf des Objekt-/Layout-Editors für Menüs und Masken.

less [<option> … [<file> …]
L [<option> …] [<file> …]

less ist ein Programm zum Blättern durch Textdateien: Bedienungsanleitung mit Taste 'h'.

lpcat [-c] [-7 | -8] [-d<fxterm>] [<file> …]
lpcat [-h]

Kopiert die angegebenen Dateien nach stdout, wobei die geräteunabhängigen Umschaltcodes ^A, ^B, ^C, ^D, ^E, ^N, ^O und Semigrafikzeichen durch die dafür in der Gerätebeschreibung `<fxterm>` definierten Steuersequenzen ersetzt werden. Wird <fxterm> nicht angegeben, wird stattdessen der Wert der Umgebungsvariablen FXPRINTER bzw. FXTERM verwendet. Ist FXTERMPATH nicht gesetzt, wird die Gerätebeschreibung im Unterverzeichnis `fxtermcap` von `$FXDIR/etc` erwartet.

Der Schalter -h gibt weitere Erläuterungen (→ **hardcopy**).

mdemo <beschreibungsdatei> [<option> …]

Aufruf des Prototyping-Tools, mit dem Objektbeschreibungen getestet werden können.

mkfixenv [-y] [<directory> [<dbname>]]

Dient zum interaktiven Einrichten der typischen Verzeichnisse und Dateien einer *FIX*-Anwendung.

msgdeprep [-7 | -8] <file>

Extrahiert die Texte und die zugeordneten Nummern aus der binären Form einer Meldungsdatei.

<file> ist eine Datei im von **msgprep** erzeugten Format. Die Ausgabe erfolgt nach stdout.

msgprep [-7 | -8] <file> [<outfile>]

Erstellt die binäre Form einer Meldungsdatei.

<file> ist eine Textdatei im geeigneten Format. Die Ausgabe erfolgt nach <outfile> bzw. stdout, wenn <outfile> nicht angegeben wird.

mlyconv [-t pan] [-7 | -8] [-ocs <outcharset>] [-o <outfile>] [-cs <charset>] -h <height> -w <width> <file>

Erstellt zu einem binären Layout eine Layoutbeschreibung.

obtyp <ob_state>

Entschlüsselt die Kodierung eines Objektstatus (Komponente ob_state).

panconv [-t <outformat>] [-ocs <outcharset>] [-o <outfile>] [<file>]

Erstellt zu einer Layoutbeschreibung ein binäres Layout.

perfix <beschreibungsdatei> [<option> …]

Aufruf des Maskeninterpreters.

rled [-7 | -8] [<beschreibungsdatei>]

Aufruf des "eingeschränkten" Layout-Editors.

run <programname> [<arg> …]

Dient zur Ausführung eines Programms im Verzeichnis `$FXHOMESYS/bin`.

styp [<styp>]

Entschlüsselt die Kodierung von Feldeigenschaften.

tcout <capability>

Liest aus der Bildschirmbeschreibung für das Terminal $FXTERM den zugehörigen Eintrag und schreibt die entsprechende Sequenz nach stdout.

tgoto <row> <column>

Liest aus der Bildschirmbeschreibung für das Terminal $FXTERM den Eintrag zur Cursor-Positionierung und schreibt die mit <row> und <column> versorgte Sequenz nach stdout.

Erforderlich sind weiter folgende Hilfsscripts:

choice

Hilfsscript zum Programm **led**.

convenient

Hilfsscript zu **acr** etc.

fxask[-t] <string> …

Abfrage auf Ja oder Nein mit entsprechendem Exit-Status 1 bzw. 0.

Der Schalter -t (“trap”) verhindert einen Abbruch durch die INTERRUPT-Taste. Mit der Taste ‘!’ kann in eine Shell verzweigt werden.

fxdisplay [-n] <string> …

echo mit Schalter -n wie bei BSD, sonst konform zu System V.

lnorcp <filename> <linkname>

Wie **ln**, aber ersatzweise **cp**, wenn **ln** scheitert, z.B. weil die Filesysteme verschieden sind.

selo

Hilfsscript zum Programm **led**.

stripcomment [-a] <file>

```
/*
 *    Entfernt “Senkrecht-Kommentare” in C-Sourcen wie in diesem Beispiel.
 */
```

Der Schalter -a entfernt alle Kommentare.

Dem Entwickler werden einige Tools für die Arbeit mit einer IBM Informix-Datenbank angeboten:

err [<errcode> …]

Gibt die Fehlertexte zu Datenbank-Fehlernummern aus.

load <table> …

Lädt die Tabellen <table> aus den Dateien `<table>.unl` in die Datenbank $DBNAME.

unload [<table> …]

Entlädt alle bzw. die angegebenen Tabellen der Datenbank $DBNAME in Dateien mit Namen `table.unl`.

Von vielen Benutzern als Komfort empfunden, aber nicht unbedingt erforderlich sind:

graphic

Dient zur Ausgabe des terminalspezifischen alternativen Zeichensatzes.

path [<option> …] <command>

Dient zum Auffinden einer Kommandodatei entlang der Umgebungsvariablen PATH.

Schalter:

- -l Ausgabe der Verzeichnis-Information, Langform (ls -li)
- -h Ausgabe eines erweiterten Hilfetextes

Bei allen übrigen Scripts in `$FXDIR/cmd` handelt es sich um intern benötigte Hilfsscripts oder optionale Zugaben.

2 Die Programme in $FXDIRSYS/bin

Das Verzeichnis $FXDIRSYS/bin enthält die zu *FIX* gehörenden Binärprogramme. Für den Entwickler von Interesse sind zunächst das *FIX*-Entwicklermenü **fxm** und einige Utilities:

fgmask [<option> …] <file> [<basename>]
fgmenu [<option> …] <file> [<basename>]
fxm [-7 | -8]
led [-7 | -8] [<file>]
mdemo <file> [<option> …]
mlyconv [-t pan] [-7 | -8] [-ocs <outcharset>] [-o <outfile>] [-cs <charset>] -h <height> -w <width> <file>
panconv [-t <outformat>] [-ocs <outcharset>] [-o <outfile>] [<file>]
perfix <file> [<option> …]
perfix-p <file> [<option> …]
rled [-7 | -8] [<file>]

msgprep [-7 | -8] <file> [<outfile>]
msgdeprep [-7 | -8] <file>
less [<option> …] <file> …
lpcat [-c] [-7 | -8] [-d<gerät>] [<file>]
fxdisplay [-n] <string> …
tcout <capability>
tgoto <row> <column>

Von der Kommandoebene aus werden sie üblicherweise über Scripts gleichen Namens in $FXDIR/cmd gestartet (vgl. Seite 71).

Hinweis:

Die zweite Gruppe ist auch im Runtime-System enthalten.

Weiter finden sich hier die Programme zur Registrierung und Steuerung des Lizenz-Servers:

fxlicense
liserver
likill
fxlishow

und zum Erstellen einer Tastaturbeschreibung:

keyinit

Eine dritte Gruppe bilden selten benötigte, aber manchmal nützliche Utilities; hierzu gehören

datum [<datum>] [-t<tage>] [-w<wochen>] [-m<monate>] [-j<jahre>]
datum <datum1> -d<datum2>
fixsizes
jahr [<datum>] [-t<tage>] [-w<wochen>] [-m<monate>] [-j<jahre>]
monat [<datum>] [-t<tage>] [-w<wochen>] [-m<monate>] [-j<jahre>]
tag [<datum>] [-t<tage>] [-w<wochen>] [-m<monate>] [-j<jahre>]
tausch (<old> | -o<oldfile>) (<new> | -n<newfile>) [-c<anz>] file [-f<targetfile>] [-a] [-z]

xcho

Sie sind in `$FXDIRSYS/bin/READ_ME` kurz beschrieben.

Die Programme **datum**, **tag**, **monat**, **jahr** bewerten ihre Argumente unabhängig von der Anordnung der Schalter -t*n*, -m*n* -j*n* in der Aufrufzeile in der Reihenfolge “Anzahl Jahre - Anzahl Monate -Anzahl Tage”.

Bei den übrigen Programmen handelt es sich um nicht für den Entwickler bestimmte, aber von *FIX* benutzte Hilfsprogramme.

12 Attribute von Objekten

Ein Objekt wird definiert durch seine Attribute. Unter *Attribut* werden hierbei alle konstanten oder sich dynamisch ändernden Eigenschaften des Objekts verstanden. Die Attribute, die direkt der Beschreibungsdatei entnommen oder aus deren Angaben abgeleitet werden, werden als *statische Attribute* bezeichnet (einige können gleichwohl per Programm modifiziert werden).

Die folgenden Abschnitte geben einen kurzen Überblick über die statischen Attribute, der in den nachfolgenden Kapiteln dann vertieft wird.

1 Menüs

Zu den statischen Attributen eines Menüs zählen

- der Name, der das Menü eindeutig identifizieren sollte
- Höhe, Breite und Position des Menü-Windows

sowie optional

- Titel, Rahmen und Layout für das Window
- anwendungsspezifische Information
- ein Prompttext
- eine C-Funktion, die die Anwendungslogik realisiert
- Kriterien zur Anwahl der Menüpunkte, z.B.
  soll bei der Anwahl über Buchstaben zwischen Groß- und Kleinschreibung unterschieden werden?
- Kriterien zur Auswahl von Menüpunkten, z.B.
  soll ein mittels Anfangsbuchstabe ausgewählter Menüpunkt sofort ausgelöst werden?

Zu den statischen Attributen eines Menüpunktes zählen

- seine Position als Element
- der Name, der den Menüpunkt eindeutig identifizieren sollte und zugleich den zugehörigen Hilfetext angibt
- der Text, durch den der Menüpunkt im Window dargestellt wird
- die Position, an der der Text im Window platziert wird
- die Art der Hervorhebung, mit der der Text bei Anwahl dargestellt wird
- die Aktion, die beim Auslösen des Menüpunktes erfolgt

sowie optional

- anwendungsspezifische Information
- ein Prompttext
- ein Iconset (für *FIX/Win* oder *FIX/Web*)

2 Masken

Zu den statischen Attributen einer Maske gehören u. a.

- der Name, der die Maske eindeutig identifizieren muss
- Höhe, Breite und Position des Masken-Windows
- das Element, bei dem die Maskenbearbeitung beginnen soll (*Startelement*; wenn nicht explizit angegeben, das erste Element)

sowie optional

- Titel, Rahmen und Layout für das Window
- anwendungsspezifische Information
- ein Prompttext
- eine C-Funktion, die die maskeneigene Anwendungslogik realisiert
- das Verhalten bei Bedienung des Programms mit der Maus
- eine Aktion, die beim Verlassen der Maske ausgeführt wird
- Varianten
- Angaben zur Datenbeschaffung, die *FIX* bei der Bearbeitung der Maske berücksichtigt

bei allen Mehrsatz-Masken

- ob eine Satzanzeige mitzuführen ist, die Auskunft über den aktuellen und die insgesamt vorhandenen Sätze gibt (Default: ja)

bei Tabellenmasken (die ersten drei Attribute werden von *FIX* berechnet)

- die Anzahl der gleichzeitig darstellbaren Sätze
- die Anzahl der Bildschirmzeilen, die ein einzelner Satz belegt
- der Abstand zwischen oberem Rand des Windows und dem obersten darin dargestellten Satz

sowie optional

- Zeilentypen und eine Vorschrift zur Bestimmung des gültigen Zeilentyps

Zu den statischen Attributen eines Feldes zählen

- seine Position als Element
- der Name, der das Feld eindeutig identifizieren muss und zugleich den zugehörigen Hilfetext angibt
- der Datentyp und ob NULL unterstützt wird
- die Position, an der das Feld im Window platziert wird
- das Format (Formatstring bzw. Länge)
- die Feldeigenschaften

sowie optional

- anwendungsspezifische Information
- ein Prompttext
- ein Iconset (für *FIX/Win* oder *FIX/Web*)
- ein Defaultwert
- Wertvorgaben in Form einer Auflistung
- ein Verweis auf ein Auswahl-Objekt - ein Selo oder eine Choice - und die Art seiner Verwendung
- Plausibilitätsprüfungen in Form regulärer Ausdrücke oder C-Funktionen
- eine Bindung an eine Datenbankspalte
- ein Bezeichner für eine Variable

- eine Linkbeziehung zu einem anderen Feld

Zu den statischen Attributen einer eingebetteten Maske zählen

- ihre Position als Element
- der Elementtyp
- die Art der Einbettung

3 Selos

Zu den statischen Attributen eines Selo gehören

- der Name
- die Breite des Selo-Windows (Höhe und Position werden i. Allg. dynamisch bestimmt)
- die Struktur der Datensätze
- die Datenbeschaffungsvorschrift

sowie optional

- eine Position für das Window
- anwendungsspezifische Information
- eine Vorgabe für die Anzahl der gleichzeitig sichtbaren Sätze

Ein individueller Prompttext wird bei Selos nicht unterstützt.

Zu den statischen Attributen der Selo-Elemente (Spaltenreferenzen) gehören

- die Position als Element (dynamisch bestimmt)
- das Darstellungsformat (wenn nicht vorgegeben, wird ein Defaultformat benutzt)
- die Überschrift (wenn nicht vorgegeben, wird der Spaltenname benutzt)

sowie optional

- eine Vorschrift zur Übernahme des Wertes im ausgewählten Satz in ein Maskenfeld

4 Choices

Zu den statischen Attributen einer Choice gehören

- der Name, der die Choice eindeutig identifizieren sollte
- Höhe, Breite und Position des Choice-Windows
- die Darstellungsmatrix, angegeben durch Anzahl Zeilen, Anzahl Spalten und Spaltenabstand
- die Vorschrift, wie ein Datensatz darzustellen ist

sowie optional

- Titel, Rahmen und Layout für das Window
- anwendungsspezifische Information
- ein Prompttext
- die Struktur der Datensätze

- die Datenbeschaffungsvorschrift
- eine Vorschrift, um die Selektion mit einem Feldwert abzugleichen, solange sich die Choice passiv auf dem Bildschirm befindet
- eine Vorschrift, ob die vorhandene Selektion erhalten bleiben soll, wenn die Choice erneut aktiv wird
- Kriterien zur Selektion, z.B.
  soll bei der Selektion über Buchstaben zwischen Groß- und Kleinschreibung unterschieden werden?
  ist die Anzahl der selektierbaren Sätze nach unten und/oder oben beschränkt?
  wird die Choice bei Erreichen der vorgegebenen Anzahl selektierter Sätze automatisch verlassen?
- eine C-Funktion, die nach einer Selektion automatisch aufgerufen wird, um diese auszuwerten

5 Hilfetexte

Zu den statischen Attributen eines Hilfetextes gehören

- der Name (identisch mit dem der Datei, die den Hilfetext enthält)
- der Text
- die Seiteneinteilung des Textes
- Höhe und Breite des Hilfetext-Windows (automatisch aus dem Text bestimmt)

13 Die Erstellung von Menüs

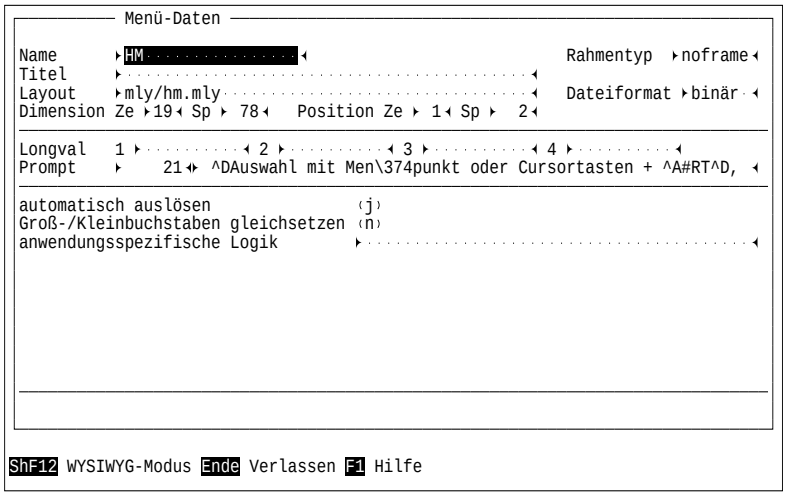
Zum Modifizieren von Menüs stellt *FIX* das Tool **led** zur Verfügung, das Sie entweder aus dem Entwicklermenü heraus benutzen oder mittels

led [<beschreibungsdatei>]

von der Shell aus aufrufen können. Menübeschreibungsdateien werden nach Konvention im Unterverzeichnis `men` abgelegt.

Nach dem Start von **led** (und ggf. der Auswahl des zu bearbeitenden Menüs) sehen Sie eine Maske, in der Sie die Definition Ihres Menüs ändern können:

Abb. 18 Definition eines Menüs



1 Die statischen Attribute eines Menüs

Name

FIX benutzt den Namen eines Menüs, der das Objekt eindeutig identifizieren sollte, bei der Source-Generierung.

Bei *Name* geben Sie den Namen des Menüs ein. Der Name darf nur aus Buchstaben, Ziffern und dem Zeichen '_' bestehen.

Rahmentyp

Der übliche Rahmen um ein Objekt-Window kann bei einem Menü wahlweise entfallen.

Bei *Rahmentyp* wählen Sie, ob das Menü-Window einen Rahmen besitzen soll (**frame**) oder nicht (**noframe**).

Titel

Ein Menü besitzt optional einen Titel, der am linken oberen Rand des Windows eingeblendet wird (wie “Menü-Daten” in Abb. 18).

Bei *Titel* können Sie einen Titel angeben. Der angegebene Text darf die *FIX*-spezifischen Umschaltcodes für Videoattribut und Zeichensatz enthalten (siehe Seite 112). ‘#’, gefolgt von einer Ziffernfolge, interpretiert *FIX* als Nummer eines Textes aus der Datei `messages`; als Titel wird dann dieser Text verwendet.

Layout

Ein Menü kann mit einem Layout versehen werden, d.h. einen Hintergrund, auf dem Rahmen, Titel und Menüpunkte erscheinen. Layouts werden getrennt von der übrigen Objektbeschreibung abgelegt.

Bei *Layout* muss der Name der Datei stehen, die das Layout des Menü enthalten soll. Nach Konvention ist dies bei *Dateiformat* “ASCII” die Datei `pan/<menüname>.pan`, bei *Dateiformat* “binär” die Datei `mly/<menüname>.mly`.[1] Ein mit **led** erstelltes Layout kann nur gespeichert werden, wenn hier eine Angabe gemacht ist.

Wenn Sie als Layout-Datei `/dev/null` angeben, wird kein Layout gespeichert, aber beim Laden der Maske ein “weißes” Layout erzeugt, das per Programm modifizierbar ist (→ layout_putstr()).

Dimension

Ein Menü besitzt eine feste Dimension.

Bei *Dimension Ze* geben Sie die Höhe des Windows in Zeilen an.

Bei *Dimension Sp* geben Sie die Breite des Windows in Spalten an.

Position

Ein Menü besitzt eine feste Position.

Bei *Position Ze* geben Sie den vertikalen Abstand des oberen linken Eckpunktes des Windows zu dem des Bildschirms an.

Bei *Position Sp* geben Sie den horizontalen Abstand des oberen linken Eckpunktes des Windows zu dem des Bildschirms an.

Anwendungsspezifische Information

Zu einem Menü kann anwendungsspezifische Information hinterlegt werden, die sich *FIX* merkt, selbst aber nicht verwendet. Diese umfasst u.a. acht long-Werte.

Bei *Longval-1* bis *Longval-8* können Sie dem Menü Zahlen zuordnen, die *FIX* als longval1 bis longval8 hinterlegt (vgl. Seite 140). Unterstützt werden nur Werte zwischen 0 und 2147483647, wobei **led** 0 wie NULL behandelt. Die Eingabefelder für *Longval-5* bis *Longval-8* erreichen Sie über die Taste f5, wenn Sie in einem derFelder *Longval-1* bis *Longval-4* stehen.

Prompt

Ein Menü besitzt optional einen Prompttext. Das ist ein in der Datei `messages` gespeicherter Text, der am unteren Bildschirmrand erscheint, wenn das Menü aktiv ist.

Bei *Prompt* können Sie die Nummer eines Textes angeben. Als Quelle für die Anzeige im folgenden Feld wird nicht `messages`, sondern deren “Source”-Datei `fx_texte` benutzt. Ist die Umgebungsvariable FXMSG mit einem anderen Dateinamen besetzt, wird diese Datei verwendet.

Damit ein neuer Text bei der Ausführung des Menüs erscheint, muss `messages` neu erstellt werden.

Kriterien zur Anwahl

Üblicherweise wird bei der Anwahl eines Menüpunktes über einen Präfix des Menüpunkt-Textes zwischen Groß- und Kleinschreibung unterschieden.

1. vgl. Kapitel *18*.

Wenn Sie bei *Groß-/Kleinbuchstaben gleichsetzen* Ja angeben, wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Kriterien zur Auswahl

Üblicherweise muss in Menüs der angewählte Menüpunkt mittels der Taste RT explizit ausgelöst werden.

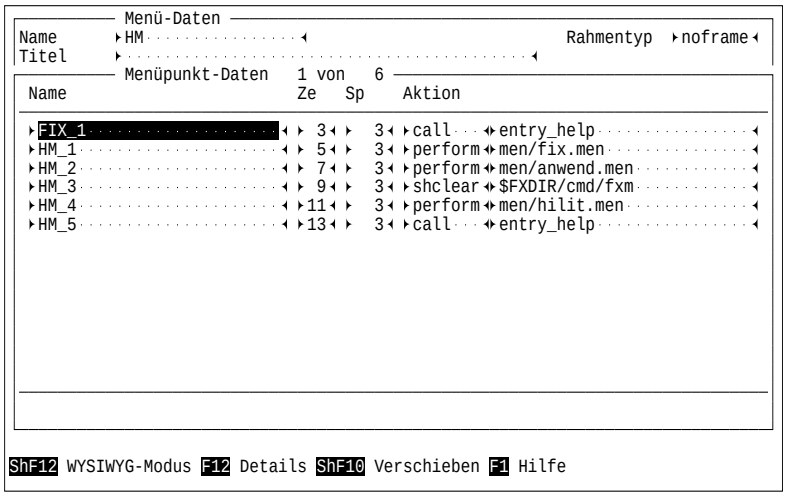
Wenn Sie bei *automatisch auslösen* Ja angeben, wird ein mittels seines Anfangsbuchstabens angewählter Menüpunkt sofort ausgelöst. Es sollte dann sichergestellt sein, dass die Anfangsbuchstaben der Menüpunkt-Texte paarweise verschieden sind.

Anwendungsspezifische Logik

Bei *anwendungsspezifische Logik* können Sie den Namen einer *FIX* bekanntgemachten, ins Programm eingebundenen C-Funktion angeben, die *FIX* bei der Bearbeitung des Menüs als Anwendungslogik benutzen soll.

Eine Menü kann bis zu 300 Menüpunkte besitzen, die Sie im **led** zunächst - in einer Tabellenmaske - im Überblick sehen.[1] Dabei gelangen Sie durch Drücken von g0 in einen Modus, in dem das aktuelle Element mit den Tasten ku und kd verschoben werden kann. Verlassen wird dieser Modus durch erneutes Drücken von g0.

Abb. 19 Überblick über die Elemente

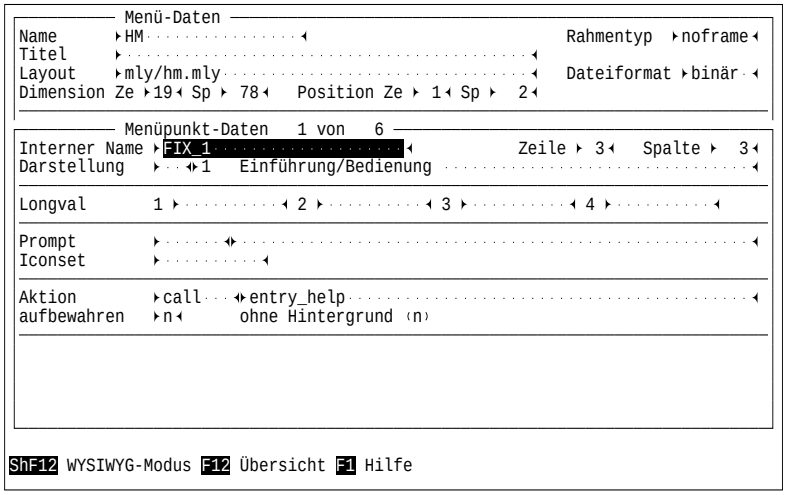


In der Statusanzeige sehen Sie die Anzahl der Elemente, hier 6.

Durch Drücken der Taste MD gelangen Sie in eine Detailmaske (Rollmaske), in der Sie die Definition der einzelnen Elemente vervollständigen oder modifizieren können.

1. vgl. aber Ressource StartWithElementTable auf Seite 119.

Abb. 20 Definition eines Menüpunktes



2 Die statischen Attribute von Menüpunkten

Name

Der Name wird benutzt als Name für eine Hilfetext-Datei.

Bei *Interner Name* geben Sie den Namen des Menüpunktes an. Der Name darf nur aus Buchstaben, Ziffern und dem Zeichen '_' gebildet sein.

Text und Art der Hervorhebung

Die auslösbare Aktion wird dem Anwender durch einen Text verdeutlicht.

Bei *Darstellung* legen Sie den Menüpunkt-Text und die Art fest, wie er hervorgehoben werden soll, wenn der Menüpunkt angewählt wird. Im ersten Feld geben Sie die Anzahl der Bildschirmzellen ein, die *FIX* bei Anwahl hervorheben soll. Bei 0 oder einem unzulässig kleinen Wert wird nur der Text hervorgehoben. Im folgenden Feld geben Sie den Text selbst an. '#', gefolgt von einer Ziffernfolge, interpretiert *FIX* als Nummer eines Textes aus der Datei `messages`: zur Darstellung des Menüpunktes wird dann dieser Text verwendet.

Leerzeichen am Ende des Textes sind nicht signifikant.

Position

Die Position legt fest, wo der Menüpunkt-Text innerhalb des Windows erscheint.

Bei *Zeile* geben Sie die vertikale Position des Textes *relativ zum oberen Menürand* an.

Bei *Spalte* geben Sie die horizontale Position des Textes *relativ zum linken Menürand* an.

Anwendungsspezifische Information

Zu einem Menüpunkt kann, wie zum Menü selbst, anwendungsspezifische Information hinterlegt werden (vgl. Seite 82).

Bei *Longval-1* bis *Longval-8* können Sie positive Zahlen als Werte für *longval1* bis *longval8* angeben. Die Eingabefelder für *Longval-5* bis *Longval-8* erreichen Sie über die Taste f5, wenn Sie in einem derFelder *Longval-1* bis *Longval-4* stehen.

Prompt

Ein Menüpunkt besitzt optional einen Prompttext. Das ist ein in der Datei `messages` gespeicherter Text, der am unteren Rand des Menüs erscheint, wenn dieser Menüpunkt angewählt wird und das Menü so dimensioniert ist, dass zwischen unterstem Menüpunkt und Rand mindestens eine Zeile verbleibt.

Bei *Prompt* können Sie die Nummer des Textes angeben. Näheres siehe Seite 82.

Iconset

Einem Menüpunkt kann optional die ID eines Iconset zugeordnet werden (vgl. “Icons” auf Seite 259). Diese Angabe ist nur relevant, wenn das Programm, das das Menü benutzt, durch *FIX/Win* oder *FIX/Web* bedient wird.

Bei *Iconset* können Sie die ID eines Iconset[1] angeben.

Aktion

Mit einem Menüpunkt können folgende Aktionen verbunden werden:

- das Ausführen eines Kommandos (Programm),
- der Aufruf einer in das Programm eingebundenen und *FIX* bekanntgemachten parameterlosen Funktion,
- die Bearbeitung eines Menüs, einer Maske oder einer Choice: das Objekt wird geladen, mittels perform() bearbeitet[2] und anschließend gelöscht;
- die Bearbeitung einer Maske mittels der Funktion perfix().

Bei *Aktion* wählen Sie im ersten Feld die Art der Aktion, die Sie mit einem Menüpunkt verbinden möchten, und geben im folgenden Feld deren Argument an. Als Aktion können Sie wählen:

- **sh**: als Argument ist ein Kommando anzugeben, das mittels /bin/sh -c "*arg*" ausgeführt werden kann. Es sollte keine Shell-Metazeichen enthalten; ggf. kann das Kommando in einem Script hinterlegt werden, das vom Menü gestartet wird.
- **shclear**: wie **sh**, jedoch wird zuvor der Bildschirm gelöscht und das Programm anschließend erst nach Drücken der Return-Taste fortgesetzt, so dass auch Ein-/Ausgabe vom bzw. auf das Terminal möglich ist.
- **call**: Argument ist der für die Funktion vergebene Name.
- **perform**: Argument ist der Name der Beschreibungsdatei.
- **perfix**: Argument ist der Name der Maskenbeschreibungsdatei.

Üblicherweise bleiben alle zum Zeitpunkt des Auslösens der Aktion am Bildschirm vorhandenen Objekte während der Ausführung dargestellt.

Wenn Sie bei *ohne Hintergrund* Ja angeben, verdeckt *FIX* für die Dauer der Ausführung der Aktion die augenblicklich am Bildschirm befindlichen Windows. Dies kann sinnvoll sein, wenn die Ausgabe der Aktion nicht mit dem vorhandenen Bildschirminhalt vermengt werden soll.

Im Normalfall werden die zu bearbeitenden Objekte geladen, ausgeführt und anschließend gelöscht.

Die Angabe Ja bei *aufbewahren* ist nur sinnvoll, wenn die Aktion das Ausführen eines Menüs ist. Bei Ja wird das Untermenü weiter im Speicher gehalten, braucht also bei Wiederbetreten nicht neu geladen zu werden. Es wird erst gelöscht, wenn das übergeordnete Menü gelöscht wird.

1. Als IDs sind ganze Zahlen aus dem Intervall [1000 - 2147483647] zulässig.
2. An ein Untermenü gibt *FIX* die Logik weiter, die für das übergeordnete Menü benutzt wird, bei anderen Objekten wird die Defaultlogik weitergegeben.

3 Beschreibungsdatei

Vorbemerkung zur Syntax von Beschreibungsdateien

FIX zerlegt eine Beschreibungsdatei beim Lesen in durch Leerzeichen, Tabulatoren und Kommata getrennte Symbole; hierbei ist ein Komma ein Symbol für sich. Symbole können optional in Anführungszeichen eingeschlossen werden; obligatorisch ist dies nur bei Leerstrings oder wenn ein Symbol Leerzeichen, Tabulatoren, Kommata oder Anführungszeichen enthält. Ein Anführungszeichen innerhalb eines Symbols muss durch ein vorangestelltes '\' geschützt werden.

Beginnt ein Symbol mit dem Zeichen '$', gefolgt von einem Bezeichner (d.h. einem Buchstaben und einer beliebigen Folge von Buchstaben, Ziffern und dem Zeichen '_'), so wird dieser Teil des Symbols als Platzhalter für eine Umgebungsvariable interpretiert und, sofern die Umgebungsvariable definiert ist, durch deren Wert ersetzt.

FIX kennt - im Gegensatz zu Programmiersprachen - keine Symbolklassen wie reservierte Wörter oder Zahlen. Vielmehr wird ein Symbol nur dann als Schlüsselwort interpretiert, wenn das gleichlautende Schlüsselwort an dieser Stelle zulässig ist; Analoges gilt für Ziffernfolgen und Zahlen.

Groß- und Kleinschreibung werden unterschieden. Zur Problematik von 8Bit-Zeichen vgl. Kapitel *48*.

Bei der Beschreibung von Dateiformaten steht

- Fettdruck für Symbole mit Schlüsselwort-Bedeutung
- <identifier> für einen Bezeichner
- <*xxx*datei> für einen Datei-Pfad
- <funktionsname> für den Namen, mit dem *FIX* eine C-Funktion bekanntgemacht wurde
- <integer> für eine beliebige ganze Zahl
- <unsigned> für eine vorzeichenlose ganze Zahl
- <string> für ein nicht näher bestimmtes Symbol.

Angaben in [ ] sind optional. Alternative Angaben sind durch | getrennt; ( ) fassen Angaben zusammen und + steht für eine beliebig häufige Wiederholung der vorangehenden Angabe.

Aus drucktechnischen Gründen sind manche Einträge der Beschreibung hier auf mehrere Zeilen verteilt im Stile von Fortsetzungszeilen wiedergegeben; *FIX* unterstützt diese Schreibweise nicht!

Da die Beschreibungsdatei eines Menüs - nach Konvention `men/<objektname>.men` - gewöhnlich mit dem Tool **led** erstellt wird, ist das Format hier ohne nähere Erläuterung wiedergegeben.

Die Beschreibungsdatei ist *zeilenorientiert*. Sie beginnt mit einem zweizeiligen Kopf, der die Menüattribute enthält.

menue <objektname> <titel> <layoutdatei> [**noframe**]
[**function** <funktionsname>] \
<hoehe> <breite> <ypos> <xpos> [**set** <daten>] [**prompt** <unsigned>] [**ignorecase**] \
    [**auto**]

wobei

<objektname> ::= <identifier>
<titel> ::= <string> | #<unsigned>
<hoehe> ::= <unsigned>
<breite> ::= <unsigned>
<ypos> ::= <unsigned>
<xpos> ::= <unsigned>
<daten> ::= <unsigned>
| <datenelement>[:<datenelement>]$^{\{1,7\}}$ anwendungsspez. Information
<datenelement> ::= <unsigned> |

Auf den Kopf folgen die Menüelemente. Jeder Menüpunkt belegt genau eine Zeile und muss in folgender Form angegeben sein:

<name> <y> <x> <markierung> <text> <aktion> [**set** <daten>] [**prompt** <unsigned>] \
[**iconset** <unsigned>] [**nobackground**] [**steady**]

wobei

<name> ::= <identifier>
<y> ::= <unsigned>
<x> ::= <unsigned>
<markierung> ::= <unsigned>
<text> ::= <string> | #<unsigned>
<aktion> ::= **sh** <string>
| **shclear** <string>
| **call** <funktionsname>
| **perform** <objektdatei>
| **perfix** <maskendatei>
<daten> ::= <unsigned>
| <datenelement>[:<datenelement>]$^{\{1,7\}}$ anwendungsspez. Information
<datenelement> ::= <unsigned> |

Abgeschlossen wird die Menübeschreibung durch eine Zeile, die mit dem Symbol

END_OF_MFO

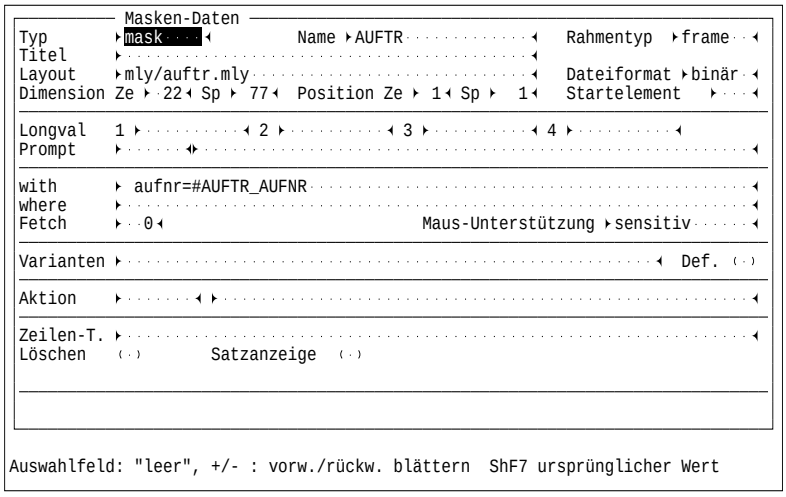
beginnt. Darauf noch folgende Zeilen behandelt *FIX* als Kommentar.

14 Die Erstellung von Masken

Zum Modifizieren von Masken können Sie ebenfalls das Tool **led** (vgl. Seite 81) benutzen. Maskenbeschreibungsdateien werden nach Konvention im Unterverzeichnis `mfo` abgelegt.

Nach dem Start von **led** (und ggf. der Auswahl der zu bearbeitenden Maske) sehen Sie eine Maske, in der Sie die Definition Ihrer Maske ändern können:

Abb. 21 Definition einer Maske



1 Die statischen Attribute einer Maske

Maskenart

FIX unterscheidet zwischen Einzelsatz-Masken und verschiedenen Formen von Mehrsatz-Masken.

Bei *Typ* wählen Sie die Art der Maske: **mask** - Einzelsatz-Maske, **submask** - Submaske, **rollmask** - Rollmaske, **table** - Tabellenmaske.

Name

FIX benutzt den Namen einer Maske beim Auswerten von Feld-Referenzen und bei der Source-Generierung.

Bei *Name* geben Sie den Namen der Maske ein. Der Name darf nur aus Buchstaben, Ziffern und dem Zeichen ‘_‘ bestehen.

Rahmentyp

Der übliche Rahmen um ein Objekt-Window kann bei einer Maske wahlweise entfallen.

Bei *Rahmentyp* wählen Sie, ob das Masken-Window einen Rahmen besitzen soll (**frame**) oder nicht (**noframe**) oder ob der Bereich des Rahmens leer gelassen werden soll (**empty**).

Titel

Eine Maske besitzt optional einen Titel, der am oberen Rand des Windows eingeblendet wird (wie “Masken-Daten” in Abb. 21).

Bei *Titel* können Sie einen Titel angeben. Der angegebene Text darf die *FIX*-spezifischen Umschaltcodes für Videoattribut und Zeichensatz enthalten (vgl. Seite 112). ‘#’, gefolgt von einer Ziffernfolge, interpretiert *FIX* als Nummer eines Textes aus der Datei `messages`; als Titel wird dann dieser Text verwendet.

Layout

Eine Maske kann mit einem Layout versehen werden, d.h einem Hintergrund, auf dem Rahmen, Titel und Felder erscheinen. Layouts werden getrennt von der übrigen Objektbeschreibung in einer eigenen Datei abgelegt.

Bei *Layout* muss der Name der Datei stehen, die das Layout der Maske enthalten soll. Nach Konvention ist dies bei *Dateiformat* “ASCII” die Datei `pan/<maskenname>.pan`, bei *Dateiformat* “binär” die Datei `mly/<maskenname>.mly`. Näheres siehe Seite 82.

Dimension

Eine Maske besitzt eine feste Dimension.

Bei *Dimension Ze* geben Sie die Höhe des Windows in Zeilen an.

Bei *Dimension Sp* geben Sie die Breite des Windows in Spalten an.

Position

Eine Maske besitzt eine feste Position.

Bei *Position Ze* geben Sie den vertikalen Abstand des oberen linken Eckpunktes des Windows zu dem des Bildschirms an.

Bei *Position Sp* geben Sie den horizontalen Abstand des oberen linken Eckpunktes des Windows zu dem des Bildschirms an.

Anwendungsspezifische Information

Zu einer Maske kann anwendungsspezifische Information hinterlegt werden, die sich *FIX* merkt, selbst aber nicht verwendet. Diese umfasst u.a. acht long-Werte.

Bei *Longval-1* bis *Longval-8* können Sie der Maske Zahlen zuordnen, die *FIX* als longval1 bis longval8 hinterlegt (vgl. Seite 140). Unterstützt werden nur Werte zwischen 0 und 2147483647, wobei **led** 0 wie NULL behandelt. Die Eingabefelder für *Longval-5* bis *Longval-8* erreichen Sie über die Taste f5, wenn Sie in einem derFelder *Longval-1* bis *Longval-4* stehen.

Prompt

Eine Maske besitzt optional einen Prompttext. Das ist ein in der Datei `messages` gespeicherter Text, der am unteren Bildschirmrand erscheint, wenn die Maske aktiv ist und kein anderer Prompt, z.B. der eines Maskenelements, Vorrang hat.

Bei *Prompt* können Sie die Nummer eines Textes angeben. Als Quelle für die Anzeige im folgenden Feld wird nicht `messages`, sondern deren “Source”-Datei `fx_texte` benutzt. Ist die Umgebungsvariable FXMSG mit einem anderen Dateinamen besetzt, wird diese verwendet.

Damit ein neuer Text bei der Ausführung der Maske erscheint, muss `messages` neu erstellt werden.

Startelement

Jede Maske definiert ein Element, das beim Neubetreten der Maske bzw. eines Satzes als erstes besucht wird.

Bei *Startelement* können Sie die lfd. Nummer des Elements angeben, das als erstes besucht werden soll. Die Nummerierung der Elemente beginnt mit 0. Default ist das erste Element der Maske.

Angaben zur Datenbeschaffung

Diese Angaben sind nur für die Source-Generierung und die Standardlogik (→ perf()) relevant und regeln die Form und den Zeitpunkt der Datenbeschaffung. Details finden Sie im Kapitel *Source-Generierung für Masken*.

Bei with können Sie, durch Leerzeichen getrennt, eine Folge von Korrespondenzen angeben in der Form <datenbankspalte>=#<feldname>.

Bei *where* können Sie eine Bedingung eingeben, wie sie die where-Klausel eines select-Statements enthalten kann. Wo SQL eine Hostvariable erlaubt, können Sie mit #<feldname> auch auf den Wert eines Feldes Bezug nehmen.

Bei *Fetch* können Sie die lfd. Nummer eines Elements angeben. Als Default verwendet *FIX* das Startelement.

Angaben zur Maus-Unterstützung

Diese Angabe ist vor allem bei Einsatz eine Frontends wie *FIX/Win* relevant und bestimmt, wie *FIX* reagieren soll, wenn der Benutzer auf ein Feld dieser Maske klickt.

Bei *Maus-Unterstützung* können Sie zwischen “insensitiv” (ignorieren), “sensitiv” (positionieren) und “logische Taste” (kompatibel zu *FIX* 2.9.2) wählen.

Varianten

Eine Maske kann Varianten besitzen. Varianten sind in eigenen Beschreibungsdateien hinterlegt.

Bei *Varianten* können Sie, durch Leerzeichen getrennt, Beschreibungsdateien von Varianten angeben. Die Position in der Auflistung bestimmt die Nummer, unter der die Variante im Programm angesprochen wird (beginnend mit 1, 0 entspricht der Hauptmaske). Geben Sie bei *Def.* Ja an, wenn bei einem erneuten Betreten der Maske statt der zuletzt benutzten Variante stets die Hauptmaske aktiviert werden soll (Restart-Modus).

Aktion

Mit einer Maske kann ein Kommando verbunden werden, das *FIX* automatisch ausführt, wenn die Maske verlassen wird.

Bei *Aktion* wählen Sie im ersten Feld die Art der Aktion (keine, **call, sh**, **shclear**) und geben im nachfolgenden Feld deren Argument an. Zu den Aktionsarten vgl. Seite 85.

Bei **sh** und **shclear** ersetzt *FIX* im Argument $1, $2, … vor der Ausführung durch die Werte des ersten, zweiten, … Feldes der Maske, $∗ durch die Folge der Werte aller Felder.

Erweiterte Eigenschaften von Mehrsatz-Masken

Mehrsatz-Masken besitzen gegenüber Einzelsatz-Masken zusätzliche Eigenschaften.

Bei *Zeilen-T.* können Sie satzspezifische Beschreibungsdateien für Tabellenmasken angeben (Näheres hierzu auf Seite 99).

Löschen ist für zukünftige Erweiterungen vorgesehen.

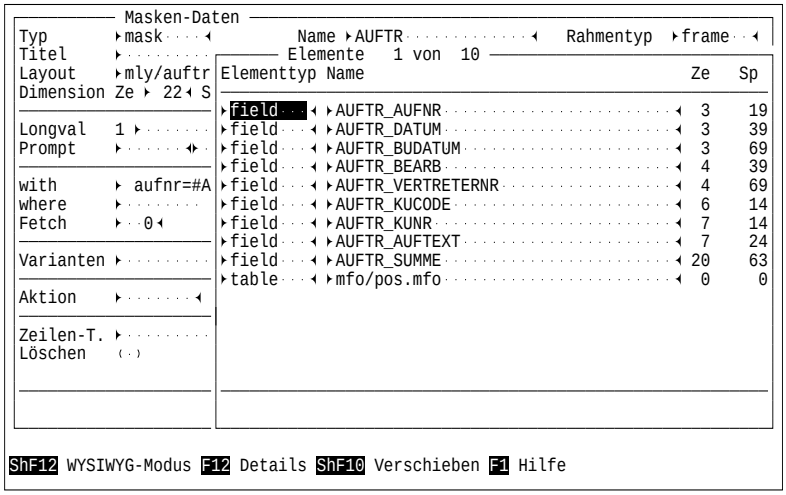
Geben Sie bei *Satzanzeige* Ja ein, wenn eine Satzanzeige “m von n” erfolgen soll.

Weitere statische Eigenschaften von Masken können nur bei der Einbettung in andere Masken oder per Programm definiert werden.

Eine Maske kann bis zu 300 Elemente besitzen, die Sie im **led** zunächst - in einer Tabellenmaske - im Überblick sehen.[1] Dabei gelangen Sie durch Drücken von g0 in einen Modus, in dem das aktuelle Element mit den Tasten ku und kd verschoben werden kann. Verlassen wird dieser Modus durch erneutes Drücken von g0.

1. vgl. aber Ressource StartWithElementTable auf Seite 119.

Abb. 22 Überblick über die Elemente



In der Satzanzeige sehen Sie die Anzahl der Elemente, hier zehn. Die ersten neun Elemente sind Felder, das letzte eine eingebettete Tabellenmaske.

Bei *Elementtyp* wählen Sie die Art des Elements: Feld (**field**) oder eingebettete Maske (**mask**, **submask**, **rollmask**, **table**).

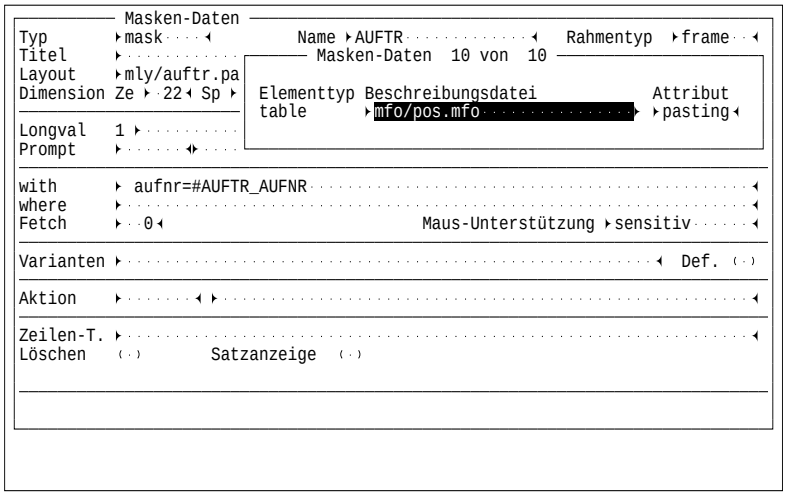
Bei *Name* legen Sie entweder den Namen des Feldes oder den der Beschreibungsdatei der eingebetteten Maske fest.

Durch Drücken der Taste MD kommen Sie in eine vom Elementtyp abhängige Detailmaske, in der Sie die Definition des Elements vervollständigen oder modifizieren können.

Definition eingebetteter Masken

Für eingebettete Masken hat diese Detaildarstellung folgende Form:

Abb. 23 Definition eingebetteter Masken



FIX kann die verschiedenen Typen von Mehrsatz-Masken aufeinander abbilden. So kann eine eigentlich als Tabellenmaske definierte Maske als Rollmaske eingebettet werden. Der Elementtyp hat Vorrang gegenüber der in der Beschreibungsdatei angegebenen Maskenart.

Unter *Elementtyp* sehen Sie die Art der Einbettung, wie Sie sie in der Übersichtstabelle ausgewählt haben.

Bei *Beschreibungsdatei* geben Sie den Namen der Beschreibungsdatei der Maske ein, die Sie einbetten möchten.

Attribute zur Window-Behandlung

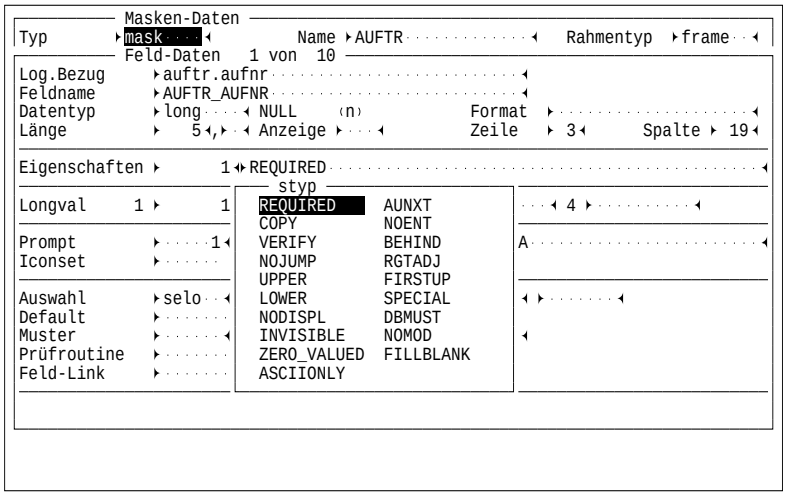
Standardmäßig ist eine eingebettete Maske nur sichtbar, während sie besucht wird. Wird sie **steady** eingebettet, bleibt sie nach der Bearbeitung weiter am Bildschirm, bis die einbettende Maske verschwindet. Bei **pasting** stellt *FIX* zusätzlich sicher, dass sich das Window der eingebetteten Maske stets über dem der einbettenden Maske befindet.

Bei *Attribut* können Sie die Art der Einbettung wählen: Default, **steady** oder **pasting**. *FIX* propagiert diese Eigenschaften an die eingebettete Maske.

Definition von Feldern

Für Felder hat die Detaildarstellung folgende Form:

Abb. 24 Definition von Feldern



2 Die statischen Attribute von Feldern

Logischer Bezug

Ein Feld entspricht häufig unmittelbar der Spalte einer Datenbanktabelle. Deshalb kann ein Feld bei der Definition mit einer Datenbankspalte assoziiert werden.

Bei *Log. Bezug* können Sie in der Form <db_tabelle>**.**<db_spalte> den Namen einer Datenbankspalte angeben, der das Feld entspricht. Diese Angabe ist für die Source-Generierung und perf() relevant und regelt die Art der Datenbankoperationen.

Alternativ können Sie den Namen für eine C-Variable angeben, die bei der Source-Generierung deklariert und zur Abspeicherung der Feldwerte benutzt werden soll. Das Feld wird dann - wie auch im Falle, dass überhaupt keine Angabe erfolgt - von der Standardlogik bei Datenbankoperationen nicht berücksichtigt.

Feldname

Der Name eines Feldes wird unter anderem als Name für eine Hilfetext-Datei und für Referenzen auf das Feld benutzt. Wird zu der Maske Programmcode generiert, findet er außerdem als Makro Verwendung.

Bei *Feldname* geben Sie den Namen des Feldes an. Der Name darf nur aus Buchstaben, Ziffern und dem Zeichen ‘_’ gebildet sein.

Feld-Link

Ein Feld kann als *Link* auf ein anderes Feld gleichen Datentyps und gleicher Länge definiert werden (vgl. “Feld-Links” auf Seite 182). Ein Feld-Link teilt sich mit dem Feld, auf das er sich bezieht, u.a. die Variable, in der der Feldwert gespeichert ist, wodurch sich eine Änderung in einem der Felder automatisch auf die anderen auswirkt. Der Entwickler hat selbst sicherzustellen, dass der Link keine Integritätsbedingungen verletzt, d.h. die beiden Felder sollten sich nur in Darstellungskomponenten, nicht aber in der logischen Bedeutung unterscheiden. Ein Link auf ein Feld derselben oder einer darin eingebetteten Maske ist nicht möglich.

Bei *Feld-Link* können Sie in der Form **#**[<maskenname>]**.**<feldname> ein Feld angeben, auf das das Feld gelinkt wird. Fehlt <maskenname>, wird *FIX* bei Auswertung des Links das Feld <feldname> in der die Maske einbettenden Maske suchen.

Datentyp

Die von *FIX* unterstützten Feldtypen sind im Kapitel “Felder” auf Seite 33 ausführlich erläutert.

Bei *Datentyp* wählen Sie den Datentyp des Feldes. **led** erlaubt nur die Auswahl aus *char*, *truth*, *graphics*, *short*, *long*, *float*, *double*, *money*, *decimal*, *date*, *datetime* und *interval*.

Datentyp	interne Typbezeichnung	Variablentyp
char	FXCHARTYPE	char[…]
truth	FXTRUTHTYPE	char[2]
graphics	FXGRAPHICSTYPE	char[…]
short	FXSHORTTYPE	short
long	FXLONGTYPE	long
float	FXFLOATTYPE	float
double	FXDOUBLETYPE	double
money	FXMONEYTYPE	double
decimal	FXDECIMALTYPE	dec_t
date	FXDATETYPE	long
datetime	FXDTIMETYPE	dtime_t
interval	FXINVTYPE	intrvl_t

Wenn Sie bei *NULL* Ja eingeben, unterstützt *FIX* bei diesem Feld NULL als Wert. Bei FXGRAPHICSTYPE, FXDATETYPE-, FXDTIMETYPE- und FXINVTYPE-Feldern ist dies automatisch der Fall.

Format, Länge

Das Format eines Feldes bestimmt, wie ein Feldwert dargestellt und eingegeben wird.

Bei *Format* können Sie ein Format angeben (vgl. Seite 35, Seite 39 und Seite 39). Bei FXCHARTYPE-, FXTRUTHTYPE-, FXGRAPHICSTYPE und FXDATETYPE-Feldern sind Formate nicht zulässig. Bei FXDTIMETYPE- und FXINVTYPE-Feldern ist die Angabe eines Formats obligatorisch. ‘#’, gefolgt von einer Ziffernfolge, interpretiert *FIX* als Nummer eines Textes aus der Datei `messages`; als Format wird dann dieser Text verwendet.

Wird kein Format vorgegeben, muss für das Feld eine Länge angegeben werden. Sie bestimmt, wie viele Zeichen die textuelle Darstellung des Feldwertes umfasst bzw. maximal umfassen kann. Da *FIX* voraussetzt, dass jederzeit zwischen Feldwert und textueller Darstellung konvertiert werden kann, muss sie hinreichend groß sein, um alle in der Anwendung auftretenden Werte wiederzugeben.

Im Regelfall bestimmt die Länge zugleich die Anzahl von Bildschirmzellen, mit denen der Feldinhalt im Window wiedergegeben wird; nur bei FXCHARTYPE-Feldern kann gesondert eine *Display-Länge* bestimmt werden: es wird dann nicht der gesamte Feldinhalt dargestellt, sondern nur ein Ausschnitt, der über den Feldwert gescrollt werden kann. Von diesen *Scrollfeldern* sollte der Entwickler nur sparsam Gebrauch machen, da stets die Gefahr besteht, dass der Anwender sich des nicht sichtbaren Teils des Feldwertes nicht bewusst ist. Aus diesem Grund werden sie auch nur bei CHARACTER-Feldern unterstützt.

Bei *Länge* geben Sie die Anzahl der Zeichen an, die das Feld maximal fassen kann ($1 \le l \le 2048$). Bei FXDATETYPE-Feldern sind nur die Längen 5 und 7 bzw. 6, 8 und 10 erlaubt (vgl. Seite 37).

Bei Feldern mit Typ FXFLOATTYPE, FXDOUBLETYPE und FXDECIMALTYPE können Sie im auf das Komma folgenden Feld die Anzahl der bei der Darstellung erwünschten Nachkommastellen ($0 \le l \le 9$) vorgeben.

Bei *Anzeige* können Sie eine Display-Länge angeben. Der Wert muss bei numerischen Feldern größer/gleich der Länge des Feldes sein, bei char-Feldern darf sie auch kleiner als die Länge des Feldes werden.

Dadurch ist es möglich, Felder mit unterschiedlichen Eingabelängen hinsichtlich der Begrenzer optisch gleich lang darzustellen. Bei Feldern, die rechts ausgerichtet sind, wird die linke Seite aufgefüllt, bei Feldern, die links ausgerichtet sind, die rechte Seite. Für die Füllstellen wird ein Zeichen des Grafikzeichensatzes verwendet (vgl. Seite 271).

Für Felder, die ein Format besitzen, kann keine Display-Länge (und auch keine Länge und keine Anzahl Nachkommastellen) angegeben werden. Hier ist die Ausrichtung über das Format zu definieren, indem an der entsprechenden Stelle Leerzeichen hinzugefügt werden.

Position

Ein Feld besitzt eine feste Position innerhalb des Windows.

Bei *Zeile* geben Sie die vertikale Position des Feldes *relativ zum oberen Maskenrand* an.

Bei *Spalte* geben Sie die horizontale Position des Feldes *relativ zum linken Maskenrand* an.

Feldeigenschaften

FIX kennt eine Vielzahl von Feldeigenschaften, die die Darstellung, die Logik der Bearbeitung etc. beeinflussen. Diese Eigenschaften können teilweise miteinander kombiniert werden. Welche Kombinationen sinnvoll sind, hängt vom Datentyp des Feldes ab (vgl. “Feldeigenschaften” auf Seite 44).

Das Feld *Eigenschaften* enthält eine Kodierung für die Feldeigenschaften. Sein Wert kann mittels einer Choice-Auswahl zusammengestellt werden. Das nachfolgende, nicht veränderbare Feld erläutert die Kodierung.

Anwendungsspezifische Information

Zu einem Feld kann, wie zur Maske selbst, anwendungsspezifische Information hinterlegt werden (vgl. Seite 90).

Bei *Longval-1* bis *Longval-8* können Sie positive Zahlen als Werte für *longval1* bis *longval8* angeben. Die Eingabefelder für *Longval-5* bis *Longval-8* erreichen Sie über die Taste f5, wenn Sie in einem derFelder *Longval-1* bis *Longval-4* stehen.

Prompt

Ein Feld besitzt optional einen Prompttext. Das ist ein in der Datei `messages` gespeicherter Text, der am unteren Bildschirmrand erscheint, wenn das Feld manuell erfasst wird. Wenn Sie keinen Prompt angeben, benutzt *FIX* den Prompttext der Maske (sofern definiert).

Bei *Prompt* können Sie die Nummer des Textes angeben. Näheres siehe Seite 90.

Iconset

Einem Feld kann optional die ID eines Iconset zugeordnet werden (vgl. “Icons” auf Seite 259). Diese Angabe ist nur relevant, wenn das Programm, das die Maske benutzt, durch *FIX/Win* oder *FIX/Web* bedient wird.

Bei *Iconset* können Sie die ID eines Iconset[1] angeben.

Defaultwert

Ein Feld besitzt einen *Defaultwert*, mit dem es beim Laden der Maske bzw. Anlegen eines neuen Satzes initialisiert wird.

Bei *Default* können Sie den zur Initialisierung des Feldes benutzten Wert eingeben.[2] Bei einem Feld-Link oder einem Feld mit Wertvorgabe (siehe unten) darf kein Wert angegeben werden, da sich deren Defaultwert bei Ersterem aus dem des Feldes, auf das der Link besteht, bei Letzterem aus dem in runde Klammern eingeschlossenen bzw. ersten vorgegebenen Wert ergibt. Schreiben Sie bei FXDATETYPE-Feldern “today”, wenn Sie das Tagesdatum wünschen.

1. Als IDs sind ganze Zahlen aus dem Intervall [1000 - 2147483647] zulässig.

2. Bei FXGRAPHICSTYPE-Feldern muss der Defaultwert z.Z. noch aus implementierungstechnischen Gründen mittels eines Texteditors in der entsprechenden Beschreibungsdatei nachgetragen werden.

(“today+*days*” oder “today-*days*” verhalten sich sinngemäß). Bei FXDTIMETYPE-Feldern ergibt “today” den Zeitpunkt, zu dem der Defaultwert ausgewertet wird.

Wenn Sie keinen Wert angeben, initialisiert *FIX* ein Feld mit 0/“ ” bzw. NULL.

Auswahl

Einem Feld kann ein Selo oder eine Choice als *Auswahl-Objekt* zugeordnet werden, was dem Anwender ermöglicht, aus einer Menge zulässiger Werte auszuwählen. Häufig wird das Feld selbst dann gar nicht dargestellt und sein Wert nur über die Auswahl zu verändern sein.

Bei *Auswahl* wählen Sie zunächst die Art der Auswahl: keine (Feld leer), Selo (**selo**) oder die verschiedenen Anschlussmodi für eine Choice (**use**, **offer**, **choice**).

use	*FIX* läßt ein Erfassen des Feldwertes nur über die Choice zu
offer	*FIX* bietet zur Feldwerterfassung zunächst die Choice an,[1] erlaubt aber auch eine direkte Änderung des Feldinhalts
choice	die Choice erscheint ähnlich wie ein Selo nur auf Anforderung

Wenn Sie eine Auswahl angeben, müssen Sie im folgenden Feld den Namen der entsprechenden Beschreibungsdatei angeben. Im Falle einer Choice können Sie im Feld dahinter “steady” oder “pasting” angeben.

Wertvorgabe und Wert-Muster

Ist die Menge der zulässigen Werte eines Feldes konstant und klein, kann sie durch *Aufzählung* vorgegeben werden. Der Anwender kann dann bei der Bearbeitung des Feldes nur zwischen diesen Werten blättern.

Eine allgemeinere Möglichkeit, Feldwerte einzuschränken, bieten *reguläre Muster*. Das Feld kann nur erfolgreich verlassen werden, wenn sein Inhalt aus dem regulären Ausdruck hergeleitet werden kann.

Bei *Muster* können Sie die erlaubten Feldwerte beschränken. Im ersten Feld wählen Sie die Art der Beschränkung: keine (Feld leer), Wertvorgabe (**values**) oder regulärer Ausdruck (**reg**). Bei **values** müssen Sie im folgenden Feld die zulässigen Werte, getrennt durch ‘|’, aufzählen[2]; bei **reg** geben Sie dort einen regulären Ausdruck an. Leerzeichen sind, außer am Feldende, signifikant.

Prüfroutine

Die flexibelste Möglichkeit, einen Feldwert auf Plausibilität zu prüfen, bietet eine eigene C-Routine. Einem Feld kann eine ins Anwendungsprogramm eingebundene und *FIX* bekanntgemachte C-Funktion zugeordnet werden, die nach dem Erfassen des Feldes den eingegebenen Feldwert akzeptieren muss, damit das Feld erfolgreich verlassen werden kann. Mehr hierzu finden Sie auf Seite 210.

Bei *Prüfroutine* können Sie den Namen angeben, unter dem Sie *FIX* die Funktion bekanntgemacht haben.

3 Beschreibungsdatei

Die Beschreibungsdatei ist *zeilenorientiert*. Sie beginnt mit einem zweizeiligen Kopf, der allgemeine Angaben enthält.

(**mask** | **submask** | **rollmask** | **table**) <objektname> <titel> <layoutdatei> [**noframe** | **emptyframe**]
<hoehe> <breite> <ypos> <xpos> [**set** <daten>] [**prompt** <unsigned>] \
[<start-element> [<fetch-element>]]

wobei

<objektname>	::= <identifier>
<titel>	::= <string> | #<unsigned>

1. nur, wenn das Feld leer ist und die Eigenschaft REQUIRED besitzt.

2. Bei FXGRAPHICSTYPE-Feldern muss die Wertvorgabe z.Z. noch aus implementierungstechnischen Gründen mittels eines Texteditors in der entsprechenden Beschreibungsdatei nachgetragen werden.

<hoehe> ::= <unsigned>
<breite> ::= <unsigned>
<ypos> ::= <unsigned>
<xpos> ::= <unsigned>
<daten> ::= <unsigned>
| <datenelement>[:<datenelement>]$^{\{1,7\}}$ anwendungsspez. Information
<datenelement> ::= <unsigned> |
<start-element> ::= <unsigned>
<fetch-element> ::= <unsigned>

Auf den Kopf folgen optionale Eigenschaften und Elemente der Maske. Zu den Eigenschaften zählen

die Art der Maus-Unterstützung:

(**disable** | **enable**) **mouse**

Varianten der Maske:

zoom (<maskendatei>)+ [**restart**]

Zeilentypen (nur bei Tabellenmasken):

use (<maskendatei>)+

eine Aktion, die bei Verlassen der Maske ausgeführt wird:

sh <string> | **shclear** <string> | **call** <funktionsname>

das Verhalten bei Verwendung als Submaske:

disable [**rowcount**] [**delete**]
with (<zuordnung>)+
where <string>

wobei

<zuordnung> ::= <column>=#<feldname> (*ein* Symbol !)

Jede der Angaben darf in der Beschreibung *höchstens* einmal vorkommen.

Jedes Element - dies ist entweder ein Feld oder eine eingebettete Maske - belegt genau eine Zeile und muss in folgender Form angegeben sein:

Feld:

field <name> <y> <x> <form_spec> <styp> <dtyp> \
[**NULL** | <variable> | <table>**.**<column>]

wonach in beliebiger Abfolge weitere optionale Angaben folgen können:

set <daten>
prompt <unsigned>
iconset <unsigned>
link #[<maskenname>].<feldname> (*ein* Symbol !)
selo <selodatei> | (**use** | **offer** | **choice**) <choicedatei> [**steady** | **pasting**]
(**values** | **regex**) <string>
check <funktionsname>
def <string>

wobei

<name> ::= <identifier>
<y> ::= <unsigned>
<x> ::= <unsigned>

```
<form_spec>          ::= <formatstring>
                       | #<unsigned>
                       | <feldlaenge>[.<nachkommastellen>|:<display_laenge>]
                                                                  (ein Symbol !)
<formatstring>       ::= <string>
<feldlaenge>         ::= <unsigned>
<nachkommastellen>::= <unsigned>
<display-laenge>     ::= <unsigned>
<styp>               ::= <unsigned>          (Kodierung für Feldeigenschaften)
<dtyp>               ::= <unsigned>          (Kodierung für Datentyp)
                         | <unsigned>:null
<variable>           ::= <identifier>
<table>              ::= <identifier>
<column>             ::= <identifier>
<maskenname>         ::= <identifier>
<feldname>           ::= <identifier>
<funktionsname>      ::= <identifier>
```

eingebettete Maske:

(**mask** | **submask** | **rollmask** | **table**) <maskendatei> [**steady** | **pasting**]

Abgeschlossen wird die Maskenbeschreibung durch eine Zeile, die mit dem Symbol

END_OF_MFO

beginnt.

Darauf noch folgende Zeilen behandelt *FIX* als Kommentar.

4 Maskenvarianten

Häufig möchte man dem Anwender die gleiche Information in verschiedenen Sichten anbieten, beispielsweise zunächst nur in Form einer groben Übersicht über die vorhandenen Sätze und dann, auf Wunsch, die Sätze im Detail.

FIX bietet hierzu eine komfortable Methode an: *Hauptmasken* mit *Varianten*. Beide werden wie gewöhnliche Masken definiert und können auch so benutzt werden. Für die Verwendung einer Maske als Variante zu einer Hauptmaske müssen allerdings eine Reihe von Bedingungen erfüllt sein:

- Die Hauptmaske muss die Beschreibungsdatei als Variante aufführen (vgl. Seite 91).
- Die Art der Variante darf nicht komplexer sein als die der Hauptmaske, d.h. eine Einzelsatz-Maske darf als Varianten nur Einzelsatz-Masken besitzen. Die verschiedenen Arten von Mehrsatz-Masken sind aber untereinander kombinierbar.
- Die Variante selbst darf keine Varianten oder Zeilentypen (vgl. hierzu Seite 99) enthalten.
- Die anwendungsspezifische Zusatzinformation, Angaben zur Datenbeschaffung (with, where, Fetch-Element) und Aktion werden aus der Hauptmaske übernommen: die Angaben in der Beschreibungsdatei werden bei der Verwendung als Variante ignoriert.
- Hauptmaske und Varianten müssen die gleichen Elemente besitzen, d.h. korrespondierende eingebettete Masken müssen identisch sein und korrespondierende Felder müssen bzgl. Name, Typ, NULL-Unterstützung, Format/Länge und logischer Bindung übereinstimmen.
- Hauptmaske und Varianten müssen hinsichtlich der Art der Mausunterstützung übereinstimmen.
- Feld-Links, anwendungsspezifische Zusatzinformation, Auswahl-Objekte, Defaultwerte und Feldprüfungen werden aus der Hauptmaske übernommen; die Angaben in der Beschreibungsdatei werden bei der Verwendung als Variante ignoriert. Eine Ausnahme bildet die Art der Choice-Anbindung (vgl. Seite 96): Hier kann die Variante dieselbe Choice in einem anderen Modus benutzen.

Unterscheiden dürfen sich Hauptmaske und Varianten somit hinsichtlich

- Maskenart (bei Mehrsatz-Masken)
- Maskenname
- Größe, Titel, Layout, Position
- Prompttext
- Satzanzeige
- Startelement

sowie die Felder hinsichtlich

- Position im Window
- Display-Länge (bei CHARACTER-Feldern)
- Feldeigenschaften (Ausnahme: ZERO_VALUED)
- Prompttext
- Iconset-ID
- Gebrauch einer zugeordneten Choice

FIX stellt dem Entwickler Funktionen zum Wechseln der Variante und zum Bestimmen der aktuellen Variante zur Verfügung. Gemäß der Standardlogik kann der Anwender die Variante mit der Taste MD wechseln.

Hinweis zur Programmierung:

Die Variante einer Maske wird nicht zusammen mit der Hauptmaske geladen, sondern erst, wenn erstmals zu der Variante umgeschaltet werden soll. Beim Umschalten der Variante werden die entsprechenden Attribute der Hauptmaske modifiziert (die aktuellen Werte werden vorher gerettet).

Bei Wechsel der Variante bleiben aktueller Satz und TOUCHED-Eigenschaft von Feldern unverändert. Beim Freigeben der Hauptmaske werden auch alle Varianten freigegeben.

5 Zeilentypen bei Tabellenmasken

Hinweis:

Hierbei handelt es sich um ein höchst komplexes Feature, das in Anwendungen sparsam benutzt werden sollte.

Innerhalb einer Tabellenmaske können Sätze unterschiedlich dargestellt werden. Hierzu muss neben der Tabellenmaske - die die Default-Darstellung beschreibt - jeder weitere *Zeilentyp* durch eine eigene Beschreibungsdatei definiert werden und in der Tabellenmaskenbeschreibung auf diese Dateien verwiesen werden (vgl. Seite 91).

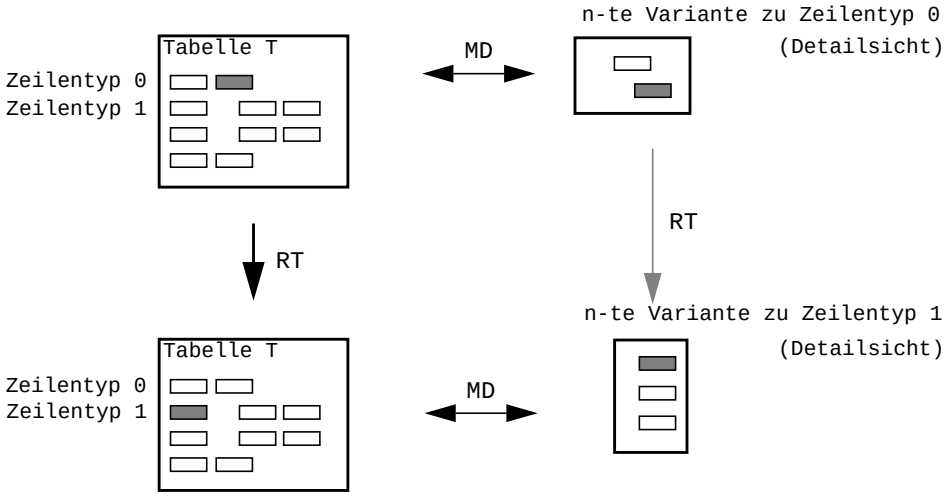
Für Zeilentypbeschreibungen gelten die gleichen Einschränkungen wie für Varianten von Masken (vgl. “Maskenvarianten” auf Seite 98). Es kommen aber weitere Beschränkungen hinzu:

- Jede Zeilentypbeschreibungsdatei muss eine Tabellenmaske beschreiben.
- Maskenname sowie Dimension, Lage und Rahmen des Windows müssen bei allen Zeilentypen übereinstimmen.
- Mit Ausnahme des Startelements werden alle nicht *elementbezogenen* Angaben wie Titel, Layout usw. ignoriert und aus der Grundmaske übernommen.
- Die Attribute *Anzahl Bildschirmzeilen pro Satz*, *Höchstzahl sichtbarer Sätze* und *Abstand des obersten Satzes vom oberen Window-Rand*, die beim Laden aus der Höhe der Maske und den Feldpositionen abgeleitet werden, müssen übereinstimmen.

Benutzt wird die unterschiedliche Darstellung erst, nachdem der Entwickler im Programm ein Feld und eine C-Funktion bestimmt hat, mit der *FIX* - in Abhängigkeit vom Wert dieses Feldes - den zum Satz passenden Zeilentyp bestimmen kann (→ t_discriminate()). Die Berücksichtigung des passenden Zeilentyps erfolgt dann ohne Zutun des Entwicklers.

Zu jedem Zeilentyp können eigene Maskenvarianten definiert werden. Programmgesteuert oder mittels der Taste MD ansprechbar sind jedoch nur die Varianten des Zeilentyps, dem der aktuelle Satz angehört. Bedingt - bei augenblicklichem Gebrauch der *n*-ten Variante - ein Wechsel des Satzes einen anderen Zeilentyp, schaltet *FIX* auf die *n*-te Variante des neuen Zeilentyps um, sofern eine solche existiert, anderenfalls kehrt *FIX* zur Hauptmaske des neuen Zeilentyps zurück.

Abb. 25 Beispiel



Hinweis zur Programmierung:

Zeilentypen werden von *FIX* unmittelbar im Anschluss an die Grundmaske geladen.

FIX wendet die vom Entwickler vorgesehene Funktion beim Neuaufbau der Darstellung (→ wrktoinfo()) des ausgezeichneten Feldes auf das Feld an. Liefert die Funktion eine Zahl $0 < n <$ Zahl der Zeilentypen, wird zur Darstellung des Satzes der *n*-te Zeilentyp, anderenfalls der der Grundmaske (≅ Zeilentyp 0) verwendet. Findet ein Wechsel des Zeilentyps statt, werden alle Felder des Satzes neu ausgegeben.

15 Die Erstellung von Selos

Die Beschreibungsdatei für ein Selo muss mit einem Texteditor erstellt werden[1]. Das Format der Datei ist auf Seite 104 aufgeführt. Ein Layout wird nicht unterstützt.

Beispiel 1

```
selo
    size 6                              - Maximalzahl der Sätze pro Seite
select kunr, code, name from kunde
    where code > " " ;                  - selektierte Menge
code "Code",                            - Spaltenreferenzen mit optionaler Überschrift
kunr "Kundennr",
name "Kundenname"
end
```

Beispiel 2

```
selo
execute procedure kunden(#)             - selektierte Menge und
into    kunr long, code char(8),          Satzstruktur der Ergebnismenge (Namen opt.)
        name char(25) ;
code "Code",                            - Spaltenreferenzen mit optionaler Überschrift
kunr "Kundennr",
name "Kundenname"
end
```

Hier werden nur jene Selos beschrieben, die auf SQL basieren; zu den alten, noch weiterhin unterstützten C-ISAM-Selos vgl. *Anhang B*.

1 Die statischen Attribute eines Selos

Name

Der Name eines Selos ist z.Z. ohne Bedeutung.

anwendungsspezifische Information

vgl. Seite 82.

Satzstruktur

Wird keine Satzstruktur vorgegeben (→ into-Klausel), leitet *FIX* sie von den vom Datenbanksystem mitgeteilten Spaltentypen ab.

1. Bei der Definition eines Feldes im **led** kann ein Selo-Prototyp generiert werden.

ESQL/C-Typ der Spalte	*FIX*-Typ der Spalte
CHAR(*n*), NCHAR(*n*)	FXCHARTYPE (gleiche Länge)
VARCHAR(*n*,…), NVARCHAR(*n*,…)	FXCHARTYPE (Maximallänge)
SMALLINT	FXSHORTTYPE
INTEGER, SERIAL	FXLONGTYPE
SMALLFLOAT	FXFLOATTYPE
FLOAT, MONEY	FXDOUBLETYPE
DECIMAL	FXDECIMALTYPE
DATE	FXDATETYPE
DATETIME	FXDTIMETYPE (gleicher Qualifier)
INTERVAL	FXINVTYPE (gleicher Qualifier)

Datenbeschaffungsvorschrift

Sie besteht, wie im Kapitel *Selos* beschrieben, entweder

- aus einer obligatorischen Grundanweisung, die die Ergebnismenge einer Datenbankanfrage definiert, und optionalen Zusatzbedingungen, die die Ergebnismenge weiter einschränken und nur in einem bestimmten Kontext zum Tragen kommen,

oder

- aus dem Aufruf einer SPL-Prozedur.

Näheres zur Form finden Sie auf Seite 105.

Anzeige- und Exportelemente

Aus der durch die Datenbeschaffungsvorschrift definierten Ergebnistabelle können Spalten zur Anzeige und/oder Wertübertragung ausgewählt werden. Die Reihenfolge und die Häufigkeit, mit der dies geschieht, bleibt dem Entwickler freigestellt. Jedes Element wird definiert durch:

Spaltenreferenz

Dies muss der Name einer Ergebnisspalte sein, wobei allerdings Groß- und Kleinschreibung nicht unterschieden wird. Der Name der Ergebnisspalte kann bei der Beschreibung der Satzstruktur vorgegeben werden (→ into-Klausel). Anderenfalls übernimmt *FIX* im Falle einer Pseudo-select-Anweisung die vom Datenbanksystem mitgeteilten Namen (bzgl. der Benennung der Ergebnisspalten von select-Anweisungen wird auf die IBM Informix ESQL/C-Dokumentation verwiesen). Für nicht explizit oder implizit benannte Spalten generiert *FIX* Namen der Form *_spaltennummer*.

Format

Dies muss folgende Form haben:

- Leerstring,
- Formatstring im Sinne von printf() mit einer geeigneten %-Umwandlung[1]
  (nicht bei FXDTIMETYPE und FXINVTYPE),
- Formatstring analog zum Format eines Feldes
  (nicht bei FXCHARTYPE, bei FXGRAPHICSTYPE und FXDATETYPE),
- *

Ist als Format * angegeben oder fehlt die Formatangabe (dies ist aufgrund der Syntax möglich), greift ein Defaultformat:

Typ der Spalte	Darstellung(sbreite)

1. Bei FXDATETYPE-Spalten muss dies %s oder %*n*s sein, wobei *n* ein als Feldlänge eines Datum-Feldes zulässiger Wert sein muss (vgl. Seite 37).

FXCHARTYPE(*n*)	*n*
FXGRAPHICSTYPE(*n*)	*n*
FXTRUTHTYPE	1
FXSHORTTYPE	5
FXLONGTYPE	8
FXFLOATTYPE	11
FXDOUBLETYPE	11
FXDECIMALTYPE	11
FXDATETYPE	6 (Tag, Monat, zweistelliges Jahr ohne Trennzeichen)
FXDTIMETYPE	19 (abhängig vom Qualifier, normkonform)
FXINVTYPE	19 (abhängig vom Qualifier, normkonform)

Überschrift

Dies ist entweder

- ein String,
- ein Verweis auf die Datei `messages`

oder

- *

Ist als Überschrift * angegeben oder fehlt die Angabe einer Überschrift (dies ist aufgrund der Syntax möglich), benutzt *FIX* den Namen der Ergebnisspalte als Überschrift.

Vorschrift zur Wertübernahme

Jedem Element *kann* ein Maskenfeld zugeordnet werden. Nach einer Auswahl im Selo wird versucht, den Spaltenwert in dieses Feld zu übernehmen. Ergebnisspalte und Maskenfeld müssen den gleichen oder zumindest einen ähnlichen Typ haben; bei numerischen Spalten versucht *FIX*, den Wert in den Feldtyp zu konvertieren, wenn dieser ebenfalls numerisch ist. Die Werte von Wahrheitswert-Spalten können nur in Wahrheitswert- oder CHARACTER-Felder, die von Semigrafik-Spalten nur in höchstens gleich lange Semigrafik-Felder übernommen werden.

Zwei Elementen darf nicht das gleiche Maskenfeld zugeordnet werden.

Bei der Festlegung von Format und Überschrift ist zu beachten:

- Wird das Defaultformat verwendet, wird die Überschrift auf dessen Länge verkürzt oder mit Leerzeichen aufgefüllt. Bei einem vorgegebenen Format setzt *FIX* voraus, dass Überschrift und Format zueinander passen, d.h. Überschrift und formatierte Werte gleichlange Zeichenketten ergeben.
- Wird als Format und Überschrift "" angegeben, so erscheint die Spalte nicht in der Ausgabe, d.h. auch das Trennzeichen zwischen den Spalten wird unterdrückt.

Breite

Die Breite des Selo-Window ergibt sich aus den Überschriften der angezeigten Spalten der Ergebnistabelle. Sie wird jeweils berechnet, wenn das Selo auf dem Bildschirm erscheint.

FIX platziert die angezeigten Spalten, durch ein Leerzeichen getrennt, nebeneinander und stellt ihnen eine Markierung der Form “ a ” voran. Die Breite des Selos ergibt sich in folgender Weise:

Rand - Lz. - Markierung - Lz. - Spalte - Lz. - … - Lz. - Spalte - Lz. - Rand

Höhe, Anzahl gleichzeitig sichtbarer Sätze

Im Regelfall bestimmt *FIX* die Höhe eines Selo-Windows zu dem Zeitpunkt, zu dem es auf dem Bildschirm erscheint, in Abhängigkeit von dem Feld, zu dem das Selo gehört, so, dass dieses Feld möglichst nicht verdeckt wird. Mittels einer **size**-Angabe kann der Entwickler aber die Anzahl Sätze je Seite und damit indirekt die Höhe vorgeben.

Position

Im Regelfall bestimmt *FIX* die Position eines Selo-Window zu dem Zeitpunkt, zu dem es auf dem Bildschirm erscheint, in Abhängigkeit von dem Feld, zu dem das Selo gehört, so, dass dieses Feld möglichst nicht verdeckt wird.

Mittels einer **at**-Angabe kann der Entwickler aber die Platzierung an einer bestimmten Stelle des Bildschirms erzwingen.

2 Beschreibungsdatei

Ein Selo wird beschrieben durch eine Selobeschreibungsdatei, nach Konvention sel/<objektname>.sel. Die Beschreibungsdatei eines Selos ist im Gegensatz zu der der meisten anderen Objekte nicht zeilenorientiert und hat folgenden Aufbau:

selo [<objektname>] [**at** <ypos> <xpos>] [**size** <anz>] [**set** <daten>]
(<query> [**restrict** <conditions>] | <procedure_call>)
<item>
(, <item>)*
end

wobei

<objektname> ::= <identifier>
<ypos> ::= <unsigned>
<xpos> ::= <unsigned>
<anz> ::= <unsigned>
<daten> ::= <unsigned>
| <datenelement>[:<datenelement>]$^{\{1,7\}}$ anwendungsspez. Information
<datenelement> ::= <unsigned> |

<query> ::= Liste von Symbolen, die mit dem Symbol select beginnt und mit einem Semikolon abgeschlossen ist;
optional mit einer into-Klausel wie unten
<conditions> ::= Liste von Symbolen, die mit einem Semikolon abgeschlossen ist

<procedure_call> ::= **execute procedure** <proc_name> (<proc_arg_list>)
[<into_clause>] ;
<proc_name> ::= SPL-Prozedur-Name
<proc_arg_list> ::= /* leer */
| <identifier> = <value> [, <identifier> = <value>] *
| <value> [, <value>] *
<value> ::= Zeichenketten-Konstante (in ' eingeschlossen)
| numerische Konstante
| TODAY
| Datetime-Literal
| CURRENT … TO …
| Intervall-Literal
| <integer> UNITS …
| NULL
| # vgl. Seite 27
| #<maskname>.<fieldname>
| #+<unsigned>
| #-<unsigned>
| ?<maskname>.<fieldname>
| ?+<unsigned>
| ?-<unsigned>

<into_clause> ::= **into** <column_spec> [, <column_spec>] *
<column_spec> ::= [<column_name>] <column_type>
<column_name> ::= <identifier>

```
<column_type>  ::= CHAR(<len>)                          → FXCHARTYPE
                 |  CHARACTER(<len>)                    → FXCHARTYPE
                 |  GRAPHICS(<len>)                     → FXGRAPHICSTYPE
                 |  TRUTH                               → FXTRUTHTYPE
                 |  SHORT                               → FXSHORTTYPE
                 |  LONG                                → FXLONGTYPE
                 |  FLOAT                               → FXFLOATTYPE
                 |  DOUBLE                              → FXDOUBLETYPE
                 |  DECIMAL [ (<prec>,<scale>) ]        → FXDECIMALTYPE
                 |  DATE                                → FXDATETYPE
                 |  DATETIME … TO …                     → FXDTIMETYPE
                 |  INTERVAL … TO …                     → FXINVTYPE
<len>          ::= <unsigned>    (in Bytes)
<prec>         ::= <unsigned>
<scale>        ::= <unsigned>

<item>         ::= <column_name> [ < heading> [ <format> [ <target> ] ] ]
<heading>      ::= <string> | #<msgnr>
<format>       ::= <string> | #<msgnr>
<target>       ::= #<maskname>.<fieldname>
                 |  #+<unsigned>
                 |  #-<unsigned>
```

Erläuterungen:

Referenzen der Form ‘#±*n*’ bzw. ‘?±*n*’ bezeichnen ein Maskenelement *relativ* zu dem Feld, an das das Selo gebunden ist. +*n* steht für das Maskenelement *n* Positionen hinter, -*n* für das *n* Positionen vor diesem Feld, das selbst mit +0 oder -0 angesprochen werden kann.

In der into-Klausel wird bei den Bezeichnungen der Typen nicht zwischen Groß- und Kleinschrift unterschieden. Sämtliche Bestandteile von Typbezeichnungen (z.B. YEAR, SECOND) werden als Schlüsselwort interpretiert und sind daher als Spaltenname unzulässig.

Die Symbolfolge <query> muss, zu einem String zusammengefasst, eine *Pseudo-select-Anweisung* ergeben. Darunter ist ein String zu verstehen, der eine Embedded-SQL-select-Anweisung bildet, die

1. keine Unterabfragen beinhaltet,
2. keine Vereinigungsmenge (union) konstruiert und
3. deren where-Klausel *anstatt Hostvariablen* Feld-Referenzen der Form #<maskname>.fieldname enthalten kann[1].

So wie bei der Ausführung einer Embedded-SQL-Anweisung im Programm Hostvariablen-Bezeichner durch die Werte dieser Variablen ersetzt werden, so ersetzt *FIX* Feld-Referenzen durch die Werte dieser Felder.

Der aus der Symbolfolge <conditions> gebildete Text muss eine zulässige Erweiterung der where-Klausel der Anweisung <query> bilden in folgendem Sinne:

- Enthält die Anweisung bereits eine where-Klausel, muss der Text hinter dem Schlüsselwort **where** eingefügt und mit dem Operator **and** mit den ursprünglichen Bedingungen verknüpft werden können.
- Enthält die Anweisung keine where-Klausel, kann das Schlüsselwort **where** und der Text hinter der Tabellenliste der from-Klausel ein- bzw. anfügt werden.
- Neben Feld-Referenzen der obigen Form darf die Symbolfolge <conditions> solche der Form **?**<maskname>.<fieldname> enthalten (vgl. hierzu Seite 26).

1. Aus Kompatibilitätserwägungen werden auch Referenzen der Form #<fieldname> unterstützt, von deren weiterer Verwendung jedoch abgesehen werden sollte. *FIX* erwartet bei einer solchen Referenz das Feld in der Maske, bei deren Bearbeitung das Selo aktiviert oder zur Feldprüfung eingesetzt wird.

16 Die Erstellung von Choices

Die Beschreibungsdatei für eine Choice muss mit einem Texteditor erstellt werden.[1] Das Format der Datei ist auf Seite 109 aufgeführt. Um ein Layout anzulegen - dies ist in der Praxis selten -, definiert man am besten eine entsprechend große Dummy-Maske ohne Felder.

Beispiel

```
choice FILESELECT "Dateiauswahl"          - Name und Titel
10 20 0 0                                 - Dimension und Position
8 1 3                                     - Darstellungsmatrix
data from sh "/bin/ls *.doc" into char(14); - Datenbeschaffung
range 1 1 auto                            - Auswahlkriterien
END_OF_MFO
```

Diese Beschreibung definiert eine Choice, die - in einem mit Rahmen versehenen Fenster der Größe 10 x 20 in der oberen linken Ecke des Bildschirms - in einer 8 x 1 Matrix mit 3 Zeichen Abstand vom linken und rechten Rand die augenblicklich im aktuellen Verzeichnis vorhandenen, auf .doc endenden Dateien anzeigt (oder genauer nur die ersten 12 Zeichen des Dateinamens, da Rahmen und Zwischenraum 8 der vorhandenen 20 Stellen benötigen).

Nach Auswahl einer Datei wird die Choice automatisch verlassen.

1 Die statischen Attribute einer Choice

Name

Er kann zur Identifikation der Choice benutzt werden, hat aber sonst für *FIX* keine Bedeutung.

Rahmentyp, Titel, Layoutdatei, Dimension und Position

vgl. Seite 81.

anwendungsspezifische Information und Prompttext

vgl. Seite 82.

Darstellungsmatrix

Die Datensätze einer Choice - bzw. ein Teil davon - werden in Matrixform dargestellt. Die Matrix wird durch drei ganze Zahlen - *zeilen*, *spalten*, *abstand* - definiert.

zeilen bestimmt, wie viele Elemente in vertikaler, *spalten*, wie viele in horizontaler Richtung dargestellt werden (wird *zeilen* ein '-' vorangestellt, verwendet *FIX* ggf. auch weniger Zeilen, wenn die Datenmenge dies zulässt). *abstand* definiert den Abstand der Spalten untereinander und vom Rand. Da die oberste Zeile für Titel und Satzanzeige reserviert ist, muss *zeilen* stets kleiner als die Höhe des Windows sein. *FIX* platziert die Matrix unmittelbar über dem unteren Rahmen bzw. am Fuße des Windows.

1. Bei der Definition eines Feldes im **led** kann ein Choice-Prototyp generiert werden.

Aus *spalten* und *abstand* ergibt sich, wie viele Zeichen zur Darstellung eines Datensatzes zur Verfügung stehen, wobei folgende Aufteilung angestrebt wird:

Rand - Abstand - Datensatz - Abstand - … - Abstand - Datensatz - Abstand - Rand

bzw.

Abstand - Datensatz - Abstand - … - Abstand - Datensatz - Abstand

Die Darstellung für einen Datensatz wird bestimmt durch eine C-Funktion (siehe unten); diese muss dem Satz einen String zuordnen, den *FIX* auf die Darstellungslänge verkürzt bzw. mit Leerzeichen auffüllt.

Satzstruktur

Die Datenmenge einer Choice besteht aus strukturierten Sätzen, d.h. jeder Satz ist gegliedert in Spalten eines bestimmten Typs. Wird keine Satzstruktur angegeben (die entsprechende Angabe ist optional), leitet *FIX* sie aus der Art der Datenbeschaffung ab.

Datenbeschaffung

Die Daten können beschafft werden (die Angaben in den Klammern beziehen sich auf die Syntaxbeschreibung)

1. durch das Anwendungsprogramm (wenn eine explizite Angabe fehlt)
2. aus einer beliebigen Datei (**file** <datendatei>)
3. aus der Standardausgabe eines Kommandos (**sh** <cmd>)
4. mittels Datenbankanfrage (<query>, <procedure_call>)

und werden intern durch eine Memory Relation (vgl. Seite 149) repräsentiert.

Die Datenmenge kann *dynamisch* bei jeder Benutzung, d.h. wenn die Choice aktiv wird oder sie zur Feldprüfung verwendet wird (vgl. Seite 214), oder einmalig beim Laden der Choice (→ **static data**) bestimmt werden.

Bei einer *dynamischen* Choice erhalten alle Datensätze nach dem Lesen automatisch den Zustand “nicht ausgewählt”. Datenmenge und vorgenommene Selektion bleiben bis zu einer Neubestimmung der Datenmenge erhalten. Bei einer *statischen* Choice hingegen bleibt nicht nur die Datenmenge, sondern i. Allg. auch die zuletzt getroffene Selektion bei einer erneuten Benutzung erhalten; durch den Zusatz **restart** wird *FIX* veranlasst, vor einer erneuten Bearbeitung alle Datensätze wieder in den Zustand “nicht ausgewählt” zu versetzen.

Fehlt eine Angabe zur Datenbeschaffung (Fall 1), geht *FIX* davon aus, dass der Choice im Anwendungsprogramm eine Datenmenge zugeordnet wird (→ ch_exchange()).

In den Fällen 2 und 3 muss die Ergebnismenge Datensätze mit genau einer CHARACTER-Spalte beinhalten. Wenn keine Satzstruktur vorgegeben ist, nimmt *FIX* als Länge 131 an. Jede gelesene Zeile (exklusive ‘\n’) ergibt einen Datensatz, wobei der über die Spaltenbreite hinausgehende Teil der Zeile verlorengeht. Zu kurze Werte werden mit Leerzeichen aufgefüllt.

Bei einer **file**-Angabe werden die Daten aus der angegebenen Textdatei gelesen (ein relativer Pfadname wird auf das Verzeichnis `$FXHOME` bezogen), bei einer **sh**-Angabe analog von der Standardausgabe des angegebenen Kommandos.

Weiter besteht die Möglichkeit, die Daten durch eine *Pseudo-select-Anweisung* oder eine SPL-Prozedur zu beschaffen. In diesem Fall ergibt sich die Spaltenstruktur, sofern nicht vorgegeben, aus der Ergebnismenge aus der Anweisung (vgl. “Satzstruktur” auf Seite 101).

Vorschrift zur Darstellung

Wie oben erwähnt, wird zur Darstellung eines Datensatzes eine Funktion benötigt, die dem Datensatz als Repräsentation einen String zuordnet. Mittels **display** *fct* kann der Entwickler hierzu eine in das Anwendungsprogramm eingebundene und *FIX* unter dem Namen *fct* bekanntgemachte C-Funktion bestimmen (Beispiele für solche Funktionen finden Sie in `$FXDIR/src`). Fehlt eine **display**-Angabe, verwendet *FIX* eine Standardfunktion (→ stddisplay()): bei Datensätzen, die mit einer CHARACTER-Spalte beginnen, benutzt diese Funktion deren Wert als Darstellung, anderenfalls liefert sie den Text “<data>”. Im Falle der Datenbeschaffung mittels Datenbankanfrage wird anstelle von stddisplay() die Funktion ch_display_columns() verwendet.

Kriterien zur Auswahl

Die Angabe **ignorecase** bewirkt, dass bei der Anwahl von Datensätzen mittels Anfangsbuchstaben nicht zwischen Groß- und Kleinschreibung unterschieden wird.

Eine **range**-Angabe veranlasst *FIX*, dem Anwender das *erfolgreiche* Verlassen der Choice (Event L_NXTFIELD) nur bei einer Mindest- und/oder Höchstanzahl ausgewählter Sätze zu erlauben. Sind beide gleich (bzw. 0 und 1) und ist zusätzlich das Schlüsselwort **auto** angegeben, wird die Choice automatisch verlassen, sobald die entsprechende Anzahl von Sätzen ausgewählt ist (Ausnahme: wenn sie gerade erst betreten wurde).

Die beiden folgenden Attribute sind nur relevant, wenn die Choice als Auswahl-Objekt mit einem Feld verknüpft ist. Hier bedingt meist die in der Choice getroffene Selektion den Feldwert, wie auch jede Änderung des Feldwertes eine Änderung der ausgewählten Sätze bedingt.

Vorschrift zur Auswahl-Abgleich

Eine **import**-Angabe veranlasst *FIX*, bei einem mit dieser Choice assoziierten Feld

- zu Beginn und nach der Erfassung des Feldinhalts mittels der Choice,
- sofern die Choice am Bildschirm dargestellt ist, am Ende des Besuchs des Feldes (→ fx_accept()),
- sofern die Choice am Bildschirm dargestellt ist und sich das Feld nicht in Bearbeitung befindet, beim Darstellen des Feldes (→ fdisp())

eine C-Funktion aufzurufen. Zweck dieser Funktion ist es, den Zustand der Choice an den Inhalt des Feldes anzupassen, d.h. abhängig vom Feldinhalt genau die korrespondierenden Datensätze als ausgewählt zu markieren.

Mittels **import** *fct* kann der Entwickler hierzu eine in das Anwendungsprogramm eingebundene und *FIX* unter dem Namen *fct* bekanntgemachte C-Funktion bestimmen (Beispiele für solche Funktionen finden Sie in `$FXDIR/src`). Fehlt der Funktionsname, d.h. ist nur **import** angegeben, verwendet *FIX* eine Standardfunktion (→ stdimport()): diese Funktion setzt voraus, dass der Typ der ersten Spalte und der des Feldes übereinstimmen und markiert genau die Datensätze als ausgewählt, bei denen der Wert der ersten Spalte mit dem Feldwert übereinstimmt; der erste (ausgewählte) Satz wird zum aktuellen Element der Choice.

Vorschrift zur Auswahl-Übernahme

Eine **export**-Angabe veranlasst *FIX*, bei einem mit der Choice assoziierten Feld nach der Erfassung des Feldinhalts mittels der Choice eine C-Funktion aufzurufen. Zweck dieser Funktion ist es, den Wert des Feldes an den Zustand der Choice anzupassen, d.h. einen zu den ausgewählten Datensätzen korrespondierenden Wert in das Feld zu schreiben.

Mittels **export** *fct* kann der Entwickler hierzu eine in das Anwendungsprogramm eingebundene und *FIX* unter dem Namen *fct* bekanntgemachte C-Funktion bestimmen (Beispiele für solche Funktionen finden Sie in `$FXDIR/src`). Fehlt der Funktionsname, d.h. ist nur **export** angegeben, verwendet *FIX* eine Standardfunktion (→ stdexport()): diese Funktion setzt voraus, dass der Typ der ersten Spalte und der des Feldes übereinstimmen und übernimmt den Wert der ersten Spalte aus dem ersten als ausgewählt markierten Datensatz als Feldwert; ist kein Datensatz ausgewählt, wird das Feld gelöscht.

2 Beschreibungsdatei

Die Beschreibungsdatei einer Choice - nach Konvention cho/<objektname>.cho - ist *zeilenorientiert*. Sie beginnt mit einem zweizeiligen Kopf, der allgemeine und objekttypische Attribute enthält.

choice <objektname> <titel> <layoutdatei> [**noframe** | **emptyframe**]
<hoehe> <breite> <ypos> <xpos> [**set** <daten>] [**prompt** <unsigned>] [**ignorecase**]

wobei

<objektname> ::= <identifier>
<titel> ::= <string> | #<msgnr>

```
<hoehe>          ::= <unsigned>
<breite>         ::= <unsigned>
<ypos>           ::= <unsigned>
<xpos>           ::= <unsigned>
<daten>          ::= <unsigned>
                     | <datenelement>[:<datenelement>]{1,7}    anwendungsspez. Information
<datenelement>   ::= <unsigned> |
```

Daran schließt sich eine Zeile mit der Definition zur Darstellungsmatrix an:

<zeilen> <spalten> <abstand>

wobei

```
<zeilen>:        ::= <unsigned> | -<unsigned>
<spalten>:       ::= <unsigned>
<abstand>        ::= <unsigned>
```

Auf den Kopf kann eine - optionale - Zeile mit der Angabe zur Datenbeschaffung (*data-Klausel*) folgen:

[**static**] **data from** <data_stmt>

wobei

```
<data_stmt>  ::= file <datendatei> [ <into_clause> ; ]
               |  sh <cmd> [ <into_clause> ; ]
               |  <query>
               |  <procedure_call>
<cmd>        ::= <string>
```

Zur Syntax von <query> und <procedure_call> siehe Seite 104 des Handbuchs. Bei beiden unterscheidet *FIX* - außer in geschützten Bezeichnern (von doppelten Hochkommata umgeben) und Zeichenketten-Konstanten (von einfachen Hochkommata umgeben) - nicht zwischen Groß- und Kleinschrift.

In Verbindung mit **static data from** <query> funktionieren relative Feldreferenzen (#+0 bzw. #feldname) nicht, da hier zum Zeitpunkt der Datenbeschaffung (beim Ladens der Choice) noch keine logische Verknüpfung zwischen der Choice und der sie verwendenden Maske besteht. Auch eine absolute Referenz auf diese Maske macht in Allgemeinen keinen Sinn, da deren Felder zu diesem Zeitpunkt stets noch ihre Defaultwerte besitzen.

Zeilenumbrüche innerhalb von <into_clause>, <query> und <procedure_call> werden von *FIX* wie Leerzeichen behandelt, d.h. beenden die data-Klausel nicht.

Verpflichtend ist eine Zeile, die Kriterien zur Auswahl beinhaltet:

[**range** <min> <max> [**auto**]] [**restart**]

wobei

```
<min>    ::= <unsigned>      (Mindestanzahl ausgewählter Optionen bei Verlassen)
<max>    ::= <unsigned>      (Höchstanzahl ausgewählter Optionen bei Verlassen)
```

Die Angabe von **restart** ist nur in Verbindung mit **static data** zulässig.

Daran können sich weitere Zeilen anschließen:

display <funktionsname>
import [<funktionsname>]
export [<funktionsname>]

Abgeschlossen wird die Choicebeschreibung durch eine Zeile, die mit dem Symbol

END_OF_MFO

beginnt. Nachfolgende Symbole werden ignoriert.

17 Die Erstellung von Hilfetexten

1 Einführung

Hilfetexte können erstellt, modifiziert oder gelöscht werden, ohne eine Änderung im Anwendungsprogramm vorzunehmen. Dies kann sogar geschehen, während die Anwendung im Einsatz ist, da ein Hilfetext stets neu gelesen wird.

Wird im Programm die Taste HP (HELP) betätigt, so sucht *FIX* die Hilfetext-Datei unter dem Namen des aktuellen Elements im Verzeichnis `$HLPPATH` (üblicherweise `hpd`).

Ähnlich wird es auch den Endanwendern ermöglicht, eigene Hilfetexte abzulegen, die sie mit der Taste g6 (CHELP) abrufen können. Diese Hilfetexte werden im Verzeichnis `$CHLPPATH` (üblicherweise `chp`) erwartet.

Das Erstellen eines Hilfetextes ist denkbar einfach. Im laufenden Programm kann durch Drücken der Taste g8 (EDITHELP) in den Editor verzweigt werden. Besteht in `$CHLPPATH` schon ein Hilfetext zu dem Feld bzw. Menüpunkt, so wird dieser in den Editor geladen, sonst wird eine neue Datei angelegt.

Zur Erstellung der Hilfetexte für die Anwendung empfehlen sich zwei Ansätze. Da die Benutzung von g8 bei der Pflege von Hilfetexten besonders einfach und bequem ist,

- können während der Entwicklung des Produktes die Variablen CHLPPATH und HLPPATH gleich (defaultmäßig auf `hpd`) gesetzt und erst bei Auslieferung wieder getrennt werden,

oder

- sollte die Anwendung im Entwicklermodus (Schalter -dev) gestartet werden: die Datei wird dann von *FIX* automatisch in `$HLPPATH` statt in `$CHLPPATH` gesucht.

Natürlich kann man Hilfetext-Dateien auch unabhängig von dem zu entwickelnden Programm editieren. Unumgänglich ist dies, wenn man einen *situationsabhängigen* Text erzeugen will. Hilfetexte können nämlich auch programmgesteuert abgerufen werden (siehe Seite 217).

Die Standard-Hilfetexte, die mit *FIX* ausgeliefert werden (Tastenhilfe etc.), befinden sich im Verzeichnis `$FXDIR/runtime/hpd`.[1]

2 Beschreibungsdatei

2.1 Struktur

Eine Hilfetext-Datei muss eine bestimmte Struktur aufweisen. Dies soll ein Beispiel zeigen:

```
Kundensuchbegriff
Durch Eingabe eines Namens wird bei Nichtvorhandensein des Begriffes automatisch in die Neuaufnahme
eines Stammkunden verzweigt. Der Kundenname muss eindeutig sein.
```

1. siehe aber Umgebungsvariable FXRUNTIMEDIR.

```
~
Dies ist Seite 2 des Hilfetextes.
Falls der Name schon vorhanden ist, wird in den Korrekturmodus übergegangen.
```

Obige Datei beschreibt einen zweiseitigen Hilfetext.

FIX rechnet zur Laufzeit selbständig die Größe des benötigten Windows aus[1]. Will der Benutzer das Window nicht zu groß werden lassen, kann er den Text mit Hilfe von '~'-Zeilen[2] in *Seiten* unterteilen. Im Window ist jeweils nur eine Seite zu sehen.

2.2 Verwendung von Videoattributen und Semigrafik

Einer der Gründe, warum Hilfetexte mittels eines Texteditor erstellt werden und nicht mit Hilfe des **led** liegt darin, auch dem Anwender, der ja nicht über den Layout-Editor verfügt, die Möglichkeit der Textpflege zu geben.

Gleichwohl ist die Verwendung von Videoattributen und Semigrafik recht einfach möglich: über *Umschaltcodes*, die im Text eingestreut werden, wird jeweils eine neue Darstellungsart eingeleitet.

Ein Umschaltcode besteht aus dem Zeichen **'^'**, gefolgt von einem Großbuchstaben:

^A	Normaldarstellung
^B	Invers
^C	Unterstrichen
^D	Halbhell
^E	Blinkend
^N	in Semigrafik-Zeichensatz wechseln
^O	in Standard-Zeichensatz wechseln

Nicht darstellbare Zeichen sind unzulässig.

2.3 Verwendung von Tastenbezeichnungen

FIX ersetzt beim Lesen von Hilfetexten #<tastenbenennung> durch die Tastenbeschriftung, das für die entsprechende Taste definiert bzw. voreingestellt ist (Näheres zu Tastenbeschriftungen finden Sie auf Seite 513). Auf diese Weise können terminalunabhängig Tastenhinweise in Texte integriert werden.

1. Bei Angabe des Schalters -dev erfolgt eine Warnung, wenn sich aus der Struktur der Datei eine unsinnigen Dimension ergibt.
2. '~' muss am Zeilenanfang stehen, dahinter kann eine Überschrift folgen. Das Zeichen '~' darf nicht in oktaler oder hexadezimaler Ersatzdarstellung angegeben werden.

18 Erstellung und Gestaltung von Layouts

Um ein Objekt-Window ansprechend zu gestalten, beispielsweise durch Beschriftung der Felder einer Maske, verfügt **led** neben dem *Beschreibungsmodus* über einen *WYSIWYG-Modus* (What You See Is What You Get), mit dem ein Layout für das Window erstellt werden kann. Auch in diesem Modus ist das Vergrößern und Verschieben von Objekt und Elementen möglich.

Zur reinen Gestaltung eines Layouts ohne Gefahr, die Objektbeschreibung zu modifizieren, dient das Tool **rled**.

1 Layout-Dateien

FIX erlaubt das Hinterlegen von Layouts in unterschiedlicher Form: als *Beschreibung* (Textdatei) und in *binärer Internform*.[1] Der Name der jeweiligen Datei wird im Beschreibungsmodus angegeben. Nach Konvention ist dies für eine Layoutbeschreibung die Datei `<objektname>.pan` im Verzeichnis `pan`, für ein binäres Layout die Datei `<objektname>.mly` im Verzeichnis `mly`.[2]

Da Layoutdateien im Allgemeinen nur mit *FIX*-Tools bearbeitet werden, soll ihr Aufbau nur kurz skizziert werden.

1.1 Layoutbeschreibungen

`.pan`-Dateien bieten den Vorteil, dass sie mit den üblichen UNIX-Tools zur Manipulation von Textdateien bearbeitet werden können. Auch sind sie, da ein Layout gewöhnlich viel Leerraum enthält, trotz der zusätzlichen Verwaltungsinformation meist kleiner als die entsprechende binäre (`.mly`-)Datei. Andererseits muss eine `.pan`-Datei beim Laden von einem Parser analysiert und anschließend interpretiert werden, was geringfügige Laufzeitverluste zur Folge hat.

Neben einem Kopf mit Verwaltungsdaten (Dimension etc.) besteht eine Layoutbeschreibung u.a. aus Operationen

- zum Schreiben von Zeichenfolgen des Standard-Zeichensatzes,
- zum “Zeichnen” von Semigrafik-Zeichenfolgen

und

- zum Setzen von Videoattributen.

In der Datei werden nur 7Bit-ASCII Zeichen verwendet. Nicht darstellbare Zeichen oder echte 8Bit-Zeichen innerhalb von “Argumenten” werden stets in oktaler (\ooo) Ersatzdarstellung angegeben.

1.2 Binäre Layouts

Hier besteht die Datei einfach aus einer Folge von Codes für die im Layout benutzten Videoattribute und Zeichen.

---

1. Gegenwärtig sind beide Repräsentationen gleich mächtig. Zukünftig dürfte jedoch die Beschreibungsform Gestaltungsmöglichkeiten bieten, die über die der binären Form hinausgehen.
2. Beim Laden von Layouts erwarten *FIX*-Programme in Dateien mit der Endung `.pan` eine Layoutbeschreibung.

2 Layout-Erstellung mittels led

Im Beschreibungsmodus des **led**, beschrieben in den Kapiteln *13* und *14*, bewirken die Tasten

g2	zum WYSIWYG-Modus wechseln
g3	Objekt sichern[1]
g9	Hilfetext zum aktuellen Element erstellen/modifizieren

Eine Reihe von Sondertasten haben im WYSIWYG-Modus des **led** eine abweichende Funktion:

PT	Print	Layout “dumpen” (Hardcopy des Windows)
f8	F8	Elemente ein-/ausblenden
g3		Layout sichern
g2		zurück zum Beschreibungsmodus
EN	End	zurück zum Beschreibungsmodus

Im WYSIWYG-Modus werden wiederum zwei Modi unterschieden: *Edit Mode* und *Position Mode*. Bei Betreten des WYSIWYG-Modus befindet man sich zunächst im Edit Mode.

Edit Mode

Der Edit Mode erlaubt es, mit der Schreibmarke über das Layout zu wandern, sein Aussehen zu verändern und die Felder/Menüpunkte zu positionieren.

kh	Home	zur linken oberen Ecke des Layouts
ku	CursorUp	nach oben
kd	CursorDown	nach unten
kl	CursorLeft	nach links
kr	CursorRight	nach rechts
BT	Backtab	8 Spalten nach links
TB	Tab	8 Spalten nach rechts
RT	Return	zum Anfang der nächsten Zeile
CI	Character Insert	Zeichen vor Schreibmarke einfügen
CD	Character Delete	Zeichen unter Schreibmarke löschen
DL	Backspace	Zeichen links neben der Schreibmarke löschen
LI	Line Insert	Zeile einfügen
LD	Line Delete	Zeile löschen
f2	F2	Videoattribut ‘invers’ ein/aus
f3	F3	Videoattribut ‘unterstrichen’ ein/aus
f4	F4	Videoattribut ‘halbhell’ ein/aus
f5	F5	Grafik-Zeichensatz ein/aus
PU	PageUp	in den Elementen rückwärts blättern
PD	PageDown	in den Elementen vorwärts blättern
ST	Start	aktuelles Element an die Position der Schreibmarke bringen
MD	Mode	Position Mode aktivieren

1. nicht während der Elementbearbeitung

Position Mode

Im Position Mode - erkennbar an dem Text “ on ” oberhalb der Taste, die mit “Position Mode” erklärt ist - lässt sich die Größe und Lage des Layouts ändern.

kh	Home	Objekt in die linke obere Bildschirmecke schieben
kl	CursorLeft	Objekt nach links verschieben
kr	CursorRight	Objekt nach rechts verschieben
BT	Backtab	Objekt 8 Spalten nach links verschieben
TB	Tab	Objekt 8 Spalten nach rechts verschieben
ku	CursorUp	Objekt nach oben verschieben
kd	CursorDown	Objekt nach unten verschieben
PU	PageUp	Objekt 8 Zeilen nach oben verschieben
PD	PageDown	Objekt 8 Zeilen nach unten verschieben
CI	Character Insert	Breite vergrößern
CD	Character Delete	Breite vermindern
LI	Line Insert	Höhe vergrößern
LD	Line Delete	Höhe vermindern
f6	F6	Size Mode aktivieren
MD	Mode	Position Mode deaktivieren (Rückkehr in Edit Mode)

Size Mode

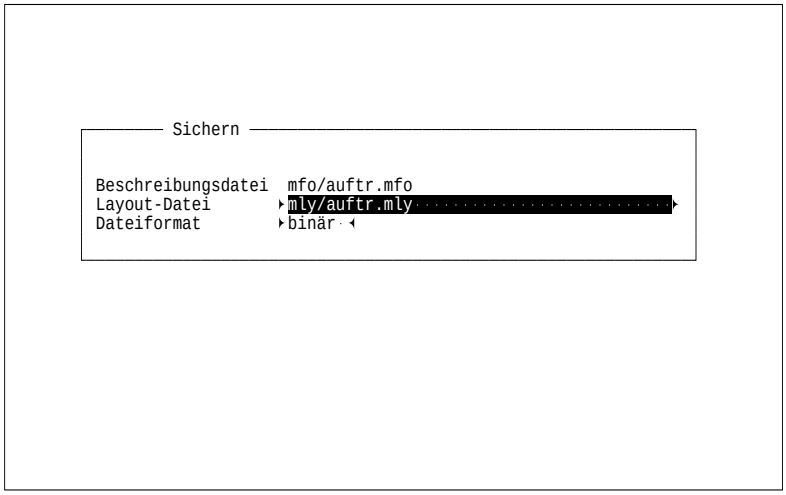
Der *Size Mode* ist ein Untermodus des Position Mode. Er ist erkennbar an dem Text “ on ” oberhalb der Taste, die mit “Size Mode” erklärt ist. Der Size Mode erlaubt nur, die Größe des Objekts mit den Pfeiltasten zu verändern. Dabei wird immer die untere rechte Ecke verschoben.

kr	CursorRight	Breite vergrößern
kl	CursorLeft	Breite vermindern
kd	CursorDown	Höhe vergrößern
ku	CursorUp	Höhe vermindern
f6	F6	Size Mode deaktivieren
MD	Mode	Position Mode deaktivieren (Rückkehr in Edit Mode); deaktiviert automatisch auch den Size Mode

Sichern

Im WYSIWYG-Modus blendet die Taste g3 die folgende Sicherungsmaske auf:

Abb. 26 Sicherungsmaske (WYSIWYG-Modus)



Unter dem Namen der bearbeiteten Datei können der Name für die Layoutdatei eingegeben und ihr Format ausgewählt werden (Default sind die Werte aus dem Beschreibungsmodus bzw. einer vorangegangenen Sicherung). Beim Verlassen der Maske fragt *FIX* nach, ob gesichert werden soll:

Layout sichern? (j/n)

Beim Sichern als Beschreibung werden benachbarte Worte, die durch maximal ein Leerzeichen getrennt sind, zu einer Zeichenfolge zusammengefasst.

Hinweis

Werden im WYSIWYG-Modus Änderungen bzgl. Lage oder Größe von Objekt oder Elementen vorgenommen, muss beim Verlassen des **led** die Objektbeschreibungsdatei ebenfalls abgespeichert werden.

led verwendet zum Abspeichern von Nicht-ASCII-Zeichen in Layoutbeschreibungen immer deren oktale Ersatzdarstellung.

3 Layout-Modifikation mittels rled

rled erlaubt im Gegensatz zum WYSIWYG-Modus des **led** lediglich die Gestaltung des Layouts von Menüs und Masken, jedoch keine Änderung der Objektbeschreibung. Folglich werden ein Verschieben und Redimensionieren des Objekts und die Positionierung seiner Elemente nicht unterstützt.

Beim Start liest **rled** Objekt- und Layoutdatei und der Entwickler findet sich in einer Umgebung wieder ähnlich dem Edit Mode der WYSIWYG-Komponente von **led**. Er kann nun über das Layout wandern:

kh	Home	zur linken oberen Ecke des Layouts
ku	CursorUp	nach oben
kd	CursorDown	nach unten
kl	CursorLeft	nach links
kr	CursorRight	nach rechts
BT	Backtab	8 Spalten nach links
TB	Tab	8 Spalten nach rechts
RT	Return	zum Anfang der nächsten Zeile

Zeichen einfügen oder löschen:

CI	Character Insert	Zeichen vor Schreibmarke einfügen

CD	Character Delete	Zeichen unter Schreibmarke löschen
DL	Backspace	Zeichen links neben der Schreibmarke löschen

Zeilen verschieben:

LI	Line Insert	Zeile einfügen
LD	Line Delete	Zeile löschen

Videoattribute ein- oder ausschalten:

f2	F2	Videoattribut ‘invers’ ein/aus
f3	F3	Videoattribut ‘unterstrichen’ ein/aus
f4	F4	Videoattribut ‘halbhell’ ein/aus

zwischen Standard- und Grafik-Zeichensatz wechseln:

f5	F5	Grafik-Zeichensatz ein/aus

eine Hardcopy des Layouts erstellen:

PT	Print	Layout “dumpen” (Hardcopy des Windows)

das augenblickliche Layout sichern:

g3		Objekt sichern

Darüberhinaus werden angeboten:

g1	REBUILD	Bildschirm neu aufbauen
HP	Help	situationsbezogenen Hilfetext anzeigen
sh	Shell	in eine Shell verzweigen

Verlassen wird **rled** mit der Taste EN. In diesem Fall erscheint die gleiche Sicherungsmaske wie beim Drücken von g3 (vgl. Abb. 26). Nach dem Dialog wird das Programm beendet.

19 Zusätzliche Features des Layout-Editors

1 Konfiguration

led und **rled** sind durch eine Vielzahl von Ressourcen konfigurierbar:

Mfodirs

Werden **led** oder **rled** ohne Dateiargument aufgerufen, erscheint eine Auswahlmaske, in der ein Verzeichnis und eine Datei in diesem Verzeichnis ausgewählt werden können. Beim (bestätigenden) Verlassen des Datei-Feldes wird die ausgewählte Objektbeschreibungsdatei zur Bearbeitung geladen. Nach der Bearbeitung kehrt der Layout-Editor wieder in die Dateiauswahl zurück.

Mittels der Ressource Mfodirs können die Verzeichnisse vorgegeben werden, die in der Auswahlmaske angeboten werden. Als Trennzeichen zwischen den Verzeichnissen ist '|' zu verwenden. Die Angabe ist relativ zu `FXHOME` zu sehen. Wird die Ressource nicht angegeben, dann werden die Verzeichnisse `mfo` und `men` verwendet.

StartWithElementTable

Mittels dieser Ressource kann gesteuert werden, ob zu Beginn die tabellarische Übersicht (TRUE) oder die Detailansicht (FALSE) zur Bearbeitung der Elemente angezeigt wird. Voreingestellt ist TRUE.

Die Ressource ist für **rled** irrelevant.

SwapAfterLastField

Mittels dieser Ressource kann gesteuert werden, ob bei der Bearbeitung der Elemente nach dem Verlassen des letzten Feldes der Detailansicht zur tabellarischen Übersicht zurückgekehrt wird (TRUE) oder nicht (FALSE). Voreingestellt ist TRUE.

Die Ressource ist für **rled** irrelevant.

ActiveEntryAttr

Mittels dieser Ressource kann das Attribut (aus `video.h`) bestimmt werden, das im WYSIWYG-Modus zur Darstellung des aktuellen Menüpunktes verwendet wird. Voreingestellt ist INVERS.

ActiveFieldAttr

Mittels dieser Ressource kann das Attribut (aus `video.h`) bestimmt werden, das im WYSIWYG-Modus zur Darstellung des aktuellen Feldes verwendet wird. Voreingestellt ist INVERS.

IncludeFieldDelimiter

Mittels dieser Ressource kann gesteuert werden, ob die Begrenzer als zum Feld gehörig behandelt werden (TRUE) oder nicht (FALSE). Je nachdem wird beim Positionieren und Anklicken im WYSIWYG-Modus der linke Begrenzer als erste Position angesehen oder nicht. Voreingestellt ist FALSE.

Die Ressource ist für **rled** irrelevant.

InitNewSpace

Mittels dieser Ressource kann gesteuert werden, ob beim Vergrößern der Maske im Position Mode neue Zeilen oder Spalten reinitialisiert werden (TRUE) oder ob der vom vorherigen Entfernen der Zeile (Spalte) vorhandene Inhalt erhalten bleibt (FALSE). Voreingestellt ist TRUE. Wird die Ressource auf FALSE gesetzt, kann die Maske im Position Mode verkleinert und wieder vergrößert werden, ohne dass die aus dem Layout herausgefallenen Zelleninhalte verloren gehen.

Die Ressource ist für **rled** irrelevant.

BeforeLoadCmd
AfterSaveMfoCmd
AfterSavePanCmd
AfterSaveMlyCmd
AfterEditCmd

Siehe Abschnitt 2.

2 Aufruf von Programmen zur Vor- und Nachbehandlung

Durch Einträge in `fix.rc` lassen sich Programme (Shell-Scripte) definieren, die vor und nach der Bearbeitung oder nach einem Speichern aufgerufen werden. Hierfür sind folgende Ressourcen mit dem Pfad des Programms zu belegen:

BeforeLoadCmd

Das Programm wird vor dem Laden von Objektbeschreibungsdatei und Layout(beschreibungs)datei mit dem Namen der zu ladenden Objektbeschreibungsdatei als Argument aufgerufen.

AfterSaveMfoCmd

Das Programm wird nach dem Speichern der Objektbeschreibungsdatei mit deren Namen als Argument aufgerufen.

Die Ressource ist für **rled** irrelevant.

AfterSavePanCmd

Das Programm wird nach dem Speichern einer Layoutbeschreibungsdatei mit deren Namen als Argument aufgerufen.

AfterSaveMlyCmd

Das Programm wird nach dem Speichern einer binären Layoutdatei mit deren Namen als Argument aufgerufen.

AfterEditCmd

Das Programm wird am Ende der Bearbeitung eines Objekts mit dem Namen der Objektbeschreibungsdatei als Argument aufgerufen unabhängig davon, ob zuvor gespeichert wurde.

FIX für Windows

Endet der Name eines Programms mit `.cmd`, `.bat` oder `.exe`, dann wird für den Dateinamen das Zeichen \ als Trenner verwendet. Außerdem wird die Variable FxExecShell auf 1 gesetzt, was zur Folge hat, dass das Programm über die Funktion system() gestartet wird.

3 Benutzung unter *FIX/Win*

Hierzu muss ein Startscript erstellt werden, das die Umgebung der Anwendung aufbaut und das Programm `$FXDIR-SYS/bin/led` mit dem Schalter -service startet. Die Angabe einer Beschreibungsdatei ist nicht notwendig, da die Datei über einen Dialog ausgewählt werden kann.

Die Benutzung des Layout-Editors unter *FIX/Win* bietet einige Vorteile:

- Grafikzeichen werden so dargestellt, wie sie später auch in der Anwendung zu sehen sind.
- Der Layout-Editor kann mit der Maus bedient werden.
- Zum Wechsel zwischen Maskendefinition, Felddefinition und Layoutbearbeitung können Buttons verwendet werden.
- Das Speichern und das Abbrechen der Abarbeitung erfolgt ebenfalls über Buttons. Hier entfällt die Ja/Nein-Abfrage.
- In der Tabellenansicht kann die Elementreihenfolge über Buttons nach Zeilen oder Spalten sortiert werden.
- Bei der Layoutbearbeitung wird der Hintergrund anders gezeichnet, so dass auch bei Objekten ohne Rahmen der Bereich des Objekts optisch erkennbar ist.
- Während der Layoutbearbeitung zeigen Tooltips auf den Elementen die wichtigsten Daten an.
- Viele Funktionen der Layoutbearbeitung sind auch über ein Kontextmenü zugänglich. So kann beispielsweise das aktuelle Element an eine bestimmte Mausposition verschoben werden.
- Bei der Layoutbearbeitung können die aktuelle Position bzw. das aktuelle Element des Objekts durch Anklicken mit der Maus bestimmt werden.
- Im Position Mode kann eine der vier Ecken des Objekts angeklickt werden. Mit einem weiteren Mausklick wird diese Ecke und damit das Objekt verschoben.
- Im Position Mode kann auf den (neuen) SIZE-Modus umgeschaltet werden. Im Size Mode kann die Größe des Objekts auch durch Anklicken mit der Maus bestimmt werden. Dabei wird immer die untere rechte Ecke verändert.

20 Konversion von Layouts

Ab *FIX* 3.1.0 werden zwei neue Programme angeboten, um zwischen Layoutbeschreibungen und binären Layouts zu wandeln.

panconv

panconv dient zum Konvertieren von Layoutbeschreibungen in die binäre Form.

panconv [-t *outformat*] [-ocs *outcharset*] [-o *outfile*] [*panfile*]

panconv erwartet, dass die Datei *panfile* in dem bei der Installation von *FIX* gewählten Zeichensatz (→ FXCHARSET) abgefasst ist. Die Angabe in der Datei selbst wird - analog zum Runtime von *FIX* - nicht berücksichtigt.

Wird *panfile* weggelassen oder ist *panfile* gleich -, so wird von stdin gelesen.

Wird der Schalter -o nicht angegeben oder ist *outfile* gleich -, so wird das Ergebnis (binär) nach stdout geschrieben.

Mittels des Schalters -t kann bestimmt werden, welches binäre Format erzeugt werden soll. Hier ist derzeit nur die Angabe -t old sinnvoll. Fehlt die Angabe, wird -t old angenommen.

Mittels des Schalters -ocs kann ein vom Quellzeichensatz abweichender Zielzeichensatz festgelegt werden. *outcharset* muss einer der Werte ISO-15, ISO-2, IBM437 oder GERMAN7 sein. Ohne die Angabe wird als Zielzeichensatz der für die Quelldatei angenommene Zeichensatz verwendet.

Enthält das zu konvertierende Layout Zeichen, die im Zielzeichensatz nicht existieren, werden die entsprechenden Layoutpositionen in der Form

panconv: *outfile* (*zeile*,*spalte*): char U+*XXXX* not in *outcharset* (using '?')

gemeldet (*XXXX* steht hier für den hexadezimalen Code des Zeichens gemäß dem Unicode-Standard) und in der Ausgabe mit dem Zeichen '?' belegt.

mlyconv

mlyconv dient zum Konvertieren von Layouts in binärer Form in Layoutbeschreibungen.

mlyconv [-t pan] [-7 | -8] [-ocs *outcharset*] [-o *outfile*] [-cs *charset*] -h *height* -w *width mlyfile*

mlyconv verlangt eine Datei als Eingabe, d.h. das Lesen des Quelllayouts aus einer Pipe ist nicht möglich. **mlyconv** erwartet, dass die Datei *mlyfile* in dem bei der Installation von *FIX* gewählten Zeichensatz (→ FXCHARSET) abgefasst ist; siehe aber Schalter -cs.

Wird der Schalter -o nicht angegeben oder ist *outfile* gleich -, so wird das Ergebnis nach stdout geschrieben.

Der Schalter -t ist für künftige Erweiterungen gedacht.

Neben der Breite muss auch die Höhe angegeben werden; **mlyconv** ergänzt für die Berechnung fehlende bzw. ignoriert überzählige Positionen im Layout.

Mittels des Schalters -cs kann ein - von dem bei der Installation von *FIX* gewählten Zeichensatz (→ FXCHARSET) abweichender - Zeichensatz festgelegt werden, den **mlyconv** beim Einlesen von *mlyfile* zu Grunde legen soll. *charset* muss einer der Werte ISO-15, ISO-2, IBM437 oder GERMAN7 sein.

Mittels des Schalters -ocs kann ein vom Quellzeichensatz abweichender Zielzeichensatz festgelegt werden. *outcharset* muss einer der Werte ISO-15, ISO-2, IBM437 oder GERMAN7 sein. Ohne die Angabe wird als Zielzeichensatz der für die Quelldatei angenommene Zeichensatz verwendet.

Achtung: In dieser Version kann als Zielzeichensatz nur der Quellzeichensatz gewählt werden!

Standardmäßig werden – analog zu **mly2pan** - ein Zeichen mit einem Code oberhalb von 127 in oktaler Ersatzdarstellung geschrieben:

Beispiel: (ISO-15) Ä → \304 (4 Bytes), Ö → \326, …

Mittels des Schalters -8 kann eingestellt werden, dass stattdessen Bytes mit gesetztem höchstwertigem Bit ausgegeben werden.

Achtung: In dieser Version ist der Schalter -8 wirkungslos!

In die Layoutbeschreibung wird als Versionsnummer 293 eingetragen. Bei der CharacterSet-Angabe trägt **mlyconv** den Namen des Zielzeichensatzes ein. Diese Angabe dient allerdings nur zur Dokumentation.[1]

Hinweis:

Der Kompatibilität zu **mly2pan** wegen validiert **mlyconv** die gefundenen Zeichencodes i. Allg. nicht, wenn der Zielzeichensatz mit dem Quellzeichensatz übereinstimmt. Eine Ausnahme bilden z.B. Codes größer-gleich 128 beim Quellzeichensatz GERMAN7: hier wird die ungültige Layoutposition in der Ausgabe mit dem Zeichen '?' belegt.

1. Das Runtime von *FIX* 3.1.0 wertet die Zeichensatzangabe in der Datei beim Einlesen nicht aus, sondern erwartet, dass die Datei in dem bei der Installation von *FIX* gewählten Zeichensatz abgefasst ist (→ FXCHARSET).

21 Die Erstellung von Meldungstexten

Ausgangsbasis für die externen Meldungstexte in der Datei `messages` ist eine Textdatei, nach Konvention `fx_texte`. In ihr müssen die einzelnen Meldungen in folgender Form aufgeführt sein:

<unsigned> [, <unsigned>]* : <text>

Dem gleichen Text können mehrere Nummern zugeordnet werden, nicht aber umgekehrt. Die den Texten zugeordneten Nummern müssen größer-gleich 0 und kleiner als 1073741824 sein.

Als Text betrachtet *FIX* alle Zeichen vom ersten Nicht-Leerzeichen hinter : bis zum ersten # (mit dem ein Kommentar eingeleitet wird) oder dem Zeilenende. Erstreckt sich der Text nicht bis zum Zeilenende, enthält er das Zeichen '#' oder beginnt oder endet er mit Leerzeichen, so muss er in Anführungszeichen eingeschlossen werden. Das Zeichen '"' innerhalb des Textes muss dann durch ein vorangestelltes '\' geschützt werden.

Die Verwendung von Videoattributen, Semigrafik und Tastenbezeichnungen erfolgt in der gleichen Weise wie bei Hilfetexten (vgl. Seite 112).

Bevor ein *FIX*-Programm auf die Texte zugreifen kann, muss die Datei `messages` erstellt werden (→ **msgprep**).

Auch die Standardmeldungen der *FIX*-Funktionen sind in dieser Form hinterlegt, allerdings im Verzeichnis `$FXDIR/runtime`.[1] Die *FIX*-Tools benutzen Meldungstexte aus dem Verzeichnis `$FXDIR/de`.

1. siehe aber Umgebungsvariable FXRUNTIMEDIR.

22 Aufbau eines *FIX*-Programms

1 Struktur

Ein *FIX*-Programm muss zumindest enthalten:

Präprozessor-Anweisungen zum Einschließen der benötigten Header-Dateien von *FIX*

Da - zumindest bei Benutzung des generierten `makefile` - der Compiler beim Aufruf instruiert wird, auch das Verzeichnis `$FXDIR/include` nach Header-Dateien zu durchsuchen, genügt ein relativer Pfad bzgl. dieses Verzeichnisses.

Ein Hauptprogramm

Darin müssen die *FIX*-spezifischen Programmschalter interpretiert und vor dem Gebrauch weiterer Funktionen aus der *FIX*-Library die internen Strukturen von *FIX* initialisiert werden. Dies geschieht durch Aufruf der Funktionen arginspect() und fxinit(). Nach dem Aufruf von fxinit() sollte das Programm keine I/O-Routinen aus der C-Library mehr zur Ausgabe auf den Bildschirm benutzen, da sie die Bildschirmbehandlung von *FIX* stören (vgl. “Bildschirmverwaltung” auf Seite 273). Durch Aufruf von fx_database_connect() wird eine Verbindung zur Datenbank aufgebaut, durch fx_database_disconnect() wieder abgebaut. Vor dem Beenden des Programms muss fxquit() aufgerufen werden.

```
main(int argc, char *argv[])
{
    arginspect(&argc, argv);
    Interpretation verbliebener anwendungsspezifischer Schalter
    …
    fxinit(0, (char **)0);                          ┐
    …                                               │
    (void)fx_database_connect((char *)0);           │
    …                                               │ FIX-
    eigentliche Programmlogik                       │ Bildschirm-
    …                                               │ verwaltung
    (void)fx_database_disconnect();                 │
    …                                               │
    fxquit();                                       ┘
    …
    exit(0);
}
```

2 Header-Dateien

Generierte Programme schließen ein:

`fix/fix.h`

`fix/release.h`	Definition des Makros FIX_VERSION (Release-Stand des *FIX*-Entwicklungssystems)
`fix/basics.h`	Basis-Makros
`fix/obj.h`	Definition der Datenstrukturen für Objekte; schließt u.a. ein:

`fix/props.h`	
`fix/box.h`	
`fix/fixdecimal.h`	
`fix/fixdate.h`	
`fix/fixdatetime.h`	
`fix/fxtypes.h`	
`fix/dbio.h`	
`fix/sql.h`	
`fix/mrel.h`	
`fix/selo.h`	
`fix/macros.h`	
`fix/upgrade.h`	
`fix/fxctypes.h`	Variablen-bezogene Definitionen
`fix/accept.h`	Feldtypen und feldbezogene Vereinbarungen
`fix/keys.h`	Definition der Events
`fix/error.h`	Definition von Fehlercodes
`fix/fxmodul.h`	Datenstruktur modul_struct
`fix/gen.h`	Makros für Zugriffsarten (fx_io())
`fix/fx_io.h`	Datenbankbezogene Definitionen
`video.h`	Definition von Makros für Videoattribute
`fix/libfix.h`	Deklarationen/Prototypen der *FIX*-Funktionen; schließt ein:
	`proto/*.h`
`fix/fxdbval.h` [1]	Deklarationen/Prototypen von Funktionen zur Wandlung von Werten

Damit sollten alle wesentlichen Vereinbarungen abgedeckt sein.

Die Header-Dateien in `$FXDIR/include` und `$FXDIR/include/proto` sind nicht für den Entwickler bestimmt.

3 Interpretation der *FIX*-spezifischen Schalter

Durch den Aufruf der Funktion

```
void    arginspect(int *argc_p, char **argv)
```

mit der Adresse von *argc* und der Argumentliste *argv* werden die in der Kommandozeile angegebenen *FIX*-spezifischen Programmschalter interpretiert und aus der Liste entfernt, wobei der Wert von *argc* entsprechend vermindert wird. Außerdem wird der Programmname in einem statischen Bereich hinterlegt.

Erkannt werden folgende Schalter:

-version	Version ausgegeben und Programm beenden
-dev	Entwicklermodus
-DM	kein automatischer Dezimalpunkt bei Eingabe von Geldbeträgen
-sh	keine Shell-Berechtigung

1. nur bei Verwendung von Indikatorvariablen (vgl. Seite 466).

-ftest [<number>]	setzt die globale Variable “int F_test” auf den angegebenen Wert (vgl. Seite 440)
-7	bei 8Bit-fähigen Systemen: Verhalten wie auf nicht 8Bit-fähigen Systemen
-8	bei nicht 8Bit-fähigen Systemen: Verhalten wie auf 8Bit-fähigen Systemen
-service	Programm wird von einem Frontend bedient (*FIX/Win*, *FIX/Web*)
-test [<number>]	interner Schalter zum Debugging von *FIX*
-debug [<terminal device>]	Umlenken von stdin und stdout auf ein anderes Terminal
-trace <filename>	Eingabe protokollieren
-init [<filename> | !<cmd>]	erst Eingabe von Datei oder Pipe lesen

4 Initialisierung von *FIX*

Bevor ein Anwendungsprogramm weitere Funktionen aus der *FIX*-Library benutzen darf, muss zunächst die Funktion

```
void    fxinit(int argc, char **argv) [1]
```

aufgerufen werden.

fxinit() erfüllt eine Reihe von Aufgaben:

- Ermitteln der Umgebung, in der das Programm abläuft; dazu gehören:
  - Verzeichnisse, in denen *FIX* installiert ist (FXDIR, ggf. FXRUNTIMEDIR)
  - der Zeichensatz, den die *FIX*-Installation benutzt (FXCHARSET)
  - Verzeichnisse, die die Anwendung enthalten (FXHOME, HLPPATH, CHLPPATH)
  - bevorzugte Shell (SHELL)
- Laden der Ressourcen
- Bestimmung der Meldungsdateien von *FIX* und Anwendung
- Lizenzprüfung (FXINSTFILE)
- Ermitteln von Bildschirm und Tastatur (FXTERM, FXKEYBOARD)
- Einlesen der Bildschirmbeschreibung (FXTERMPATH)
- Einlesen des Zeichensatzes (FXCHARSETPATH) und der Umsetzungstabellen
- Einlesen der Tastaturbeschreibung (FXTERMPATH)
- Aufbau und Initialisierung interner Datenstrukturen
- Umsetzen der Signalbehandlung
- Umsetzen der Terminal-Parameter (-ICRNL, IXON, VMIN=1, VTIME=0, -ICANON, -ECHO, -ISTRIP)[2]

1. Bis Release 2.7 umfasste fxinit() die Funktionalität von arginspect(). Aus Kompatibilitätsgründen wurden die Parameter beibehalten, werden aber nicht mehr ausgewertet.

2. Zur Umsetzung der Terminalparameter wird seit *FIX* 2.9.2 nach Möglichkeit die termios- anstelle der termio-Schnittstelle verwendet.

Erkennt fxinit() Fehler, die ein korrektes Ablaufen des Programms unmöglich erscheinen lassen, bricht es das Programm ab. Die Funktion darf in einem Programm nur ein einziges Mal aufgerufen werden.

5 Ressourcen

Beim Aufruf der Funktion fxinit() durchsucht *FIX* die Dateien

1. `$FXHOME/`*`programname`*`.rc` (programmspezifische Ressourcen)

oder, sofern nicht vorhanden,

2. `$FXHOME/fix.rc` (anwendungsspezifische Ressourcen)

oder, sofern nicht vorhanden,

3. `$FXDIR/fix.rc` (installationsspezifische Ressourcen)

sowie, sofern die Umgebungsvariable FXRCFILE gesetzt ist, zusätzlich zu einer der obigen die Datei

4. `$FXRCFILE`

nach Ressourcen für das Programm.[1]

Die Ressourcedatei besteht aus

- Einschluss-Anweisungen:
  # **include** <dateiname>
- Ressourcedefinitionen:
  <programname>.<resource> **:** <value> (höhere Priorität, da spezifischer)
  bzw.
  <resource> **:** <value> (niedrigere Priorität, da allgemeiner)
- Leerzeilen

Das Zeichen ! leitet einen Kommentar ein, der sich bis zum Zeilenende erstreckt. Die Abfolge der Zeilen ist signifikant; der Einschluss einer Datei wird behandelt, als ob ihre Zeilen an dieser Stelle stünden.

<value> beginnt mit dem ersten Nicht-Leer- oder -Tabulatorzeichen, das auf den Doppelpunkt folgt, und erstreckt sich bis zu dem dem Zeilenende oder Kommentaranfang evtl. vorausgehenden Leerraum, z.B.

```
fgmask.SQLPrefix :  EXEC  SQL     ! Symbol(folge), die ESQL/C-Anweisungen einleitet
```

Hier umfasst <value> die unterstrichenen Zeichen. Sollen Leerzeichen am Anfang oder Ende eines Wertes signifikant sein, so müssen sie - wie auch das Zeichen ! - in oktaler oder hexadezimaler Ersatzdarstellung angegeben werden.

Eine Definition für eine Ressource gilt nur, wenn keine gleichnamige Umgebungsvariable definiert ist, z.B verdeckt

```
SQLPrefix="EXEC SQL" ; export SQLPrefix
```

die Definition aus obigem Beispiel.

Welche Ressourcen von jedem bzw. einem bestimmten Programm ausgewertet werden, beschreibt "Ressourcen" auf Seite 447. Es ist geplant, zukünftig auch dem *FIX*-Entwickler einen Zugriff auf Ressourcen zu ermöglichen.

Hinweis:

Wenn Ressourcewerte von der Plattform (Betriebssystem, Datenbanksystem etc.) abhängig sind, sollten sie in einer analog benannten Datei im Dateibaum $FXHOMESYS (bzw. $FXDIRSYS) abgelegt werden und diese Datei mittels

```
# include $(FXHOMESYS)/relpath
```

in der obigen Datei eingeschlossen werden. *FIX* expandiert $(…) zu dem entsprechenden Wert.

1. gegenwärtig nur, wenn der Programmname gesetzt wurde, was beim Aufruf von arginspect() geschieht.

6 Ansprechen von C-Funktionen in Objektbeschreibungen

Bevor ein *Anwendungs*programm ein Objekt lädt, dessen Beschreibung den Namen für eine C-Funktion beinhaltet, muss diesem Namen im Programm (die Adresse) eine(r) Funktion zugeordnet worden sein. Die *FIX*-Library enthält zu diesem Zweck die Funktion

```
BOOLEAN    addfct(char *name, int (*fct)())
```

die dem Namen *name* den Funktionszeiger *fct* zuordnet. Mit der Funktion

```
int  (*getfct(char *name))()
```

kann auf die *name* zugeordnete Funktion zugegriffen werden.

Beispiel:

```
int f();                    /* Deklaration einer an anderer Stelle definierten Funktion */

addfct("F", f);             /* Zuordnung der Funktion f() an den Namen “F” */

(*getfct("F"))(…);          /* Aufruf der Funktion mit dem Namen “F” */
```

Sollte die Funktion keinen int-Wert zurückgeben, muss dem getfct()-Aufruf ggf. ein geeigneter *cast* vorangestellt werden.

7 Terminierung von *FIX*

Nachdem ein Aufruf von fxinit() erfolgt ist, muss vor Verlassen des Programms unbedingt die Funktion

```
void    fxquit(void)
```

aufgerufen werden. Dies geschieht automatisch, wenn Sie das Programm mit der von *FIX* angebotenen Funktionen

```
void    fastexit(int status, char *txt)
```

oder dem Makro

```
fxexit(status)
```

beenden.

fxquit() restauriert im Wesentlichen die vor dem Aufruf von fxinit() gültigen Terminal-Parameter. Ein Rücksetzen der Signalbehandlung oder Freigeben des von *FIX*-Datenstrukturen belegten Speichers erfolgt dagegen nicht.

Hinweis:

Sollte sich Ihr Terminal nach dem gewaltsamen Abbruch eines *FIX*-Programms einmal ungewöhnlich verhalten, tippen Sie ^J (Control J) und versuchen Sie danach eines der Kommandos

```
reset^J
```

```
stty sane^J
```

oder

```
stty icrnl icanon echo^J
```

8 Signalbehandlung

Solange *FIX* den Bildschirm verwaltet, werden die Signale 1 (SIGHUP) bis 15 (SIGTERM) - SIGKILL ausgenommen - von *FIX* abgefangen. Bis auf die nachfolgend beschriebenen Ausnahmen erfolgt beim Eintreffen des Signals ein Programmabbruch mittels fastexit(), wobei das verursachende Signal dem Anwender mitgeteilt wird.

Die Ausnahme bildet - aus historischen Gründen - das Signal SIGINT. Sein Eintreffen entspricht der Taste sh, soweit für diese keine spezielle Tastenfolge definiert ist, und wird anderenfalls ignoriert.

Bei Systemen, die das Signal SIGTSTP unterstützen, restauriert *FIX* vor dem Anhalten die beim Programmstart gültigen Terminal-Parameter. Beim Fortsetzen des Programms werden wieder die für *FIX* typischen Einstellungen vorgenommen.

Sofern auf der Plattform verfügbar, benutzt *FIX* anstelle von signal() die Routine sigaction() (vgl. XPG3), um die eigene Signalbehandlung zu installieren. Dabei wird möglichst versucht, unterbrochene Systemaufrufe wieder aufzusetzen (SA_RESTART).

Eine Ausnahme bildet das Signal SIGALRM. Hier wird bei UNIX-Derivaten, die Systemaufrufe defaultmäßig wieder aufsetzen, versucht, dieses Verhalten für SIGALRM zu unterbinden (SA_INTERRUPT).

Da ein Prozess nicht zugleich beide Methoden - signal() und sigaction() - zur Installation von Signalbehandlungsroutinen benutzen sollte, müssen Anwendungen mit eigener Signalbehandlung denselben Mechanismus wie *FIX* nutzen.

23 Datenbankanschluss

Mit dem IBM Informix-Datenbankserver kommuniziert *FIX* über die ESQL/C-Schnittstelle. *FIX*-Versionen für IBM Informix SE bieten optional C-ISAM als Schnittstelle an. Für den Zugriff auf Datenbanken mit Transaktionssicherung stellt *FIX* eine an ANSI angelehnte Transaktionslogik bereit.

1 Verbindungsaufbau zur Datenbank

Die Funktion

```
long    fx_database_connect(char *dbname)
```

baut eine Verbindung zur Datenbank *dbname* auf. Ist *dbname* == (char *)0, wird der Name der Datenbank der Umgebungsvariablen DBNAME entnommen.

fx_database_connect() weist der globalen Variablen

```
char    *fxdbname;
```

eine Kopie des Datenbanknamens zu und setzt bei erfolgreichem Verbindungsaufbau die globale Variable

```
BOOLEAN     dbconnected = FALSE;
```

auf TRUE. Anhand des Wertes von dbconnected entscheidet *FIX*, ob zu einem gegebenen Zeitpunkt eine Datenbankverbindung besteht. Bei jedem erfolgreichen Verbindungsaufbau zu einer Datenbank wird ein interner Zähler um Eins erhöht, der mittels der Funktion

```
long    fx_connect_cnt(void)
```

abgefragt werden kann.

Beim Aufbau der Datenbankverbindung wird erkannt, ob die Datenbank über eine Transaktionssicherung verfügt. Wenn dies der Fall ist, wird die globale Variable

```
BOOLEAN     B_transaction = FALSE;
```

auf TRUE gesetzt und automatisch eine Transaktion begonnen. Bei Nicht-ANSI-Datenbanken erfolgt hierzu ein “begin work”.

Hinweis bei Benutzung von C-ISAM:

- Wie IBM Informix SE lässt auch *FIX* als Datenbanknamen einen absoluten Pfad (ohne Endung `.dbs`) zu und sucht bei einem relativen Namen zunächst im aktuellen Verzeichnis, dann in den in DBPATH enthaltenen Verzeichnissen nach der Datenbank.
- Beim Zugriff über C-ISAM werden die logischen Zugriffsrechte der Datenbank ignoriert. Für den Anwender müssen der Datenbank-Pfad durchsuchbar und die .idx- und .dat-Dateien einschließlich der Systemtabellen lesbar sein.

Zum Abbau der Datenbankverbindung dient die Funktion

```
long    fx_database_disconnect(void)
```

fx_database_disconnect() setzt die Variable dbconnected wieder auf FALSE zurück. Bei einer Datenbank mit Transaktionssicherung (B_transaction == TRUE) wird vor dem Verbindungsabbau automatisch die laufende Transaktion mittels “rollback work” zurückgesetzt und B_transaction erhält wieder den Wert FALSE.

2 Virtuelle Transaktionen

FIX stellt dem Entwickler Funktionen zur Verfügung, mit denen *virtuelle Transaktionen* gebildet werden können. Diese können ineinander geschachtelt werden, wobei dann eine Rollback-Anforderung bis zur äußersten Ebene propagiert wird.

```
int  fx_begin_transaction(void)
int  fx_commit_transaction(void)
int  fx_rollback_transaction(void)
```

Die augenblickliche Transaktionsstufe verwaltet *FIX* in der globalen Variablen

```
int  IN_transaction = -1;
```

Beim Aufbau einer Verbindung zu einer Datenbank mit Transaktionssicherung erhält IN_transaction den Wert 0.

Besitzt die Datenbank keine Transaktionssicherung, sind die Funktionen ohne Wirkung und IN_transaction hat einen Wert kleiner 0.

Beispiel 1:

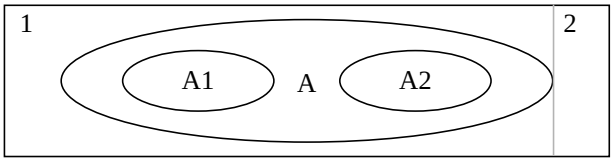
keine Schachtelung

fx_database_connect(…)	Es erfolgt ein “begin work” ; IN_transaction erhält den Wert 0.
…	
fx_commit_transaction()	Da IN_transaction == 0, wird die laufende Transaktion durch “commit work” beendet und dann eine neue Transaktion begonnen; IN_transaction behält seinen Wert.
…	
fx_database_disconnect()	Es erfolgt ein “rollback work”; der Wert von IN_transaction wird < 0.

Beispiel 2:

Die Anwendung benutzt ein Modul A, das seinerseits zwei Teilmodule A1 und A2 beinhaltet. Jedes der drei Module ist von virtuellen Transaktionsklammern eingeschlossen.

Abb. 27



Beim Verlassen von A wird die beim Öffnen der Datenbank begonnene Transaktion 1 genau dann zurückgesetzt, wenn eines der drei Module mit fx_rollback_transaction() beendet wird. Die sich daran anschließende Transaktion setzt *FIX* beim Schließen der Datenbank von sich aus zurück.

fx_database_connect(…)	Es erfolgt ein “begin work”; IN_transaction erhält den Wert 0.
…	
fx_begin_transaction()	IN_transaction wird auf 1 erhöht.
…	
fx_begin_transaction()	IN_transaction wird auf 2 erhöht.
…	
fx_commit_transaction()	IN_transaction wird auf 1 erniedrigt. Da IN_transaction > 0, wird “commit work” unterdrückt.
…	
fx_begin_transaction()	IN_transaction wird auf 2 erhöht.
…	
fx_rollback_transaction()	IN_transaction wird auf 1 erniedrigt. Da IN_transaction > 0, wird “rollback work” unterdrückt, *FIX* merkt sich aber, dass ein Rollback gefordert wurde.
…	
fx_commit_transaction()	IN_transaction wird auf 0 erniedrigt. Da IN_transaction == 0, wird die laufende Transaktion beendet, wegen des vorausgegangenen fx_rollback_-transaction() allerdings nicht durch “commit work”, sondern durch “rollback work”. Dann wird eine neue Transaktion begonnen.
…	
fx_database_disconnect()	Es erfolgt ein “rollback work” ; der Wert von IN_transaction wird < 0.

3 SQL-Fehler

Nach der Ausführung von SQL-Anweisungen ruft *FIX*, von wenigen Ausnahmen abgesehen, stets die Funktion

```
void    fx_sql_test(char *txt)
```

auf. Ist SQLCODE < 0, meldet sie den Fehler durch Aufruf der Funktion

```
void    sql_error(long sqlcode, char *txt)
```

Die Funktion sql_error() bildet ein eigenes Modul der *FIX*-Library, dessen Quellcode mitgeliefert wird. Der Entwickler kann so eine selbsterstellte Routine dieses Namens einbinden, um seine eigene Fehlerbehandlung zu realisieren.

Standardmäßig protokolliert sql_error() den Fehler in eine Log-Datei (vgl. Seite 445), weist den Anwender auf den Fehler hin und fragt ihn, ob er das Programm beenden möchte. Wird die Frage bejaht, erfolgt ein Programmabbruch mittels fastexit().

24 Datenstrukturen

Um die in Dateien beschriebenen Objekte im Programm nutzen zu können, müssen sie geladen, d.h. gelesen und in äquivalente Datenstrukturen umgesetzt werden. Diese Datenstrukturen sind in der Header-Datei `fix/obj.h`, die jedes *FIX*-Programm einschließt, definiert. Der Entwickler kann sich in der Regel auf den Umgang mit Objekt-*Zeigern* beschränken und wird sich nur in Ausnahmefällen mit dem internen Aufbau der Datenstrukturen auseinander setzen müssen.

Das Laden von Objekten erfolgt entweder *explizit* durch Aufruf einer entsprechenden Funktion im Anwendungsprogramm oder *implizit* aufgrund des internen Vorgehens von *FIX*. Zum Laden von Objekten stehen folgende Funktionen zur Verfügung:

```
menue *loadmenue(char *mname)
```

lädt ein Menü (einschl. Layout).

```
mask  *loadmask(char *mname)
```

lädt eine Maske (einschl. Layout) und die deren Feldern zugeordneten Selos und Choices sowie bei Tabellenmasken ggf. die verschiedenen Zeilentypen; verfährt anschließend ebenso mit allen eingebetteten Masken.

```
selo    *loadselo(char *mname)
```

lädt ein Selo.

```
choice *loadchoice(char *mname; MemRelType mrp)
```

lädt eine Choice (einschl. Layout); das zweite Argument erlaubt dem Programmierer, eine Datenmenge vorzugeben.

Argument ist jeweils der Name der Beschreibungsdatei. Relativen Dateipfaden, d.h. solchen, die nicht mit '/' beginnen, stellt *FIX* das Anwendungsverzeichnis voran. Bzgl. des Lesens von Beschreibungsdateien beachten Sie die Anmerkungen zur Syntax auf Seite 86.

Erkennt *FIX* beim Laden Fehler, wird der Ladevorgang mit einer Fehlermeldung beendet und i. Allg. ein Nullzeiger zurückgegeben (bei einigen Funktionen wird stattdessen ein Programmabbruch mittels fastexit() eingeleitet).

FIX führt über die erfolgreich geladenen Objekte Buch: bei der Initialisierung von *FIX* wird in der globalen Variable 'obj **objtable' die Adresse eines Array von zunächst 'NOBJ' Elementen eingetragen (vgl. hierzu Seite 441), in dem *FIX* Zeiger auf die geladenen Menüs, Masken und Choices hinterlegt. *FIX* benutzt 'objtable' u.a., um Feld-Referenzen der Form #<maskenname>.<feldname> aufzulösen.

Hinweis:

Neben den nachfolgend dokumentierten, dem Entwickler (lesend) zugänglichen Komponenten enthalten die Datenstrukturen von Objekten eine Vielzahl weiterer Komponenten, die lediglich intern verwendet werden oder nur über Funktionen der *FIX*-Library abfragbar oder modifizierbar sind. Derartige Komponenten können im Rahmen der Weiterentwicklung von *FIX* entfallen oder hinzukommen, was sich auf die Speichergröße eines Objekts auswirken kann. Der Einsatz eines neuen *FIX*-Release erfordert deshalb in der Regel ein Neu-Kompilieren der Programme.

Anwendungsprogramme dürfen keine Annahmen bezüglich der Anordnung von Objektkomponenten machen mit einer Ausnahme: alle Datenstrukturen beginnen mit einer int-Komponente, aus der die Art des Objekts hervorgeht.

Die Modifikation von Strukturkomponenten im Anwendungsprogramm sollte nur mittels der von *FIX* zur Verfügung gestellten Funktionen und Makros erfolgen.

1 Objekten anwendungsspezifische Information zuordnen

Menüs und Menüpunkte, Masken und Felder, Selos und Choices kann der Entwickler mit eigener, anwendungsspezifischer Information versehen. Diese Information, deren Art und Umfang ins Belieben des Entwicklers gestellt ist, muss in eine Struktur der Form

```
typedef struct {
    long    longval1;
    long    longval2;
    long    longval3;
    long    longval4;
    long    longval5;
    long    longval6;
    long    longval7;
    long    longval8;
    char    *ptrval1;
    char    *ptrval2;
} PrivatePropertyType;
```

verpackt werden. Die Komponenten *ptrval1* und *ptrval2* sollten hierbei als Zeiger auf einen unbestimmten Typ betrachtet werden, z.B. auf eine von der Anwendung verwaltete, beliebig komplexe Datenstruktur.

Initialwerte für die longval-Komponenten können in der Beschreibungsdatei hinterlegt werden (**set**). Wenn keine externe Vorbelegung möglich oder vorgenommen ist, initialisiert *FIX* die betreffende Komponente mit 0 bzw. (char *)0. Unterstützt werden nur Werte zwischen 0 und 2147483647.

Im Anwendungsprogramm kann die Information sowohl abgefragt als auch neu gesetzt werden. Dazu werden folgende Funktionen angeboten:

zum Abfragen:

Objekt:	PrivatePropertyType o_get_props(obj *objp)
Menüpunkt:	PrivatePropertyType en_get_props(menue_entry *f)
Feld:	PrivatePropertyType f_get_props(field *f)

zum Setzen:

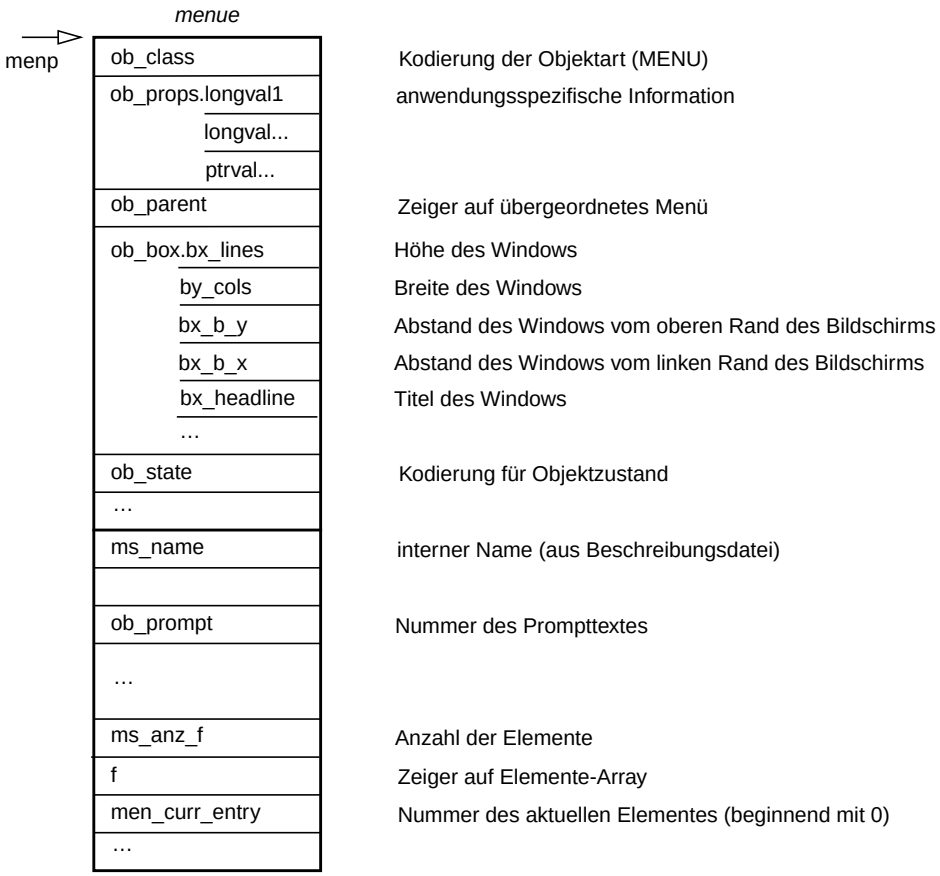
Objekt:	void	o_set_props(obj *objp, PrivatePropertyType value)
Menüpunkt:	void	en_set_props(menue_entry *f, PrivatePropertyType value)
Feld:	void	f_set_props(field *f, PrivatePropertyType value)

Abfragefunktionen liefern als Ergebnis die augenblicklich zugeordnete Information. Zuweisungsfunktionen erwarten die zu setzende Information als zweites Argument und kopieren sie in die Datenstruktur.

2 Menüs

Beim erfolgreichen Laden eines Menüs erhält der Entwickler einen Zeiger auf eine dynamisch allokierte Datenstruktur folgender Form:

Abb. 28 Menü-Datenstruktur



Anmerkungen

In der Komponente *ob_props* wird die anwendungsspezifische Information zum Menü hinterlegt (→ o_set_props(), o_get_props()). Mehr hierzu finden Sie auf Seite 140.

Bei durch eine ‘perform <datei>’-Aktion implizit geladenen Menüs enthält die *ob_parent*-Komponente einen Zeiger auf das übergeordnete Menü.

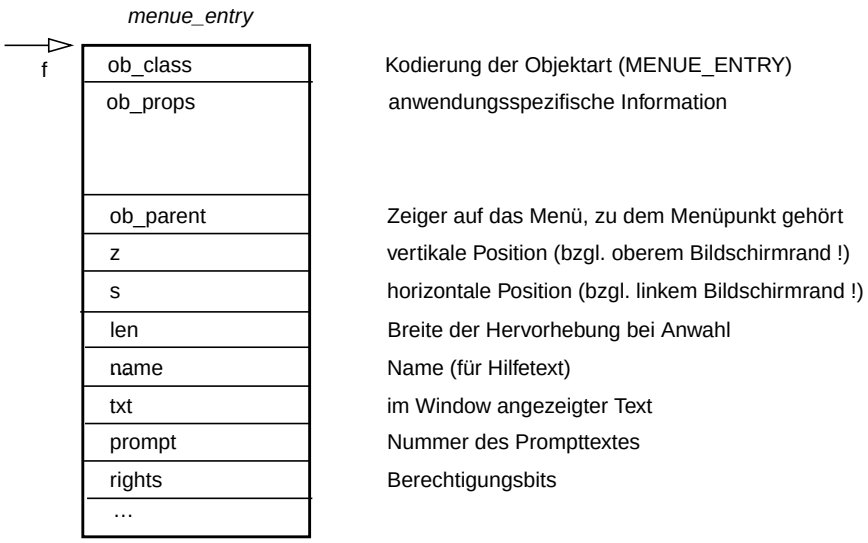
Sofern der Window-Titel nicht die Form #<unsigned> hat, hat *FIX* analog zum Laden von Meldungstexten darin vorkommende Umschaltcodes wie “^A” durch die entsprechenden Steuerzeichen ersetzt.

ob_state enthält, in kodierter Form, Eigenschaften und augenblicklichen Zustand des Menüs, unter anderem:

(ob_state & (STEADY|ABOVE)) == (STEADY|ABOVE)	Objekt muss stets oberhalb des vorangehenden Objekts liegen
(ob_state &STEADY) == STEADY	Objekt soll nach der Bearbeitung am Bildschirm bleiben
(ob_state & ACTIVE) == ACTIVE	Objekt ist in Bearbeitung

f zeigt auf ein Array von *ms_anz_f* Elementen vom Typ *menue_entry*, die die Menüpunkte repräsentieren. Die Anordnung der Elemente im Array entspricht der Reihenfolge in der Beschreibungsdatei.

Abb. 29 Menüpunkt-Datenstruktur



Anmerkungen

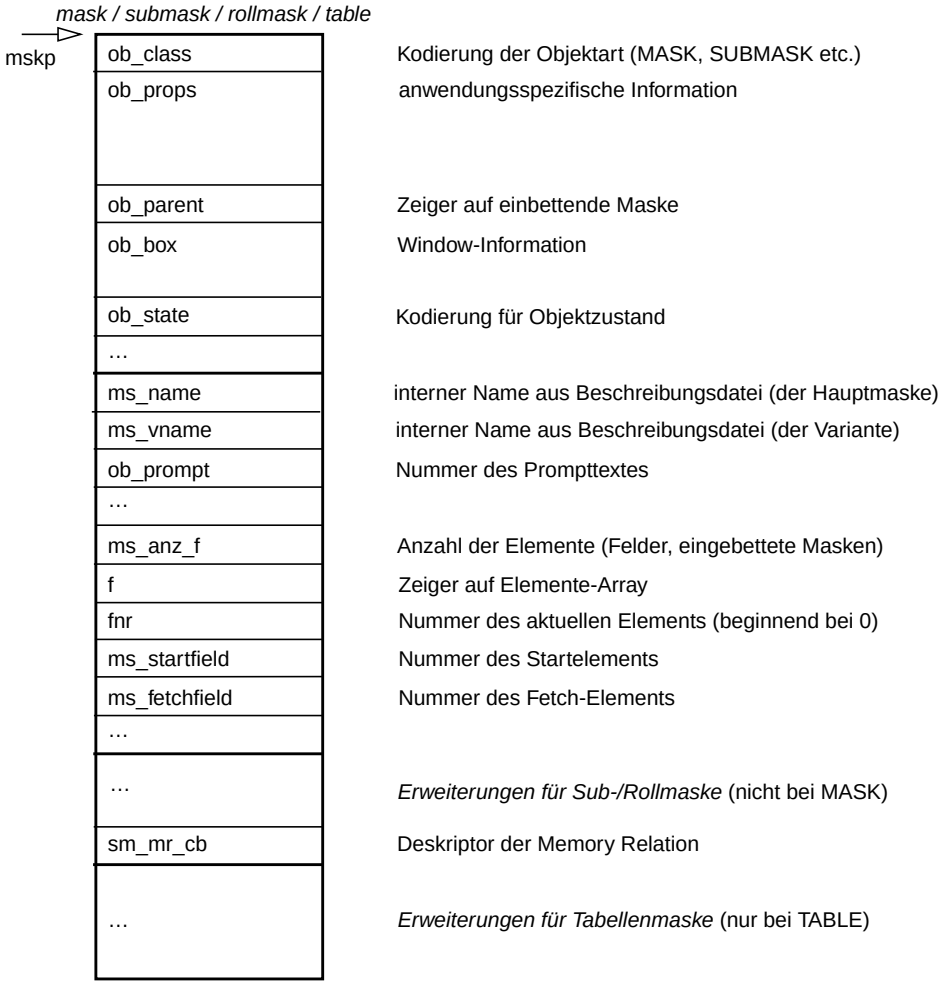
Ist keine explizite Hervorhebung angegeben oder unterschreitet sie die Darstellungsbreite des Textes *txt* (→ putstrwidth()), setzt *FIX len* auf Letztere.

Zur Bedeutung der *rights*-Komponente vgl. Seite 165f.

3 Masken

Beim erfolgreichen Laden einer Maske erhält der Entwickler einen Zeiger auf eine dynamisch allokierte Datenstruktur folgender Form:

Abb. 30 Masken- Datenstruktur



Anmerkungen

Aufbau und Bedeutung der Komponenten *ob_props* und *ob_box* sind auf Seite 141 beschrieben.

Die *ob_parent*-Komponente einer eingebetteten (also implizit geladenen) Maske enthält einen Zeiger auf die einbettende Maske.

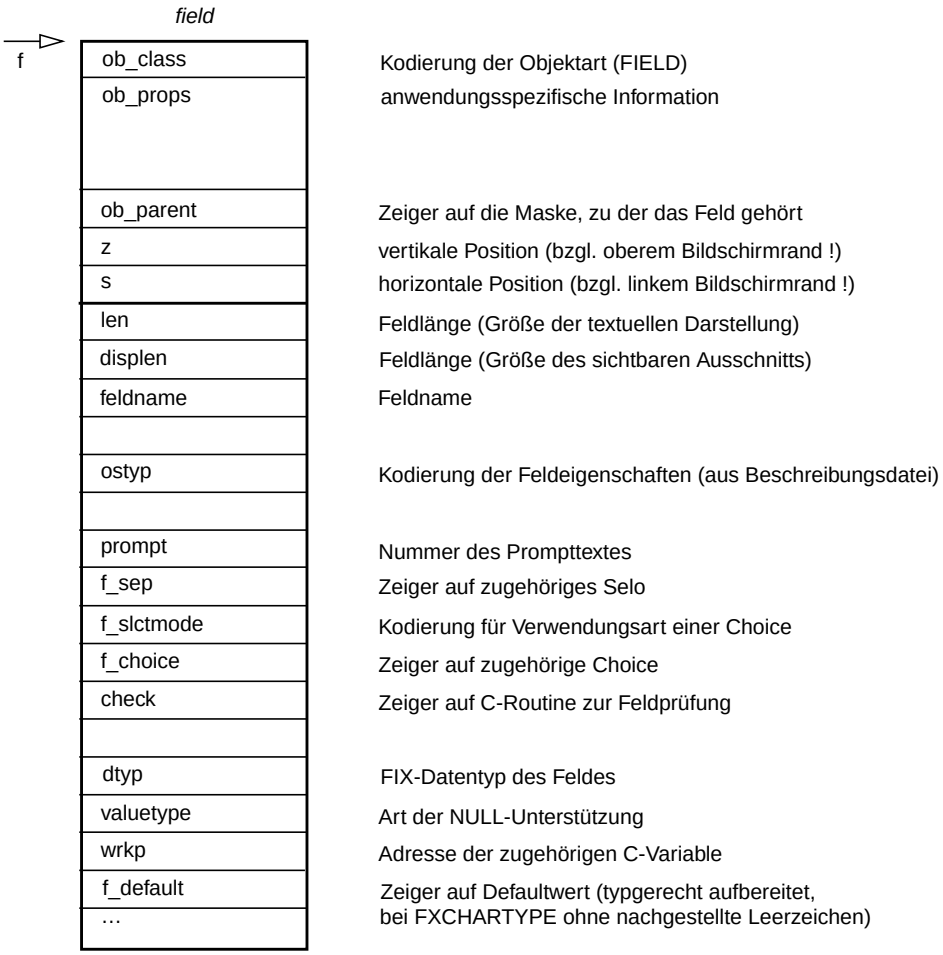
ob_state enthält, in kodierter Form, Eigenschaften und augenblicklichen Zustand der Maske, unter anderem:

(ob_state & (STEADY|ABOVE)) == (STEADY|ABOVE)	Objekt muss stets oberhalb des vorangehenden Objekts liegen (**pasting**)
(ob_state & STEADY) == STEADY	Objekt soll nach der Bearbeitung am Bildschirm bleiben (**steady**)
(ob_state & ACTIVE) == ACTIVE	Objekt ist in Bearbeitung
(ob_state & RESTART) == RESTART	bei Betreten soll Hauptmaske benutzt werden
(ob_state & NOENT) == NOENT	nur bei eingebetteten Masken: Objekt kann nicht betreten werden (wie Feldeigenschaft NOENT)

f zeigt auf ein Array von *ms_anz_f* Elementen. Jedes Element ist entweder eine Feld- oder eine Masken-Datenstruktur, unterscheidbar am Wert der - bei allen Strukturen deckungsgleichen - Komponente *ob_class*, die eine der Kodierungen FIELD, MASK, SUBMASK, ROLLMASK, TABLE enthält. Die Anordnung der Elemente im Array entspricht der Reihenfolge in der Beschreibungsdatei.

Den für den Entwickler relevanten Teil der Feld-Datenstruktur gibt die nachfolgende Abbildung wieder:

Abb. 31 Feld-Datenstruktur



Anmerkungen

In der Komponente *valuetype* ist kodiert, ob das Feld NULL unterstützt:

(valuetype & NOT_NULL) == NOT_NULL NULL ist nicht zulässig

wrkp zeigt - außer bei einem Feld-Link - bis zu einem erfolgreichen Aufruf von mfn_varbind() auf einen maskeneigenen Speicherbereich geeigneter Größe (→ fxdtypsize()).

Bei einer Tabellenmaske mit Zeilentypen enthalten die Komponenten *z*, *s*, *len*, *displen*, *ostyp*, *prompt*, *f_slctmode* der Feld-Datenstruktur jeweils die Werte des für den aktuellen Satz zutreffenden Zeilentyps, bzw. bei einer leeren Tabelle die Werte der Grundmaske.

Ist in der Beschreibungsdatei zu dem Feld ein Selo oder eine Choice angegeben, wurde auch hierfür eine entsprechende Datenstruktur aufgebaut und ihre Adresse in *f_sep* bzw. *f_choice* eingetragen. *f_sep* und *f_choice* dürfen nicht beide ungleich dem Nullzeiger sein. Ist eine Choice vorhanden, beschreibt *f_slctmode* ihre Verwendung:

ch_obligat	*FIX* läßt ein Erfassen des Feldwertes nur über die Choice zu (**use**)
ch_option	*FIX* bietet zur Feldwerterfassung zunächst die Choice an,[1] erlaubt aber auch eine direkte Änderung des Feldinhalts (**offer**)
ch_demand	die Choice erscheint nur auf Anforderung (**choice**)

Varianten

Varianten, die in der gleichen Form, aber getrennt von der Hauptmaske in einer eigenen Beschreibungsdatei abgelegt sind, werden prinzipiell wie eigenständige Masken geladen. Dies geschieht erst dann, wenn ein Wechsel zu dieser Variante erfolgen soll. Passt die Variantenbeschreibung nicht zur Hauptmaske, wird die Variante *gesperrt*, d.h. ein Wechsel zur ihr ist nicht möglich.

Bei einem Variantenwechsel werden die differierenden Attribute von Grundmaske und Variante ausgetauscht. Davon sind betroffen:

Maske:

- Maskenart (*ob_class*)
- Windowbeschreibung (*ob_box*)
- Maskenname gemäß Beschreibungsdatei (*ms_vname*)
- Prompt (*ob_prompt*)
- Startelement (*ms_startfield*)
- Layout

Abgeleitete Attribute, wie z.B. bei Tabellenmasken die Maximalzahl sichtbarer Zeilen oder die Position des aktuellen Satzes im Window, werden wenn notwendig neu bestimmt.

Feld:

- Position (*z*, *s*)
- Genauigkeit bei numerischen Feldern ohne Formatangabe (*len*)
- Display-Länge bei CHARACTER-Feldern (*displen*), Anzahl Nachkommastellen bei numerischen Feldern ohne Formatangabe
- Feldeigenschaften gemäß Beschreibungsdatei (*ostyp*)
- Prompt (*prompt*)
- Anschlussmodus der Choice, sofern vorhanden (*f_slctmode*)

eingebettete Maske:

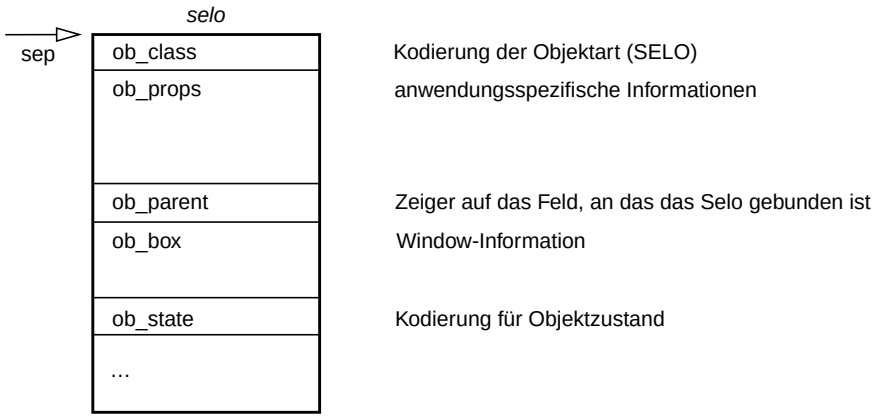
Die aktuellen Eigenschaften bleiben unverändert.

4 Selos

Selos werden im Programm durch eine dynamisch allokierte Datenstruktur folgender Form repräsentiert:

1. nur, wenn das Feld leer ist und die Eigenschaft REQUIRED besitzt.

Abb. 32 Selo-Datenstruktur



Anmerkungen

Bei der Komponente *ob_box* sind folgende Besonderheiten zu beachten:

- *bx_b_y* und *bx_b_x* enthalten nur gültige Werte, solange das Selo am Bildschirm dargestellt ist oder wenn die Position in der Beschreibungsdatei fest vorgegeben ist.
- *bx_b_lines*, *bx_cols* und *bx_headline* enthalten nur gültige Werte, solange das Selo am Bildschirm dargestellt ist.

ob_state enthält, in kodierter Form, Eigenschaften und augenblicklichen Zustand des Selos, unter anderem:

(ob_state & ACTIVE) == ACTIVE	Objekt ist in Bearbeitung

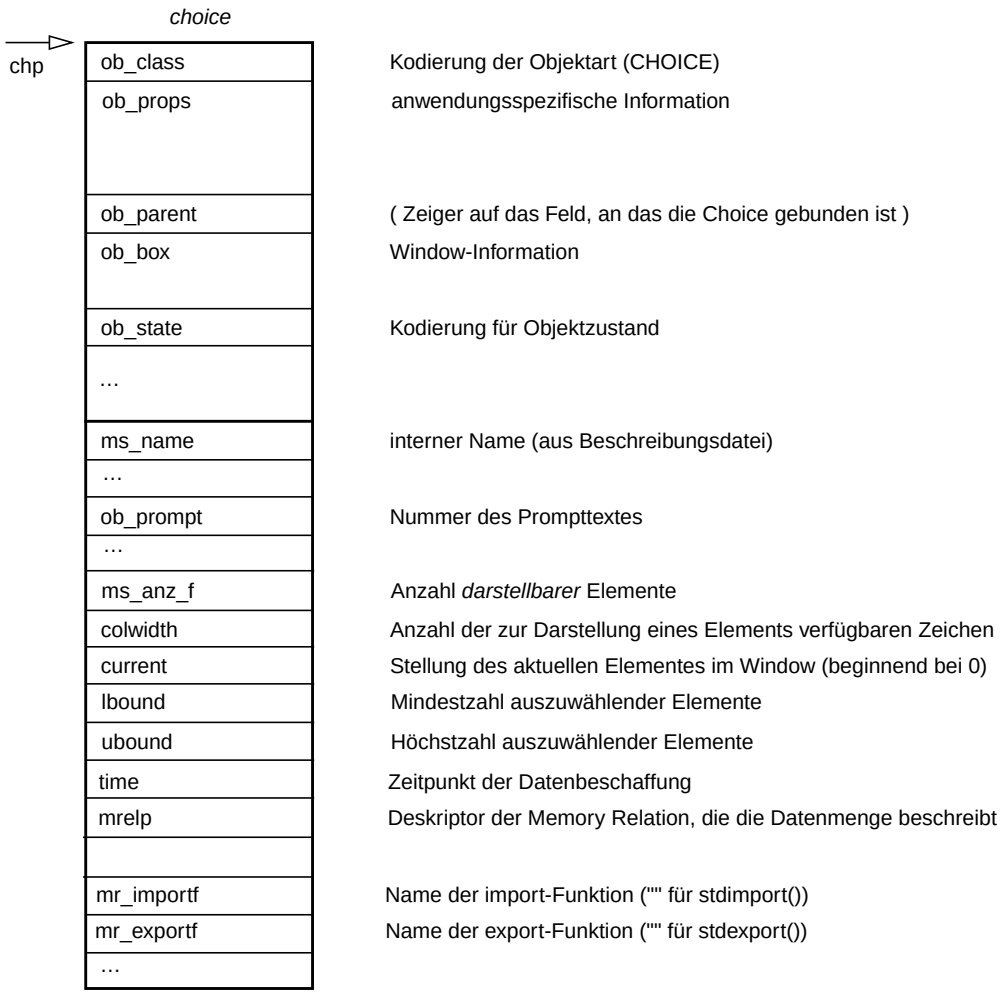
(STEADY|ABOVE) bzw. STEADY werden bei Selos nicht unterstützt, d.h. ein Selo wird von *FIX* nach der Benutzung stets zum Verschwinden gebracht.

FIX führt über geladene Selos Buch. Wenn *FIX* ein Selo zu laden versucht, und ein gleichnamiges Selo ist bereits geladen, wird die Beschreibungsdatei nicht erneut gelesen, sondern eine *partielle* Kopie des schon vorhandenen Selos benutzt (partiell deshalb, da viele Teile durch Sharing gemeinsam nutzbar sind).

5 Choices

Beim erfolgreichen Laden einer Choice erhält der Entwickler einen Zeiger auf eine dynamisch allokierte Datenstruktur folgender Form:

Abb. 33 Choice-Datenstruktur



Anmerkungen

ob_state enthält, in kodierter Form, Eigenschaften und augenblicklichen Zustand der Choice, unter anderem:

(ob_state & (STEADY|ABOVE)) == (STEADY|ABOVE)	Objekt muss stets oberhalb seines übergeordneten Objekts liegen
(ob_state & STEADY) == STEADY	Objekt soll nach der Bearbeitung am Bildschirm bleiben
(ob_state & ACTIVE) == ACTIVE	Objekt ist in Bearbeitung

Die Datensätze werden in einer Memory Relation gespeichert. Deren erstmaligen bzw. Neuaufbau steuert die Komponente *time*, deren Wert entweder CH_STATIC_DATA (bei Gebrauch vorhandene Datenmenge benutzen) oder CH_DYNAMIC_DATA (bei Gebrauch Datenmenge neu bestimmen) sein muss. *FIX* initialisiert *time* beim Laden mit CH_DYNAMIC_DATA, wenn die Datenbeschaffungsangabe in der Beschreibungsdatei nicht den Zusatz “static” beinhaltet.

6 Layout

Die Datenstruktur eines Layouts ist für den Entwickler nicht zugänglich. Daher folgen nur einige Anmerkungen zum Laden von Layouts.

Layoutbeschreibung

Eine Layoutbeschreibung wird erwartet, wenn der Name der Layoutdatei den Suffix .pan besitzt.

Stimmt die Dimension im Datei-Kopf nicht mit der Objektgröße überein, werden Zeilen- oder Spaltenbereiche bei der Zuweisung an das Objekt geeignet abgeschnitten oder aufgefüllt.

Beziehen sich zwei Anweisungen zum Schreiben oder Zeichnen auf dieselbe Position des Layouts, ist das Ergebnis undefiniert.

Betreffen zwei Anweisungen zum Setzen von Videoattributen dieselbe Position des Layouts, ist das Ergebnis undefiniert.

Binäres Layout

Bei vorzeitigem Dateiende wird das Layout mit Leerzeichen aufgefüllt, überzählige Bytes werden ignoriert.

25 Memory Relations

1 Einführung

Masken[1] und Choices basieren auf *Memory Relations*. Bei einer Memory Relation handelt es sich um eine Datenstruktur zur Bearbeitung von Satzmengen, die als eine "Tabelle im Speicher" angesehen werden kann, mit Operationen, die denen eines Scrollcursors ähnlich sind.

Wie bei einer Datenbanktabelle bestehen die Sätze einer Memory Relation aus einer oder mehreren getypten Spalten. Als Spaltentypen sind alle *FIX*-Feldtypen möglich und auch das Speicherformat für Werte ist mit dem für Feldwerte identisch.

Sofern eine Memory Relation überhaupt Sätze enthält, besitzt sie stets einen aktuellen Satz.

Jeder Satz besitzt eine bzgl. der Memory Relation eindeutige *ROWID*. Dies ist eine mit 1 beginnende fortlaufende Nummer, die bei jedem neu angelegten Satz erhöht wird. Durch das Löschen von Sätzen entstehende Lücken in der Nummerierung werden nicht wieder gefüllt.

Im Gegensatz zu einer Datenbanktabelle sind die Sätze einer Memory Relation - üblicherweise gemäß der Historie des Aufbaus der Satzmenge - *angeordnet* und jeder Satz ist zusätzlich mit Statusinformation versehen. Neben von *FIX* verwalteten Attributen (z.B. TOUCHED, "alt") stehen auch 4 anwendungsspezifische Attribute (TP_USR1, TP_USR2, TP_USR3, TP_USR4) zur Verfügung.

Viele der auf Memory Relations definierten Operationen stehen auch dem Anwendungsentwickler zur Verfügung. Memory Relations können unabhängig von Masken angelegt, manipuliert und wieder freigegeben werden. Für den Entwickler stellt sich eine Memory Relation als eine "black box" dar, zu der er mittels eines Deskriptors vom Typ *MemRelType* Zugang hat. Die Spaltenwerte eines Datensatzes sind zusammen mit dessen Statusinformation in ein *Tupel* verpackt, das durch einen Deskriptor vom Typ *TupelType* angesprochen wird.

Somit können komplexe Anwendungen Satzmengen mit der von Mehrsatz-Masken gewohnten Bequemlichkeit und vertrauten Operationen, aber ohne den mit Masken- und Bildschirmbehandlung verbundenen Overhead bearbeiten.

2 Operationen auf Memory Relations

Die nachfolgend beschriebenen Operationen dürfen, sofern sie nicht wie die in den Abschnitten 2.2, 2.4, 2.8, 2.9 nur lesend zugreifen, nicht auf Memory Relations angewendet werden, die an eine Maske gebunden sind; hier sind die entsprechenden Maskenroutinen zu verwenden.[2] Einzige Ausnahme bildet die Funktion mr_qsort().

Näheres zu den Funktionen finden Sie im Kapitel *Funktionen der FIX-Library*.

1. Seit *FIX* 2.9.5 auch Einzelsatz-Masken: hier besitzt die Memory Relation stets genau einen Satz.
2. Die Bindung einer Memory Relation kann durch fx_mrgetobject() ermittelt werden.

2.1 Anlegen einer Memory Relation

Das Erzeugen einer Memory Relation (für Mehrsatz-Masken und Choices geschieht dies automatisch, vgl. Seite 186) verläuft in mindestens 3 Schritten:

- Kreieren

  ```
  int  mr_create(MemRelType *mrp_adr, int num_of_columns)
  ```

- Definition der Spalten

  ```
  int  mr_define_column(MemRelType mrp, int colno, struct colspec *buf)
  int  mr_bind_column(MemRelType mrp, int colno, char *adr)
  ```

- Nutzbar machen

  ```
  int  mr_activate(MemRelType mrp)
  ```

Die Struktur *colspec* hat hierbei die Form

```
struct colspec {
    char *colname;          /* Name der Spalte */
    int coltype;            /* Typ der Spalte */
    int collen;             /*
                             * FXCHARTYPE, FXGRAPHICSTYPE: Länge der Spalte (in Bytes)
                             * FXDTIMETYPE, FXINVTYPE: Qualifier
                             */
    int colnull;            /* sofern > 0, werden NULL-Werte unterstützt */
    char *colbinding;       /* Variablenbindung */
    …
};
```

Anmerkung:

Bei den Typen FXDTIMETYPE und FXINVTYPE muss der Qualifier der zu bindenden Variable mit dem der Spalte übereinstimmen. Wenn der Qualifier der Variablen gleich 0 ist, vermutet *FIX* eine nicht initialisierte Variable und überträgt den Qualifier der Spalte in die Variable.

Beispiel

Angelegt werden soll eine Memory Relation mit 3 Spalten:
zuname (CHAR(10) NOT NULL), groesse (FLOAT) und gebdatum (DATE).

```
MemRelType mrp = 0;
struct colspec col;
int ret;

#define SP_ZUNAME       1
#define SP_GROESSE      2
#define SP_GEB          3

ret = mr_create(&mrp, 3);
if (ret == SUCCESS) {
    col.colname = "zuname"; col.coltype = FXCHARTYPE; col.collen = 10; col.colnull = 0;
        col.colbinding = (char *)0;
        (void)mr_define_column(mrp, SP_ZUNAME, &col);
    col.colname = "groesse"; col.coltype = FXFLOATTYPE; col.colnull = 1;
        /* eine Laengenangabe bleibt hier unausgewertet */
        col.colbinding = (char *)0;
        (void)mr_define_column(mrp, SP_GROESSE, &col);
```

```
    col.colname = "gebdatum"; col.coltype = FXDATETYPE;
        /* eine Laengenangabe bleibt hier unausgewertet, NULL-Unterstuetzung ist
           obligatorisch */
        col.colbinding = (char *)0;
        (void)mr_define_column(mrp, SP_GEB, &col);
    ret = mr_activate(mrp);
}
```

Sofern kein Fehler auftrat (ret == SUCCESS), ist die Memory Relation nun benutzbar. Allerdings gibt es noch keine Möglichkeit, eigene Werte darin abzulegen. Dies wird erst durch das Binden von Spalten mit typkonformen Variablen ermöglicht (der Einfachheit halber wollen wir hier keine Fehler behandeln):

```
char zuname[11];
float groesse;
long gebdatum;

(void)mr_bind_column(mrp, SP_ZUNAME, zuname);
(void)mr_bind_column(mrp, SP_GROESSE, (char *)&groesse);
(void)mr_bind_column(mrp, SP_GEB, (char *)&gebdatum);
```

Das Binden erfolgt hier nur zu Demonstrationszwecken separat; man kann die Bindung auch beim Aufruf von mr_define_column() mit angeben:

```
… col.colbinding = (char *)&gebdatum;
```

2.2 Erweiterung um Anwendungsdaten

Wie Objekte können auch Memory Relations mit anwendungsspezifischer Information versehen werden (vgl. Seite 140):

```
void     mr_set_props(MemRelType mrp, PrivatePropertyType value)
PrivatePropertyType mr_get_props(MemRelType mrp)
```

2.3 Aufbau mittels Datenbankanfrage

Eine Memory Relation kann auch per Pseudo-select-Anweisung (vgl. Seite 105) aufgebaut und mit Sätzen gefüllt werden:

```
int  mr_create_by_query(MemRelType *mrp_adr, char *stmt)
```

Anmerkung:

Eine into-Klausel in *stmt* wird nicht unterstützt.

2.4 Beschaffung von Strukturinformation über eine vorhandene Memory Relation

Zur Ermittlung der Spaltenanzahl und der Definition einer Spalte dienen

```
int  mr_num_of_columns(MemRelType mrp)
int  mr_explain_column(MemRelType mrp, int colno, struct colspec *buf)
```

Die als Suchspalte ausgezeichnete Spalte kann abgefragt werden mit

```
int  mr_get_selected_column(MemRelType mrp)
```

2.5 Freigeben einer vorhandenen Memory Relation

Das Löschen einer Memory Relation (Freigeben aller Datenstrukturen einschließlich der Sätze) erfolgt durch

```
int  mr_drop(MemRelType *mrp_adr)
```

2.6 Modifizieren der Satzmenge

Die nachfolgenden Funktionen setzen voraus, dass die Memory Relation aktiviert ist.

Ändern des aktuellen Satzes

```
int  mr_update_row(MemRelType mrp)
```

Einfügen vor dem aktuellen Satz

```
int  mr_insert_row(MemRelType mrp)
```

Anfügen hinter dem letzten Satz

```
int  mr_add_row(MemRelType mrp)
```

Löschen des aktuellen Satzes

```
int  mr_delete_row(MemRelType mrp)
```

Sortieren der Sätze

```
int  mr_qsort(MemRelType mrp, int colno, int dir, … /* , 0 */ )
```

Über die bloßen Werte hinaus verwaltet *FIX* zu jeder *Zelle* eine Änderungsmarkierung und setzt diese beim Belegen der Zellen neuer Sätze mit Werten und beim Eintragen von Werten, die sich von dem vorgefundenen Wert unterscheiden. Anwendungen können diese Änderungsmarkierung auslesen und zurücksetzen (vgl. Seite 154).

2.7 Lesen eines Satzes

```
int  mr_fetch_row(MemRelType mrp, int indicator, char *value, TupelType *tp_adr)
```

wechselt zu dem durch *indicator* und *value* bestimmten Satz, wobei folgende Werte möglich sind:

indicator	value
FIRST	(char *)0
LAST	(char *)0
CURRENT	(char *)0
NEXT	(char *)0
PREVIOUS	(char *)0
INDEXED	(char *)(*absolute Position des Satzes*)
RELATIVE	(char *)(*relative Position des Satzes zum aktuellen Satz*)
EQUAL	(char *)(*Zeiger auf typgleichen Vergleichswert*)
GTEQ	(char *)(*Zeiger auf typgleichen Vergleichswert*)

GREATER (char *)(*Zeiger auf typgleichen Vergleichswert*)

Voraussetzung für das Auffinden über einen Vergleichswert ist, dass eine Spalte bestimmt worden ist, auf die sich der Vergleich beziehen soll:

```
int  mr_select_column(MemRelType mrp, int colno)
```

Vergleichswerte vom Typ FXGRAPHICSTYPE müssen auch hinsichtlich der Länge mit der Spalte übereinstimmen.

Beispiel (Fortsetzung):

Vorausgesetzt, dass die Variablen name, groesse und gebdatum geeignete Werte enthalten, kann ein Satz mit diesen Werten angefügt werden durch den Aufruf

```
mr_add_row(mrp);
```

Der Satz zum Namen “Meier” kann gelöscht werden mittels

```
mr_select_column(mrp, SP_ZUNAME);
if (mr_fetch_row(mrp, EQUAL, "Meier", (TupelType *)0) == SUCCESS)
    /* der Satz mit dem Wert “Meier” in der Spalte SP_ZUNAME ist aktueller Satz
       geworden */
    mr_delete_row(mrp);
```

Die Sätze werden nach der Spalte *SP_ZUNAME* aufsteigend sortiert durch

```
mr_qsort(mrp, SP_ZUNAME, ASCENDING, 0);
```

wobei die Sortierung stabil ist, da *FIX* automatisch die vorherige Position als weiteres Sortierkriterium heranzieht.

2.8 Beschaffung von Informationen zu Satzmenge und aktuellem Satz

Anzahl der Sätze

```
int  mr_num_of_rows(MemRelType mrp)
```

Position des aktuellen Satzes

```
int  mr_act_row(MemRelType mrp)
```

ROWID des aktuellen Satzes

```
long    mr_rowid(MemRelType mrp)
```

2.9 Operationen auf Tupel-Ebene

Tupel suchen

```
int     mr_get_tupel_variadic(MemRelType mrp, int indicator, ...)
int     mr_get_tupel(MemRelType mrp, int indicator, char *indarg, TupelType *tp_adr,
                     int *act_row_adr)
```

ROWID eines Tupels ermitteln

```
long    tp_rowid(TupelType tp)
```

Attribute eines Tupels ermitteln oder ändern

```
unsigned long    tp_state(TupelType tp, int mode, unsigned long state)
```

Zeiger auf Datenteil eines Tupels ermitteln

```
char   *tp_data(TupelType tp)
```

Zeiger auf Spaltenwert eines Tupels ermitteln

```
char   *mr_colval(MemRelType mrp, int n, TupelType tp)
```

Änderungsmarkierung zu einer Tupelzelle abfragen oder zurücksetzen

```
void   mr_isCellUpdateMarkSet(MemRelType mrp, TupelType tp, int n)
void   mr_clearCellUpdateMark(MemRelType mrp, TupelType tp, int n)
```

Weiterhin stehen Operationen zur Verfügung, die einen direkten Zugriff auf bestimmte Tupel ermöglichen:

```
TupelType    mr_first_tup(MemRelType mrp)
TupelType    mr_last_tup(MemRelType mrp)
TupelType    mr_act_tup(MemRelType mrp)
TupelType    mr_nth_tup(MemRelType mrp, int n)
TupelType    tp_next(TupelType tp)
TupelType    tp_previous(TupelType tp)
```

Keine dieser Funktion führt zu einem Wechsel des aktuellen Tupels.

2.10 Memory Relations mit Auswahlmarkierung

Diese Funktionen sind speziell für Memory Relations gedacht, die als Datenmenge von Choices dienen. Deren Tupel tragen eine Markierung, die sie als “selektiert” erkennen lässt.

Tupelmarkierung setzen

```
void   mr_tp_set_mark(MemRelType mrp, TupelType tp, BOOLEAN state)
```

Tupelmarkierung auswerten

```
BOOLEAN    tp_get_mark(TupelType tp)
```

letztes markiertes Tupel finden

```
int    last_selected(MemRelType mrp, TupelType *tp_adr, int *n_adr)
```

26 Events

1 Übersicht

Der Entwickler hat im Programm umfassend Gelegenheit, Reaktionen auf das Anwenderverhalten festzulegen. Den Schlüssel hierzu bilden *Events* (“Ereignisse”) und die für sie definierten Rückwirkungen. Events werden ausgelöst durch die - tatsächlich erfolgte oder vom Programm simulierte - Betätigung von Tasten, Klicken (bei Einsatz eines entsprechenden Frontends) oder durch bestimmte Programmzustände.

Jedes Event ist durch eine ganze Zahl kodiert.

1.1 Tasten-Events

Die Kodierung der Events, die für Zeichen erzeugt werden, entspricht dem Zeichencode. Für die von Sondertasten ausgelösten Events sind in der Header-Datei `fix/keys.h` Makros (mit Werten kleiner 0 oder größer 255) definiert, die von den Tastenbezeichnungen abgeleitet sind:

Taste: Xy
Event: K_Xy

Für einige Tasten-Events existieren zusätzlich eingängigere Bezeichnungen, die meist an die Standardbedeutung der Taste bei der Bearbeitung von Masken angelehnt sind (und die auch nur in diesem Kontext benutzt werden sollten):

K_ku	L_CUR_UP
K_kd	L_CUR_DOWN
K_kl	L_CUR_LEFT
K_kr	L_CUR_RIGHT
K_TB	L_EO_FIELD
K_BT	L_SO_FIELD
K_DL	L_RUBOUT
K_CE	L_DELFIELD
K_CI	L_CUR_INS
K_CD	L_CUR_DEL
K_WD	L_DELREST
K_kh	L_FIRSTFIELD
K_PL	L_PRVFIELD
K_RT	L_NXTFIELD
K_EN	L_END
K_PU	L_BACKBROWSE
K_PD	L_FORWBROWSE
K_HP	L_HELP
K_ST	L_START

K_MD	L_MODE
K_PT	L_PRINT
K_sh	L_SHELL
K_f2	L_CPYFIELD
K_f3	L_JMPFIELD
K_f4	L_INSERT
K_f5	L_DETAIL
K_f6	L_ADD
K_f7	L_DELETE
K_f8	L_KEYFIRST
K_f9	L_KEYSELECTION
K_f0	L_KEYLAST
K_g1	L_REBUILD
K_g5	L_T_HELP
K_g6	L_CHELP
K_g7	L_UNDO

Daneben existieren einige Events, denen keine spezielle Taste zugeordnet ist (vgl. “Schritt 3: Standardlogik” auf Seite 200):

—	L_COPY_RECORD
—	L_CUT_RECORD
—	L_INSERT_CLIPBOARD
—	L_ADD_CLIPBOARD

FIX/Win

In Verbindung mit einem grafischen Frontend wird auch eine Maus unterstützt. Ein Klicken mit einer Maustaste entspricht den Events:

BT_LEFT	—
BT_RIGHT	—
BT_MIDDLE	—
BT_LEFTDBL	—
BT_MIDDLEDBL	—
BT_RIGHTDBL	—

Wird das Feature Paintarea genutzt, so werden beim Klick mit der Maus auf eine Paintarea, je nachdem welche Klicks als empfangbar an der Paintarea eingestellt wurden, folgende Events generiert.

BT_LEFT_PA	—
BT_MIDDLE_PA	—
BT_RIGHT_PA	—

1.2 Logische Events

Bei gewissen Bearbeitungszuständen von Objekten, die erfahrungsgemäß häufig anwendungsspezifische Eingriffe in den Programmablauf notwendig machen, generiert *FIX* darüberhinaus logische (d.h. nicht tastengebundene) Events:

L_LOAD_OBJECT

zeigt an, dass ein Menü erfolgreich geladen wurde (aufgrund eines Aufrufs von loadmenue() oder des Auslösens einer Menü-Aktion, die ein Untermenü benutzt).

L_FREE_OBJECT

zeigt an, dass ein Menü freigegeben wird (aufgrund eines Aufrufs von o_free() oder durch die Rückkehr in ein übergeordnetes Menü).

L_ENTER_OBJECT

erfolgt zu Beginn der Menü- und Maskenbearbeitung.

L_ENTER_ENTRY

zeigt das Betreten eines Menüpunktes an.

L_LEAVE_ENTRY

zeigt das Verlassen eines Menüpunktes an.

L_BEFORE_EXECUTION

zeigt an, dass die mit dem aktuellen Menüpunkt verknüpfte Aktion unmittelbar vor der Ausführung steht.

L_AFTER_EXECUTION

zeigt an, dass die Ausführung der mit dem aktuellen Menüpunkt verknüpften Aktion beendet ist.

L_QUIT_OBJECT

zeigt an, dass die Bearbeitung eines Menüs beendet werden soll.

L_ENTERFIELD

erfolgt jedes Mal, wenn ein Maskenelement besucht wird (auch bei einem erneuten Besuch des zuletzt besuchten Elements).

L_MSWAP

erfolgt bei der Maskenbearbeitung, wenn bei der Standardreaktion auf das Event L_MODE ein Wechsel zu einer anderen Variante stattgefunden hat.

L_SELOSUCCESS

erfolgt bei der Maskenbearbeitung, wenn bei der Standardreaktion auf das Event L_KEYSELECTION das Selo oder die Choice “vorwärts” verlassen wurde.

L_ENTER_RECORD

generiert *FIX* bei Mehrsatz-Masken, wenn ein Satz neu betreten worden ist; dieses Event kann insbesondere dazu benutzt werden, um satzbezogen Datenstrukturen zu besetzen oder Veränderungen an anderen Objekten vorzunehmen.

Bei der Behandlung von L_ENTER_RECORD darf der aktuelle Satz nicht gewechselt werden.

L_LEAVE_RECORD

generiert *FIX* bei Mehrsatz-Masken, bevor ein Satz verlassen wird; dieses Event erleichtert Konsistenzprüfungen oder Aufräumarbeiten vor Verlassen eines Satzes. Auch kann der Entwickler ein Verlassen des Satzes verhindern.

Bei der Behandlung von L_LEAVE_RECORD darf der aktuelle Satz nicht gewechselt werden.

L_LEAVE_OBJECT

kann als Gegenstück zu L_ENTER_OBJECT betrachtet werden und erleichtert es, vor dem Verlassen eines Menüs oder einer Maske - unabhängig von dem speziellen Event, dessen Behandlung dies auslöst - Konsistenzprüfungen oder Aufräumarbeiten vorzunehmen. Bei Masken kann bei inkonsistenten Daten ein Verlassen des Objekts durch den Anwender verhindert werden.

Zwei Events sind speziell als Rückgabewerte für Event-Behandlungsroutinen (vgl. Seite 163) definiert:

NOTHING

bewirkt bei der Maskenbearbeitung die Standardreaktion von *FIX* auf das der Event-Behandlungsroutine angebotene Event (siehe aber Fußnote auf Seite 171).

L_STAY

unterbindet die Standardreaktion von *FIX* auf das der Event-Behandlungsroutine angebotene Event.

Mehr zu den logischen Events finden Sie in “Menübearbeitung aus Sicht des Entwicklers” auf Seite 167 und “Maskenbearbeitung aus der Sicht des Entwicklers” auf Seite 197.

2 Empfang von Tasten-Events

Wenn ein *FIX*-Programm zum weiteren Ablauf das nächste Event benötigt, schaut es zunächst in einem internen Puffer nach, in den es zuvor möglicherweise Events zurückgestellt hat (simulierte Eingabe). Ist dieser Puffer, der maximal 80 Events aufnehmen kann, leer, wird - bei Terminalbetrieb - die Tastatur abgefragt. Es werden solange Zeichen gelesen, bis die Sequenz einer Taste erkannt ist oder offensichtlich ist, dass die empfangene Zeichenfolge keine bekannte Sequenz mehr ergeben kann. Bei Einsatz eines grafischen Frontends wird von diesem ein Event angefordert.

Verantwortlich für die Eventerkennung ist die Funktion

```
Event  dcgetch(void)
```

die das zurückgestellte/erkannte Event zurückgibt oder, im Fehlerfall, die gelesenen Zeichen verwirft und NOTHING zurückgibt.

Die Taste hc wird innerhalb der Funktion dcgetch() behandelt, wo sie die Erstellung einer Hardcopy des Bildschirms bewirkt; da sich dcgetch() dann selbst rekursiv aufruft, tritt ein entsprechendes Event auf Anwendungsebene nie auf.

Der Entwickler kann mit den Funktionen

```
Event  undcgetch(Event c)
int    undcgetstr(unsigned char *str)
```

Events in den Puffer (Stack, LIFO) zurückstellen.

undcgetch() stellt das Event *c* zurück; im Fehlerfall (Pufferüberlauf) gibt die Funktion NOTHING, sonst *c* zurück.

undcgetstr() stellt die den String *str* bildenden Zeichen so zurück, dass das erste Zeichen von *str* als erstes wieder empfangen wird; die Funktion liefert die Anzahl der erfolgreich zurückgestellten Zeichen.

Hinweis:

Da die Implementierung von *FIX* die Möglichkeit, Eingabe zu simulieren, an einigen wenigen Stellen selbst ausnutzt (vgl. Seite 199 und Seite 208), kann sich beim Zurückstellen von Events in seltenen Fällen ein anderes als das vom Entwickler erwartete Verhalten ergeben.

Ein Zugriff auf die mit BT_LEFT assoziierte Mauszeiger-Position ist nicht implementiert.

3 Empfang von Maus-Events

3.1 BT_... - Events bei der Felderfassung

Menüs, Selos und Choices reagieren nur auf das Event BT_LEFT. Die übrigen Events werden zwar an die Anwendungslogik weitergereicht, jedoch reagiert die *FIX*-Standardlogik dieser Objekte nicht darauf.

Die Anwendungslogik von klicksensitiven Masken reagiert in der gleichen Art auf alle Maus-Events. Mittels o_RetrieveItemClickedOn() und o_RetrievePositionClickedOn() kann auf die notwendigen Informationen zugegriffen werden. Wird mit einer struct clickinfo gearbeitet, dann enthält die Komponente ev das der Maustaste zugeordneten Event.

FIX/Win sendet die oben definierten Events zu den Maustasten nur, wenn den Maustasten in der Tastaturtabelle kein anderer Event zugeordnet wurde. Andernfalls wird das in der Tastaturtabelle definierte Event gesendet, das wie ein normales Tastenevent behandelt wird. Die rechte Maustaste darf nicht mit einer Kopierfunktion belegt werden (*FIX/Win*-Menü Einstellungen/Kopiermodus.../Rechte Maustaste). Ist das der Fall, dann hat die Kopierfunktion Vorrang vor dem Maustasten-Event.

3.1.1 Übliches Verfahren

Situation: Beim Besuch eines Maskenfeldes trifft ein BT_...-Event ein.

(1) Dient das Feld zur Erfassung von Query-Parametern (Selo), wird das Event ignoriert und die Felderfassung fortgesetzt.

(2a) Ist die Maske klicksensitiv (**enable mouse**), so ermittelt *FIX* die Position, wohin geklickt wurde. Um zu bestimmen, ob dort ein Feld der *aktiven* Maske liegt, wird diese im virtuellen Bildschirm neu aufgebaut, wobei zum “Malen” der Felder ein spezieller Modus benutzt wird, der im perform()-Kontext[1] “Zielbereiche” registriert (Beachte: auf Grund der Implementierung von Tabellen kann “dasselbe” Feld mehrfach an verschiedenen Positionen vorkommen). Erst nach dem Neuaufbau kann entschieden werden, ob und wenn ja auf welches Feld (welchen Satzes) der Maske geklickt wurde.

Ist kein Ziel zu finden (vgl. “Hinweis” auf Seite 160), wird das Event ignoriert und die Felderfassung fortgesetzt.

Anderenfalls wird die gewonnene Information zur späteren Verwendung durch o_RetrievePositionClickedOn() oder o_RetrieveItemClickedOn() in der Anwendungslogik bzw. zur Verwendung durch *FIX* nach deren Aufruf im perform()-Kontext bereitgestellt und die Felderfassung endet.

(2b) Ist die Maske klickinsensitiv (**disable mouse**), wird das Event ignoriert und die Felderfassung fortgesetzt.

(2c) Im Defaultfall wird das Event wie eine Sondertaste behandelt, d.h. die Felderfassung endet. Die Behandlung des Events bleibt ins Belieben der Anwendungslogik gestellt.

1. einer automatischen struct-Variablen der Funktion perform(), auf die das Objekt eine Referenz besitzt.

Hinweis

Der Entwickler sollte sich folgender Anomalie bewusst sein:

FIX/Win übermittelt bei einem Klicken mit einer Maustaste nur dann ein Event an die *FIX*-Anwendung, wenn das Klicken an einer Stelle erfolgt, die der Darstellung eines Feldes (inkl. Begrenzer) oder eines Menüpunktes dient. Grundlage der Entscheidung ist hierbei der Bildschirminhalt, den das Frontend zu diesem Zeitpunkt kennt.

Die Anwendung interpretiert das Event hingegen entsprechend ihrer Vorstellung vom Bildschirm. Zwischenzeitlich anwendungsseitig erfolgte Änderungen der Position, Dimension oder der Eigenschaften von Objekten oder -elementen, die (noch) nicht sichtbar gemacht wurden, können also unerwartete Effekte bewirken.

3.1.2 Erweiterung

Von Entwicklern wurde der Wunsch geäußert, auch Felder anderer als der aktiven Maske anklickbar machen zu können. Hierzu wurde der nachfolgend beschriebene Mechanismus geschaffen, der in einem solchen Fall ein Ende des Feldbesuchs und einen Aufruf der Anwendungslogik erlaubt.

Auf Satzebene bzw. Satzmengenebene erfolgt eine Behandlung von BT_... seitens *FIX* jedoch weiterhin lediglich in der bislang dokumentierten Weise.

struct clickinfo *S_BtLeftFromField: r/w

Zeigt diese globale Variable (Vorbelegung: (struct clickinfo *)0) auf eine clickinfo-Struktur, bleibt die Vorgabe, wie die aktive Maske auf BT_...-Events reagieren soll, unbeachtet, und bei der Behandlung des Events wird nicht nur das aktive Objekt berücksichtigt. Vielmehr kommt als Ziel eine an dieser Position sichtbare (d.h. nicht überdeckte) Stelle eines Feldes einer beliebigen Maske bzw. eines Menüpunktes eines beliebigen Menüs in Betracht.

Durch vorheriges Setzen von Steuerwerten kann *FIX* veranlasst werden, zu fordern, dass das Objekt, zu dem das angeklickte Element gehört, bestimmte Bedingungen erfüllt, z.B. für es ein Aufruf von perform() oder m_present() anhängig ist. Dies erlaubt es, gewisse Arten von Klicks zu unterdrücken, die anderenfalls unnötigerweise den Feldbesuch beenden und einen Aufruf der Anwendungslogik bewirken würden.

Die Struktur

```
struct clickinfo {
    /*
     *  von der Anwendung zu setzende Steuerwerte
     */
    BOOLEAN     fields_only;
        /*
         *  falls TRUE:
         *  FIX fuellt Struktur nur, wenn angeklicktes Element Feld eines
         *  Objekts vom Typ MASK, SUBMASK, ROLLMASK, TABLE ist
         */
    BOOLEAN     activated;
        /*
         *  falls TRUE:
         *  FIX fuellt Struktur nur, wenn angeklicktes Element zu einem
         *   Objekt gehoert, das in Bearbeitung ist.
         */


    /*
     *  von FIX gefuellte Werte
     */
    Event  ev;
        /* Art des Klickens (z.Z. stets BT_LEFT) */
    short   y;
    short   x;
        /* Position, auf die geklickt wurde */
    obj     *objp;
        /* Objekt, das das angeklickte Element beinhaltet */
```

```
        int     fnr;
            /*
             * sofern objp vom Typ MASK, SUBMASK, ROLLMASK, TABLE:
             * Element-Position des Feldes von objp, auf das geklickt wurde
             *  (0 <= fnr < ((mask *)objp)->ms_anz_f);
             * sofern objp vom Typ MENUE:
             * Element-Position des Menuepunktes von objp, auf den geklickt wurde
             *  (0 <= fnr < ((menue *)objp)->ms_anz_f);
             * sonst < 0.
             */

        int     buttonIdx;
            /*
             * Index des Buttons der angeklickt wurde oder -1.
             */

        int     row_nr;
            /*
             * sofern objp vom Typ TABLE:
             * Position des Satzes innerhalb der dargestellten Saetze
             * (1 <= row_nr <= t_max_in_window((table *)objp));
             * sonst 1.
             */
};
```

ist in der Header-Datei `click.h` im Verzeichnis `$FXDIR/include` definiert. Anwendungen, die sie nutzen wollen, müssen die Datei explizit einschließen. Die Datei enthält auch die Deklaration der Variablen S_BtLeftFromField.

Anmerkung zur Implementierung:

FIX (→ faccept()) durchsucht die am Bildschirm dargestellten Objekte vom Vordergrund zum Hintergrund nach dem obersten Objekt, dessen Window die Zielkoordinaten enthält. Für dieses werden die weiteren Bedingungen wie “ist vom richtigen Typ” oder “ist in Bearbeitung” verifiziert. Bei Misserfolg wird das Event ignoriert. Ansonsten wird dieses Objekt - und, um Konsistenz des Bildschirms zu wahren, auch alle darüberliegenden Objekte - im virtuellen Bildschirm neu aufgebaut, um das ggf. an dieser Position befindliche Feld bzw. den dort dargestellten Menüpunkt zu ermitteln (vgl. (2a) auf Seite 159). Wird kein Zielelement gefunden, wird das Event ignoriert. Anderenfalls wird die gewonnene Information in der Struktur, auf die S_BtLeftFromField zeigt, (sowie im perform-Kontext der aktiven Maske) hinterlegt und ein Beenden der Felderfassung eingeleitet.

3.2 Maus-Events bei der READONLY-Bearbeitung von Masken

Die in Abschnitt 3.1.2 beschriebene Erweiterung greift analog bei der Bearbeitung von Masken mittels m_present().

Zur Behandlung von BT_LEFT im Sourcecode vgl. “Behandlung des Events BT_LEFT” auf Seite 497.

4 Ersetzen der Funktion zum Lesen eines Events

Die *FIX*-Library enthält einen globalen Zeiger

```
Event  (*S_ReadEvent)(void)
```

der auf die Funktion zeigt, die *FIX* zum Erkennen des nächsten Events im Eingabestrom aufruft. S_ReadEvent ist mit der (Adresse der) *FIX*-Funktion

```
Event  fxReadEvent(void)
```

vorbelegt.

Diese Konstruktion ermöglicht es dem Anwendungsentwickler, einen *Wrapper* um fxReadEvent() zu schaffen, der beispielsweise “Fehler” durch die Unterbrechung der von dieser Funktion abgesetzten read()-Systemaufrufe abfängt oder einen Timeout (mit entsprechender Ausnahmebehandlung) realisiert. Daneben kann auch eine Protokollierung der Eingabe erfolgen etc.

Der Wrapper wird i. Allg. die Form

```
Event ApplReadEvent(void)
{
    Event ev;

    eigener Code
    ev = fxReadEvent();
    eigener Code
    return(ev);
}
```

haben und muss - üblicherweise vor dem Aufruf von fxinit() - durch eine Zuweisung

```
S_ReadEvent[1] = ApplReadEvent;
```

installiert werden. Der umhüllende Code darf selbstverständlich keine *FIX*-Funktionen verwenden, die wiederum eine Interaktion bedingen könnten.

1. Die Variable muss zuvor als ‘extern Event (∗S_ReadEvent)(void);’ vereinbart werden.

27 Die Behandlung von Events im Programm

Für die im Zuge der Bearbeitung eines Objekts auftretenden Events kann der Entwickler mittels einer eigenen Event-Behandlungsroutine, der *Anwendungslogik*, die vordefinierte Reaktion auf das Event ersetzen, erweitern oder übernehmen.

Events werden hierzu von *FIX* nicht sofort interpretiert, sondern zunächst der für das aktive Objekt vorgesehenen Event-Behandlungsroutine übergeben, die darauf beliebig reagieren kann. Erst deren Rückgabewert - selbst ein Event - bestimmt das weitere Verhalten von *FIX*.

Der zentralen Funktion zur Bearbeitung von Objekten

```
Event  perform(obj *objp, Event (*event_control)(obj *, Event))
```

kann aus diesem Grunde als Argument neben dem Objekt, das bearbeitet werden soll, eine hierfür entwickelte Event-Behandlungsroutine mitgegeben werden.

Anwendungslogik bei Menüs

Bei Menüs kann die anwendungsspezifische Logik in Form einer Funktion

```
Event  event_control(obj *objp, Event ev)
```

dem Objekt entweder in der Beschreibungsdatei zugeordnet, durch Aufruf der Funktion

```
Event  (*o_install_ev_control(obj *objp, Event (*ev_ctrl)(obj *, Event)))(obj *, Event)
```

zugewiesen oder perform() als Argument mitgegeben werden. In den ersten beiden Fällen bleibt die Anwendungslogik, sofern sie nicht durch einen (erneuten) Aufruf von o_install_ev_control() ersetzt wird, bis zur Freigabe des Menüs gültig und hat Vorrang vor derjenigen, die bei der Aktivierung perform() als Argument mitgegeben wird.

Die Anwendungslogik eines Menüs wird auch zur Bearbeitung von Untermenüs herangezogen, sofern diese nicht wiederum ihre eigene Logik besitzen.

Gibt die Anwendung keine Anwendungslogik vor ((Event (*)(obj *, Event))0), benutzt *FIX* eine eingebaute Routine, die lediglich ihr zweites Argument zurückgibt.

Mehr zur Anwendungslogik bei Menüs finden Sie im Kapitel “Menübearbeitung aus Sicht des Entwicklers” auf Seite 167.

Anwendungslogik bei Masken

Einer *Maske* kann im Programm ebenfalls durch Aufruf der Funktion o_install_ev_control() eine Anwendungslogik fest zugeordnet werden. Diese Zuordnung wird bei der nächsten Bearbeitung der Maske, d.h. dem nächsten expliziten oder impliziten Aufruf von perform(), wirksam.

Eine der Maske zugeordnete Funktion hat bei einer Bearbeitung der Maske (und ihrer Varianten) Vorrang vor der perform() als Argument übergebenen Funktion. Um eine bestehende Zuordnung für künftige Bearbeitungen aufzuheben, muss (Event (*)(obj *, Event))0 zugeordnet werden.

Die Anwendungslogik wird auch zur Bearbeitung von eingebetteten Masken herangezogen, sofern nicht diese wiederum ihre eigene Logik besitzen.

Gibt die Anwendung keine Anwendungslogik vor, benutzt *FIX* eine eingebaute Routine, die lediglich das mitgeteilte Event zurückgibt.

Näheres zur Anwendungslogik bei Masken finden Sie im Kapitel “Maskenbearbeitung aus der Sicht des Entwicklers” auf Seite 197.

Anwendungslogik bei Selos

Bei Selos wird eine anwendungsspezifische Logik nicht unterstützt, d.h. die Funktion bleibt unbewertet.

Anwendungslogik bei Choices

Eine Anwendungslogik wird bei Choices zurzeit nur in der Form unterstützt, dass *FIX* für jede Taste, die der Anwender betätigt, das zugehörige Tasten-Event mitgeteilt.

Für künftige Versionen sind logische Events wie “Choice wird betreten”, “Element wird angewählt”, “Element wird (de)selektiert” etc. geplant.

Globale Anwendungslogik

In *FIX*-Programme muss eine Funktion mit Namen global_event_control() eingebunden sein, die die Aufrufe der objektspezifischen Anwendungslogik verpackt:

```
Event  global_event_control(obj *objp, Event (*ev_ctrl)(obj *, Event), Event ev)
```

Sofern die Anwendung keine eigene Funktion dieses Namens definiert, wird ersatzweise ein Modul der *FIX*-Library Funktion eingebunden, das folgende Funktion beinhaltet:

```
Event global_event_control(obj *objp, Event (*ev_ctrl)(obj *, Event), ev)
{
    Event rc;

    rc = (ev_ctrl != (Event (*)(obj *, Event))0? (*ev_ctrl)(objp, ev) : ev);
    return(rc);
}
```

FIX ruft bei der Bearbeitung von Menüs und Masken die Anwendungslogik nie direkt, d.h. in der Form

```
(*ev_control)(objp ,ev),
```

sondern stets durch

```
global_event_control(objp, ev_control, ev)
```

auf.[1]

Dieser Mechanismus erlaubt eine zentrale, d.h. für das gesamte Programm gültige Behandlung von Events und soll die Entwicklung von firmen- oder projektweit angewandten Event-Behandlungsroutinen fördern. Die globale Anwendungslogik kann beispielsweise zur Bereitstellung zentraler Dienste wie Terminkalender, Taschenrechner, Druckerinfo etc. verwendet werden, aber auch, um generell in einer *FIX*-untypischen Weise auf ein Event zu reagieren.

1. Für Choices gilt dies gegenwärtig noch nicht.

28 Programmieren mit Menüs

1 Der Zugriff auf Menüelemente im Programm

Die Elemente eines Menüs sind nur über die Position in der Beschreibungsdatei ansprechbar:

```
menue_entry *men_entry(menue *menp, int i)
```

liefert einen Zeiger auf das *i*-te Element ($0 \leq i <$ Anzahl Elemente) des Menüs *menp*.

2 Verbergen von Menüpunkten

Einzelne Menüpunkte können vor einem Anwender verborgen werden. Der zugehörige Text wird dann nicht dargestellt und der Menüpunkt beim Blättern übersprungen. Es muss jedoch mindestens ein anwählbarer Menüpunkt verbleiben.

Ein Menüpunkt wird verborgen, wenn die Komponente *rights* in seiner Datenstruktur einen Wert enthält, der, maskiert mit 02, ungleich 0 ist.[1]

3 Events beim Laden und Freigeben eines Menüs

Beim Laden eines Menüs versucht *FIX*, die anwendungsspezifische Logik aus der Beschreibungsdatei zu bestimmen. Ist dort eine angegeben (**function** …), werden am Ende des Ladevorgangs das Event L_LOAD_OBJECT und beim Freigeben des Menüs das Event L_FREE_OBJECT an die Anwendungslogik übergeben. In beiden Fällen ist der Rückgabewert ohne Belang.

1. Frühere Versionen von *FIX* interpretierten das durch 01 maskierte Bit als Auslösesperre. Ab Version 2.9.2 sind alle Bits der Komponente *rights* mit Ausnahme des durch 02 maskierten für *FIX* ohne Belang.

29 Menübearbeitung aus Sicht des Entwicklers

FIX ruft die Anwendungslogik eines Menüs (vgl. Seite 163) bei einer Vielzahl von Zustandsänderungen auf, die sich bei der Menübearbeitung aus den Aktionen des Anwenders ergeben oder für die Programmlogik relevant sein könnten. Solche Zustandsänderungen sind z.B.

- das Menü wurde aktiviert (L_ENTER_OBJECT)
- ein Menüpunkt wurde betreten (L_ENTER_ENTRY)
- ein Menüpunkt soll ausgelöst werden (L_BEFORE_EXECUTION)
- ein Menüpunkt soll als “ausgeführt” gelten (L_AFTER_EXECUTION)
- das Menü soll verlassen werden (L_QUIT_OBJECT)
- ein Menüpunkt wird verlassen (L_LEAVE_ENTRY)
- das Menü wird verlassen (L_LEAVE_OBJECT)

Neben dem Event, welches die Art der Änderung angibt, wird der Anwendungslogik beim Aufruf auch ein Zeiger auf das Menüobjekt mitgegeben, wodurch dieselbe Funktion für mehrere Menüs verwendbar ist.

Innerhalb der Anwendungslogik kann der Entwickler vielfältig reagieren:

- Er kann das Event vollständig selbst behandeln (Rückgabe von L_STAY).
- Er kann eigene Aktionen mit dem Event verknüpfen, und die hierdurch bewirkte Zustandsänderung *FIX* durch Rückgabe eines entsprechenden Events mitteilen.
- Er kann durch die Rückgabe des erhaltenen Events die Standardreaktion von *FIX* wirksam werden lassen.

Ein geladenes Menü wird mittels der Funktion perform() unter der Voraussetzung, dass es mindestens einen betretbaren (d.h. nicht verborgenen) Menüpunkt besitzt, aktiviert und erscheint am Bildschirm.

Erfolgt bei der Bearbeitung kein Eingriff seitens der Anwendungslogik (d.h. nimmt deren Code keine Modifikationen vor und gibt er stets das beim Aufruf übergebene Event zurück), so greift die Standardlogik, d.h. *FIX* behandelt die der Anwendungslogik angebotenen Events. Nachfolgend wird zunächst diese Standardverhaltensweise erläutert.

1 Standardverhaltensweise

Aktivieren eines Menüs

Bei der Aktivierung wird zunächst das Event L_ENTER_OBJECT an die Anwendungslogik übergeben. Anschließend wird der erste nicht verborgene Menüpunkt betreten und dies durch das Event L_ENTER_ENTRY angezeigt.

Wenn dies geschehen ist, wartet *FIX* auf Eingabe durch den Anwender.

Zeicheneingabe

Hier wird zuerst das eingegebene Zeichen der Anwendungslogik übergeben (mit Ausnahme von ‘ ’, da *FIX* die Leertaste in die Sondertaste K_kd umsetzt).

Ein alphanumerisches Zeichen wird anschließend in einen Suchstring-Puffer übernommen und ein passender Menüpunkt[1] anzuwählen versucht. Die nachfolgenden Events hängen vom Erfolg der Suche ab, wobei zwischen drei Situationen unterschieden werden muss:

1. Es existiert ein passender Menüpunkt im aktiven Menü.
2. Es existiert ein passender Menüpunkt in einem übergeordneten Menü.
3. Es existiert kein passender Menüpunkt.

zu **1.**

Bei noch unverändertem aktuellen Menüpunkt wird die Anwendungslogik mit dem Event L_LEAVE_ENTRY gerufen. Nach dem Wechsel zu dem lokalisierten Menüpunkt wird der Anwendungslogik das Event L_ENTER_ENTRY übergeben. Der weitere Ablauf hängt ab von den Menüeigenschaften:

- Menü ohne AUTOSTART

  Es wird erneut eine Eingabe erwartet.

- Menü mit AUTOSTART

  Nach dem Event L_ENTER_ENTRY erfolgt ein Auslösen des Menüpunktes (*implizites Auslösen*), dem ein Aufruf der Anwendungslogik mit L_BEFORE_EXECUTION vorausgeht (vgl. Seite 169).

zu **2.**

Dies kann nur der Fall sein, wenn ein Menübaum definiert wurde. Dieser ist dadurch gekennzeichnet, dass das Auslösen eines Menüpunktes den Aufbau weiterer Menüs (Untermenüs) bewirkt hat.

Wird - *bei leerem Suchstring-Puffer* - in einem Untermenü ein Zeichen eingegeben, welches keinem Menüpunkt dieses Menüs zugeordnet werden kann, so durchsucht *FIX* aufsteigend vorangegangene Menüs, die auf dem Pfad zu diesem Menü liegen, nach einem passenden Eintrag. Wird kein solcher Menüpunkt gefunden, so erfolgt ein Warnton. Wird ein passender Menüpunkt ausfindig gemacht, so wird dieser zum aktuellen Menüpunkt. Um zu diesem zu gelangen, durchwandert *FIX* alle Menüs zwischen dem Menü, in welchem die Eingabe erfolgt ist, bis zu dem, in welchem der lokalisierte Menüpunkt liegt, wobei alle bis auf Letzteres beendet werden.[2] Anschließend wird wie bei **1.** fortgefahren.

zu **3.**

Der Suchstring-Puffer für die Anwahl wird gelöscht und es erfolgt ein Warnton.

Bei einem nicht alphanumerischen Zeichen wird der Suchstring-Puffer gelöscht und es erfolgt ein Warnton.

Anwahl durch Klicken mit der linken Maustaste (nur in Verbindung mit Frontend)

Zunächst ermittelt *FIX*, worauf geklickt wurde. Anschließend erreicht das Event BT_LEFT die Anwendungslogik.

Bezüglich der Bestimmung eines Menüpunktes, zu der *FIX* die zusammen mit dem Event vom Frontend übermittelte Position des Mauszeigers heranzieht, muss wie bei der Anwahl durch Zeichen zwischen drei Situationen unterschieden werden:

1. Es wird ein Menüpunkt im aktiven Menü angeklickt.
2. Es wird ein Menüpunkt in einem übergeordneten Menü angeklickt.
3. Es wird kein Menüpunkt eines Menüs auf dem Pfad angeklickt.

Das weitere Verhalten entspricht dem bei der Anwahl durch Zeichen. Insbesondere wird bei einem Menü mit der Eigenschaft AUTOSTART der angewählte Menüpunkt auch ausgelöst.

---

1. Ein Menüpunkt gilt als passend, wenn
a) er mit dem augenblicklichen Suchstring beginnt (ob dabei zwischen Groß- und Kleinschreibung unterschieden wird, hängt von der Menüeigenschaft IGNORECASE ab)
und
b) sofern das Menü nicht die Eigenschaft AUTOSTART besitzt, er nicht verborgen ist.

Passen mehrere Menüpunkt im selben Menü, entscheidet sich *FIX* für den mit der kleinsten Position.

2. Um in der Anwendungslogik festzustellen, ob ein Event im Zuge eines “Rücksprungs” auftritt, kann die Funktion fx_men_jmp() benutzt werden. Während eines Rücksprungs wird das Ergebnis der Aufrufe der Anwendungslogik von *FIX* nicht ausgewertet.

Sondertasten zur Anwahl eines Menüpunktes

In diesem Fall wird die Anwendungslogik mit dem entsprechenden Tasten-Event aufgerufen. Bei noch unverändertem aktuellem Menüpunkt erfolgt ein Aufruf der Anwendungslogik mit dem Event L_LEAVE_ENTRY. Der Taste entsprechend wird dann ein nächster Menüpunkt lokalisiert:

- K_kd: Es wird auf den nächstfolgenden (nicht verborgenen) Menüpunkt positioniert; ist der aktive Menüpunkt der letzte (nicht verborgene) Punkt im Menü, so wird auf den ersten (nicht verborgenen) Menüpunkt positioniert.
- K_kr: Es wird auf den nächstfolgenden (nicht verborgenen) Menüpunkt mit gleicher horizontaler Ausrichtung positioniert, sofern ein solcher existiert.
- K_ku: Es wird auf den nächstvorangehenden (nicht verborgenen) Menüpunkt positioniert; ist der aktive Menüpunkt der erste (nicht verborgene) Punkt im Menü, so wird auf den letzten (nicht verborgenen) Menüpunkt positioniert.
- K_kl: Es wird auf den nächstvorangehenden (nicht verborgenen) Menüpunkt mit gleicher horizontaler Ausrichtung positioniert, sofern ein solcher existiert.
- K_PD: Es wird auf den letzten (nicht verborgenen) Menüpunkt positioniert.
- K_PU, K_kh: Es wird auf den ersten (nicht verborgenen) Menüpunkt positioniert.

Nachdem dieser zum aktuellen Menüpunkt gemacht wurde, erfolgt ein Aufruf der Anwendungslogik mit dem Event L_ENTER_ENTRY.

Sondertasten zum Auslösen eines Menüpunktes

Das explizite Auslösen der mit einem aktuellen Menüpunkt verbundenen Aktion erfolgt durch die Eingabe von RT. Das Tasten-Event K_RT wird von *FIX* nicht an die Anwendungslogik übergeben.

Beim (expliziten oder impliziten) Auslösen eines Menüpunktes wird der Anwendungslogik das Event L_BEFORE_EXECUTION übergeben. Die mit dem Menüpunkt verbundene Aktion wird ausgeführt und anschließend das Event L_AFTER_EXECUTION an die Anwendungslogik übergeben.

Sondertasten zum Beenden eines Menüs

Explizit beendet wird die Bearbeitung eines Menüs mittels der Tasten EN oder ST. Bei einem Untermenü führt EN zurück ins direkt übergeordnete Menü, ST ins Hauptmenü. Ein Hauptmenü wird nur durch EN verlassen; hier hat ST keine Wirkung.

EN

Das Tasten-Event K_EN wird von *FIX nicht* an die Anwendungslogik übergeben. Vielmehr wird von *FIX* folgende Event-Reihe generiert: zunächst wird mitgeteilt, dass das Menü verlassen werden soll (L_QUIT_OBJECT), dann wird der aktuelle Menüpunkt verlassen (L_LEAVE_ENTRY), anschließend das aktuelle Menü (L_LEAVE_OBJECT).

Bei einem Hauptmenü wird nun die Bearbeitung beendet, d.h. die Funktion perform() kehrt zurück.

Handelt es sich um ein Untermenü, wird wieder das übergeordnete Menü aktiv. Darin wird als nächstes L_AFTER_EXECUTION generiert.

ST

Der Anwendungslogik wird das Tasten-Event K_ST übergeben.

Handelt es sich bei dem Menü um ein Untermenü, erfolgt danach ein Rücksprung ins Hauptmenü. Hierbei wird zunächst die gleiche Event-Reihe wie bei EN generiert. Bei allen weiteren auf dem Rückweg zum Hauptmenü durchwanderten Menüs werden der Anwendungslogik die Events L_AFTER_EXECUTION, L_QUIT_OBJECT, L_LEAVE_ENTRY, L_LEAVE_OBJECT mitgeteilt, bevor das Menü verlassen wird. Ins Hauptmenü zurückgekehrt, wird abschließend L_AFTER_EXECUTION generiert.

Sonstige Sondertasten

Der Anwendungslogik wird das entsprechende Tasten-Event übergeben.

Die Tasten bewirken folgende Reaktion:

- K_sh: es wird in die Shell verzweigt (siehe aber Programmschalter -sh).
- K_HP: der Hilfetext zum aktuellen Menüpunkt wird angezeigt.
- K_g1: der Bildschirm wird neu aufgebaut.
- K_g5: die Tastenhilfe wird angezeigt.
- K_g8 (nur im Entwicklermodus): der Hilfetext zum aktuellen Menüpunkt kann editiert werden.
- Alle sonstigen Tasten führen zu einen Warnton.

2 Eingriffe in die Menübearbeitung

Der Entwickler kann mit einem Event auch eigene Aktionen verknüpfen, die eine Zustandsänderung herbeiführen. Im folgenden ist eine Anwendungslogik folgender Struktur zugrunde gelegt:

```
Event menue_event_control(obj *objp, Event event)
{
    menue *menp = (menue *)objp;
    int item_nr;
    Event rc;                   /* Event, das an FIX zurueckgegeben wird */

    item_nr = CURRENTMENITEM(menp);
    rc = event;                 /* Initialisieren der Variablen */

    switch (event) {
    /*
     * Verarbeiten der logischen Events
     */
    …
    case L_ENTER_OBJECT :
        break;
    case L_ENTER_ENTRY :
        break;
    …
    case L_LEAVE_OBJECT :
        break;

    /*
     * Verarbeitung von Sondertasten und Zeichen
     */
    default :
        break;
    }
    return(rc);
}
```

Das von der Anwendungslogik zurückgegebene Event ist für den weiteren Ablauf generell nur dann von Bedeutung, wenn der Aufruf nicht im Verlauf eines Rücksprungs aus dem Menü erfolgt.[1] Ob und wie der Rückgabewert das Verhalten von *FIX* beeinflusst, ist abhängig sowohl von dem Event *InEv*, mit dem die Anwendungslogik aufgerufen wurde, als auch dem zurückgegebenen Event *OutEv*.

Zum einen prüft *FIX* die "Zulässigkeit" von *OutEv* bezüglich *InEv*. Bei unzulässigem Event *OutEv* verhält sich *FIX*, als sei *InEv* zurückgegeben worden.[2]

Zum anderen vergleicht *FIX OutEv* (oder bei Unzulässigkeit das ersatzweise benutzte Event) mit *InEv* und ruft in gewissen Fällen, anstatt *OutEv* direkt zu behandeln, erneut die Anwendungslogik mit *OutEv* auf.

Die folgende Tabelle gibt die Behandlung wieder. Die Einträge haben folgende Bedeutung:

- *: Das von der Anwendungslogik zurückgegebene Event ist ohne Bedeutung, weil *FIX* es ignoriert.
- 0: Die Event-Kombination ist unzulässig. *FIX* benutzt ersatzweise *InEv*.
- 1: *FIX* behandelt *OutEv* (Die Reaktion kann allerdings auch in einem akustischen Signal bestehen, wenn *OutEv* für *FIX* keine sinnvolle Bedeutung hat).
- 2: Der Anwendungslogik wird erneut ein Event angeboten, diesmal *OutEv*.

Jene Events, die evtl. auch im Verlauf eines Rücksprungs angeboten werden, sind grau hinterlegt.

OutEv \ *InEv*	L_ENTER_OBJECT	L_ENTER_ENTRY	L_BEFORE_EXECUTION	L_AFTER_EXECUTION	L_LEAVE_ENTRY	L_LEAVE_OBJECT [1]	L_QUIT_OBJECT	unterst.Sondertaste	nicht unterst. Sondertaste	BT_LEFT	Zeichen
L_ENTER_OBJECT	*	0	0	0	0	*	0	0	0	0	0
L_ENTER_ENTRY	*	1	0	0	2	*	0	0	0	0	0
L_BEFORE_EXECUTION	*	2	1	0	0	*	0	2	2	2	2
L_AFTER_EXECUTION	*	0	2	1	0	*	0	0	0	0	0
L_LEAVE_ENTRY	*	0	0	0	1	*	0	0	0	0	0
L_LEAVE_OBJECT	*	0	0	0	0	*	0	0	0	0	0
L_QUIT_OBJECT [2]	*	2	2	2	2	*	1	2	2	2	2
unterst. Sondertaste [2]	*	1	1	1	0	*	1	1	1	1	1
nicht unterst. Sondertaste [2,3]	*	1	1	1	0	*	1	1	1	1	1
BT_LEFT	*	0	0	0	0	*	0	0	0	1	0
Zeichen	*	0	0	1	0	*	0	0	0	0	1
L_STAY [2]	*	1	1	1	1	*	1	1	1	1	1

Anmerkungen:

[1] L_LEAVE_OBJECT tritt nur im Verlauf eines Rücksprungs auf.

[2] Die Reaktion auf ein derartiges Event löscht den Suchstring-Puffer.

[3] Die Reaktion besteht, sofern *OutEv* zulässig ist, einfach aus einem akustischen Signal.

1. Ob dies der Fall ist, kann die Anwendungslogik durch Aufruf der Funktion fx_men_jmp() prüfen.

2. Obwohl die Rückgabe von NOTHING z.Z. aus Kompatibilitätserwägungen möglich ist, sollte dies bei der Erstellung neuer Funktionen vermieden werden. NOTHING wird wie ein unzulässiges Event behandelt.

Im folgenden wird beschrieben, wie *FIX* bei den mit ∗ oder 1 markierten Fällen verfährt.

L_ENTER_OBJECT

L_ENTER_OBJECT als Rückgabewert wird stets verworfen.

Ein Aufruf der Anwendungslogik mit L_ENTER_OBJECT erfolgt beim Aktivieren eines Menüs. Zu diesem Zeitpunkt ist der aktuelle Menüpunkt noch unbestimmt. Erst nachfolgend bestimmt *FIX* den anfangs zu betretenden Menüpunkt, wechselt zu diesem und bietet im nächsten Zyklus das Event L_ENTER_ENTRY an.

L_ENTER_ENTRY

signalisiert *FIX* lediglich das Betreten eines Menüpunktes; nachfolgend beginnt der nächste Zyklus.

L_BEFORE_EXECUTION

Dieses Event signalisiert *FIX*, dass die mit dem aktuellen Menüpunkt verbundene Aktion auszuführen ist. *FIX* führt die Aktion aus und bietet der Anwendungslogik im nächsten Zyklus das Event L_AFTER_EXECUTION an.

L_AFTER_EXECUTION

Dieses Event veranlasst *FIX*, die mit dem Menüpunkt verbundene Aktion als abgeschlossen zu betrachten. *FIX* wartet im nächsten Zyklus auf Eingabe (→ dcgetch()).

L_LEAVE_ENTRY

Die Behandlung von L_LEAVE_ENTRY resultiert zwangsläufig aus einem Aufruf der Anwendungslogik mit L_LEAVE_ENTRY (vgl vorstehende Tabelle). Ein solcher Aufruf ist aber stets Folge eines eingeleiteten Menüpunktwechsels (vgl. Seite 173).

Dieser wird nun ausgeführt und der Anwendungslogik als nächstes behandelbares Event L_ENTER_ENTRY oder L_LEAVE_OBJECT angeboten.

L_LEAVE_OBJECT

L_LEAVE_OBJECT als Rückgabewert wird stets verworfen.

Ein Aufruf der Anwendungslogik mit L_LEAVE_OBJECT erfolgt nur im Zuge eines Rücksprungs. *FIX* deaktiviert das aktuelle Menü.

L_QUIT_OBJECT

Handelt es sich bei dem aktiven Menü um ein Untermenü, so wird ein Rücksprung ins übergeordnete Menü bewirkt. *FIX* übermittelt der Anwendungslogik eine Reihe von Events, ohne deren Ergebnis zu berücksichtigen. Generiert werden:

- L_LEAVE_ENTRY (anschließend wird der aktuelle Menüpunkt verlassen)
- L_LEAVE_OBJECT (anschließend wird zum übergeordneten Menü zurückgekehrt; darin aktuell ist der Menüpunkt, von dem aus das Untermenü ausgelöst wurde)

Der Rücksprung ist beendet. Im nunmehr aktiven Menü bietet *FIX* im nächsten Zyklus L_AFTER_EXECUTION an.

Wird L_QUIT_OBJECT in einem Hauptmenü zurückgegeben, übermittelt *FIX* der Anwendungslogik ebenfalls eine Reihe von Events, ohne deren Ergebnis zu berücksichtigen. Generiert werden:

- L_LEAVE_ENTRY (anschließend wird der aktuelle Menüpunkt verlassen)
- L_LEAVE_OBJECT (anschließend beendet *FIX* die Menübearbeitung)

L_STAY

FIX beginnt unmittelbar den nächsten Zyklus und wartet darin auf Eingabe.

Tasten-Events K_ku, K_kd, K_kl, K_kr, K_PD, K_PU, K_kh

FIX leitet einen Menüpunktwechsel ein (vgl. Seite 169) und ruft als ersten Schritt die Anwendungslogik mit dem Event L_LEAVE_ENTRY auf.

Tasten-Events K_sh, K_HP, K_g1, K_g5, K_g8

FIX startet die entsprechende Aktion (vgl. Seite 170); im nächsten Zyklus wartet *FIX* auf Eingabe.

Tasten-Event K_ST

Handelt es sich bei dem aktiven Menü um ein Untermenü, so erfolgt ein Rücksprung ins Hauptmenü. *FIX* übermittelt der Anwendungslogik eine Reihe von Events, ohne deren Ergebnis zu berücksichtigen:

- L_QUIT_OBJECT
- L_LEAVE_ENTRY (anschließend wird der aktuelle Menüpunkt verlassen)
- L_LEAVE_OBJECT (anschließend wird zum übergeordneten Menü zurückgekehrt; darin aktuell ist der Menüpunkt, von dem aus das Untermenü ausgelöst wurde)

und, im nunmehr aktuellen Menü,

- L_AFTER_EXECUTION.

Diese Sequenz wird für jedes durchwanderte Menü wiederholt, bis das Hauptmenü aktiv wird. Der Rücksprung gilt erst als abgeschlossen zu dem Zeitpunkt, zu dem im Hauptmenü der Aufruf mit L_AFTER_EXECUTION erfolgt.

Wird K_ST im Hauptmenü zurückgegeben, wartet *FIX* im nächsten Zyklus auf Eingabe.

alphanumerisches Zeichen

FIX fügt das Zeichen an den aktuellen Suchstring-Puffer an und versucht im aktuellen oder, bei einem Suchstring der Länge 1, auch in den übergeordneten Menüs einen Menüpunkt zu finden, der mit dem Inhalt des Suchstrings beginnt. Ob dabei zwischen Groß- und Kleinschrift unterschieden wird, hängt von den Menüeigenschaften ab.

Wird im aktuellen Menü ein passender Eintrag gefunden, so übermittelt *FIX* der Anwendungslogik eine Reihe von Events, ohne ihr Ergebnis zu berücksichtigen. Generiert werden:

- L_LEAVE_ENTRY (anschließend wird zum ermittelten Menüpunkt gewechselt)
- L_ENTER_ENTRY (hier ist der Rückgabewert der Anwendungslogik wieder relevant)

und, sofern das aktive Menü die Eigenschaft AUTOSTART besitzt und die Anwendungslogik nicht eingegriffen hat,

- L_BEFORE_EXECUTION

Wird ein Eintrag in einem übergeordneten Menü gefunden, so beginnt ein Rücksprung. *FIX* übermittelt der Anwendungslogik eine Reihe von Events, ohne ihr Ergebnis zu berücksichtigen:

- L_QUIT_OBJECT
- L_LEAVE_ENTRY (anschließend wird der aktuelle Menüpunkt verlassen)
- L_LEAVE_OBJECT (anschließend wird zum übergeordneten Menü zurückgekehrt; darin aktuell ist der Menüpunkt, von dem aus das Untermenü ausgelöst wurde)

und, im nunmehr aktuellen Menü,

- L_AFTER_EXECUTION

Diese Sequenz wird für jedes durchwanderte Menü wiederholt, bis dasjenige Menü aktiv wird, worin der Menüpunkt gefunden wurde.

Schließlich erfolgen weitere Aufrufe der Anwendungslogik mit den Events

- L_LEAVE_ENTRY (anschließend wird zum ermittelten Menüpunkt gewechselt)
- L_ENTER_ENTRY (hier ist der Rückgabewert der Anwendungslogik wieder relevant)

und, sofern das nun aktive Menü die Eigenschaft AUTOSTART besitzt und die Anwendungslogik nicht eingegriffen hat,

- L_BEFORE_EXECUTION.

Event BT_LEFT

FIX versucht im aktuellen oder den übergeordneten Menüs einen Menüpunkt zu finden, dessen Text an der Position des Mauszeigers dargestellt ist. Das weitere Vorgehen entspricht dem bei alphanumerischen Zeichen.

sonstiges Sondertasten-Event,
nicht alphanumerisches Zeichen

Es erfolgt ein Warnton; im nächsten Zyklus wartet *FIX* auf Eingabe.

Der Rest dieses Abschnitts zeigt exemplarisch einige Anwendungen und ein Protokoll, welches die Event-Reihen aufzeigt. Die hier gezeigten Protokolle sind Fragmente des Logfiles, welches sich der *FIX*-Entwickler erzeugen lassen kann (siehe “Event-Testhilfe” auf Seite 529).

Der Aufbau einer Zeile hat folgende Form:

```
<label>    <Menüname>/<Menüpunkt-Position>          Event (textuell) [ Event (Code) ]
```

wobei

```
<label>              ::= ← l      - an die Anwendungslogik übermitteltes logisches Event
                     |   l →      - von der Anwendungslogik zurückgegebenes logisches Event
                     |   ← c      - an die Anwendungslogik übermitteltes Zeichen
                     |   c →      - von der Anwendungslogik zurückgegebenes Zeichen
                     |   ← k      - an die Anwendungslogik übermittelte Sondertaste
                     |   k →      - von der Anwendungslogik zurückgegebene Sondertaste
                     |   ← g      - Event beim Durchlaufen eines Menüs
<Menüname>           ::=    Name des Menüs
<Menüpunkt-Position> ::=    Position des Menüpunktes (beginnend bei 0)
```

Beispiel

Beim Betätigen der Taste kd soll die mit dem Menüpunkt verbundene Aktion ausgeführt werden, sofern der Anwender hierzu berechtigt ist.

Anwendungslogik:

```
…
case K_kd :
    rc = L_BEFORE_EXECUTION;
    break;
…
case L_BEFORE_EXECUTION :
    if (! CheckPermission())
        rc = L_AFTER_EXECUTION;
    break;
```

```
…
```

Protokoll:

- Annahme: CheckPermission() liefert TRUE

```
← k    MEN1/0   K_kd […]
l →    MEN1/0   L_BEFORE_EXECUTION […]
← l    MEN1/0   L_BEFORE_EXECUTION […]          vgl. Zeile 3, Spalte 8 der Matrix
l →    MEN1/0   L_BEFORE_EXECUTION […]
/*
 * die Aktion wir ausgeführt
 */
← l    MEN1/0   L_AFTER_EXECUTION […]
l →    MEN1/0   L_AFTER_EXECUTION […]
```

- Annahme: CheckPermission() liefert FALSE

```
← k    MEN1/0   K_kd […]
l →    MEN1/0   L_BEFORE_EXECUTION […]
← l    MEN1/0   L_BEFORE_EXECUTION […]
l →    MEN1/0   L_AFTER_EXECUTION […]
/*
 * Die Ausführung der Aktion wird unterbunden.
 */
← l    MEN1/0   L_AFTER_EXECUTION […]           vgl. Zeile 4, Spalte 3 der Matrix
l →    MEN1/0   L_AFTER_EXECUTION […]
```

Beispiel

Beim Betätigen der Taste f5 soll ein Taschenrechner eingeblendet werden.

Anwendungslogik:

```
…
default :
    if (event == K_f5) {
        calculator();
        rc = L_STAY;
            /*
             * unterdrückt Reaktion von FIX (Warnton).
             */
    }
    break;
…
```

Protokoll:

```
← k    MEN1/0   K_f5 […]
l →    MEN1/0   L_STAY […]
```

Beispiel

Die Funktionalität der Tasten kr und kl soll wie folgt erweitert werden:

- Wird im letzten Menüpunkt kr betätigt, so soll auf den ersten Menüpunkt positioniert werden.
- Wird im ersten Menüpunkt kl betätigt, so soll auf den letzten Menüpunkt positioniert werden.

Anwendungslogik:

```
…
#define UNDEFINED     0
…
    static short lastkey = UNDEFINED;
    …
    case L_LEAVE_ENTRY :
        /*
         * das Positionieren wird in den beiden oben beschriebenen Fällen nicht FIX
         * ueberlassen, sondern sofort hier vollzogen
         */
        if (item_nr == 0 && lastkey == K_kl) {
                men_moveto(menp, LASTMENITEM(menp));
                rc = L_ENTER_ENTRY;
        }
        else if (itemnr == LASTMENITEM(menp) && lastkey == K_kr) {
                men_moveto(menp, 0);
            rc = L_ENTER_ENTRY;
        }
        lastkey = UNDEFINED;
        break;
    …
    case K_kr :
    case K_kl :
        lastkey = event;
        /*
         * die Positionierung erfolgt nicht direkt, um eine korrekte L_ENTER_ENTRY -
         * L_LEAVE_ENTRY-Klammerung zu bewahren
         */
        break;
    …
```

Protokoll:

- Annahme: im ersten Menüpunkt wird die Taste kl betätigt

```
← k    MEN1/0    K_kl [...]
k →    MEN1/0    K_kl [...]
← l    MEN1/0    L_LEAVE_ENTRY [...]
/*
 * Durch den Aufruf von men_moveto() ist der letzte Menüpunkt (4) nun der aktuelle.
 */
l →    MEN1/4    L_ENTER_ENTRY [...]
← l    MEN1/4    L_ENTER_ENTRY [...]                vgl. Zeile 2, Spalte 5 der Matrix
l →    MEN1/4    L_ENTER_ENTRY [...]
```

- Annahme: im letzten Menüpunkt wird die Taste kr betätigt

```
← k    MEN1/4   K_kr […]
k →    MEN1/4   K_kr […]
← l    MEN1/4   L_LEAVE_ENTRY […]
/*
 * Durch den Aufruf von men_moveto() wird der erste Menüpunkt zum aktuellen.
 */
l →    MEN1/0   L_ENTER_ENTRY […]
← l    MEN1/0   L_ENTER_ENTRY […]                    vgl. Zeile 2, Spalte 5 der Matrix
l →    MEN1/0   L_ENTER_ENTRY […]
```

Beispiel

Beim Verlassen eines Menüs durch die Taste EN sollen bis dahin verwaltete Daten gelöscht werden. Dies soll nicht geschehen, wenn ein Menü durchwandert wird.

Anwendungslogik:

```
…
case L_QUIT_OBJECT :
    if (! fx_men_jmp(menp))
        clear_buffer();
    break;
…
```

Protokoll:

Annahme: Aktueller Menüpunkt sei MEN1/0→MEN11/0 →MEN111/0. Eingegeben wird ein Zeichen, zu dem der dritte Menüpunkt des Hauptmenüs passt. Dorthin zurückgekehrt, wird die Taste EN gedrückt.

```
← c    MEN111/0    ? […]
c →    MEN111/0    ? […]
/*
 *  Der Rücksprung beginnt.
 */
← g    MEN111/0    L_QUIT_OBJECT […]
/*
 *  clear_buffer() wird nicht aufgerufen.
 */
← g    MEN111/0    L_LEAVE_ENTRY […]
← g    MEN111/0    L_LEAVE_OBJECT […]
/*
 * MEN11 wird aktives Objekt.
 */
← g    MEN11/0  L_AFTER_EXECUTION […]
← g    MEN11/0  L_QUIT_OBJECT […]
/*
 *  clear_buffer() wird nicht aufgerufen.
 */
← g    MEN11/0  L_LEAVE_ENTRY […]
← g    MEN11/0  L_LEAVE_OBJECT […]
/*
 * MEN1 wird aktives Objekt.
 */
← g    MEN1/0   L_AFTER_EXECUTION […]
```

```
← g     MEN1/0    L_LEAVE_ENTRY […]
/*
 * Es wird zum dritten Menüpunkt gewechselt. Der Rücksprung ist beendet.
 */
← l     MEN1/2    L_ENTER_ENTRY […]
l →     MEN1/2    L_ENTER_ENTRY […]
/*
 *  Die Taste EN wird gelesen.
 */
← l     MEN1/2    L_QUIT_OBJECT […]
/*
 *  clear_buffer() wird aufgerufen.
 */
l →     MEN1/2    L_QUIT_OBJECT […]
/*
 *  Der Sprung aus dem Hauptmenü beginnt.
 */
← g     MEN1/2    L_LEAVE_ENTRY […]
← g     MEN1/2    L_LEAVE_OBJECT […]
```

30 Programmieren mit Masken

1 Binden von Feldern und Variablen

In Fällen, in denen sich die Logik auf das simple Modifizieren einzelner Sätze oder dazu in Beziehung stehender Satzmengen beschränkt, kann *FIX* u.U. den Verkehr mit der Datenbank selbständig abwickeln, sofern die Maske mit entsprechender Information versehen ist (→ perfix()). Hier hält *FIX* die Daten in von ihm selbst verwalteten Speicherflächen. Sobald aber komplexe Datenbankoperationen oder Verarbeitungsschritte erforderlich sind, stößt man an Grenzen.

Datenbankzugriffe und/oder Verarbeitungsschritte müssen dann vom Entwickler im Programmcode formuliert, die Daten in Variablen abgelegt werden.[1] Da Anwendungsprogramme - im Gegensatz zu Tools wie *FIX* oder **dbaccess** - anstelle von dynamischem SQL eher fest kodierte Statements verwenden, findet auch der Austausch von Werten mit dem Datenbanksystem durch speziell gekennzeichnete Variablen der Wirtssprache, so genannte (*Datenbank*-)*Hostvariablen*, statt.

Wie aber werden Maskenfelder und Variablen miteinander verknüpft?

FIX sieht für die Verbindung von Maskenfeldern mit Variablen des Anwendungsprogramms Arrays einer speziellen Datenstruktur vor, die den Feldern der Maske, die zu einem Datenelement in Beziehung stehen, die Adressen der Variablen zuordnen, in denen die Datenelemente abgelegt sind:[2]

```
typedef struct {
    char          *b_fnam;       /* Feldname */
    char          *b_varadr;     /* Zeiger auf Variable */
    unsigned int  b_prg_nr;      /* Element-Bezeichner, vgl. Seite 180 */
    …
} mfn_bind;

mfn_bind bezeichnung[ ] = {
    …
};
```

Die einem Feld zugeordnete Variable wird als *Feld-Hostvariable* bezeichnet. Felddefinition und Variable müssen zumindest bzgl. des Datentyps zueinander passen (vgl. Kapitel *7*). Daneben sind aber auch jene Feldattribute zu berücksichtigen, die Einfluss auf die Darstellbarkeit von Werten haben, wie z.B. die Feldlänge.

Durch Aufruf der Funktion

```
void    mfn_varbind(mfn_bind *bindptr, mask *mskp)
```

werden die in dem mfn_bind-Array *bindptr* definierten Bindungen wirksam.

Technisch geht das so vor sich, dass *FIX* die in der Feld-Datenstruktur eingetragene Adresse, wo die Werte eines Feldes abzulegen sind (Komponente *wrkp*), durch die in *bindptr* spezifizierte Adresse ersetzt.[3]

1. Hierfür kann ihm *FIX* Vorlagen generieren.
2. Diese Datenstruktur kann, sofern die Maske mit entsprechender Information angereichert ist, ebenfalls generiert werden.

Der Wert an der alten Adresse wird an die neue Adresse kopiert. Dies geschieht, da bereits beim Laden einer Maske ihre Felder mit Werten belegt werden. Da die Maskenfelder zu diesem Zeitpunkt noch nicht gebunden sind, werden die Werte zunächst in internen (dynamisch allokierten) Feld-Hostvariablen abgelegt, deren Adresse in der Komponente *wrkp* der Feld-Datenstruktur eingetragen wird.

Feld-Links und mfn_varbind()

Ist in der mfn_bind-Struktur für einen Feld-Link eine Adresse ungleich (char *)0 aufgeführt, so muss diese mit der bestehenden Bindung des Root-Feldes übereinstimmen.

Wird das Root-Feld neu gebunden, so werden auch alle Feld-Links an die neue Adresse gebunden.

2 Der Zugriff auf Maskenelemente im Programm

Die Elemente einer Maske werden in Anwendungsprogrammen i. Allg. auf drei Arten angesprochen:

1. über eine logische Nummer (eine nicht negative ganze Zahl), welche das Element hinsichtlich der Maske eindeutig identifiziert:

   FIX vergibt bei der Generierung eines Programms fortlaufende Nummern an die Maskenelemente und ordnet jeder Nummer eine aussagekräftige define-Konstante, den *Element-Bezeichner*, zu, der auch in der mfn_bind-Struktur (vgl. Seite 179) hinterlegt wird.

   Diese Art des Zugriffs setzt einen Aufruf von mfn_varbind() voraus.

   Verwendet man im Anwendungsprogramm ausschließlich diese Bezeichner, können die Elemente einer Maske, da die Nummer unabhängig von der Position des Elements in der Maske ist, in der Beschreibungsdatei umgeordnet werden, ohne dass eine Änderung im Programmcode erforderlich wird.

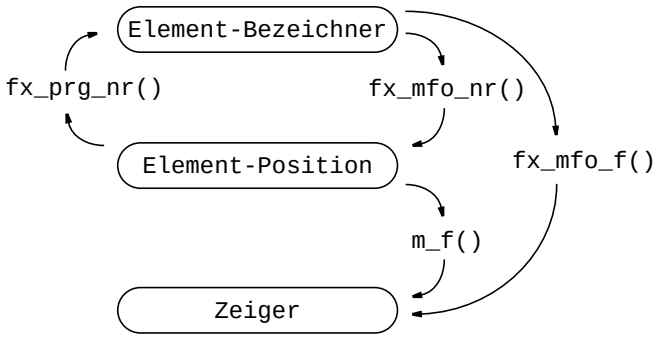
2. über die *Element-Position* in der Beschreibungsdatei, wobei die Nummerierung der Elemente bei 0 beginnt

3. über einen opaken *Zeiger* auf das Element:

   Nur beim Zugriff auf Komponenten des Elements muss, je nach Elementart (ob_class), ein Cast des Zeigers in den korrekten Typ (field *, mask *, submask *, table *) erfolgen.

Anwendungen benutzen bevorzugt die erste Methode. Da der Element-Bezeichner für *FIX*-Funktionen ohne Gehalt ist, erwarten diese meist eine Element-Position oder einen Zeiger. *FIX* stellt daher Funktionen zur Umrechnung bereit:

Abb. 34 Konversion zwischen Element-Bezeichner, Element-Position und Element-Zeiger



3. Bei den Typen FXDTIMETYPE und FXINVTYPE muss der Qualifier der zu bindenden Variable mit dem des Feldes übereinstimmen. Wenn der Qualifier der Variablen gleich 0 ist, vermutet *FIX* eine nicht initialisierte Variable und überträgt den Qualifier des Feldes in die Variable (vgl. Anmerkung auf Seite 150).

Kern einer Anwendung ist der Zugriff auf Daten, ihre Verarbeitung und Ablage. Masken haben hierbei die Aufgabe, die komfortable Bearbeitung der Daten zu ermöglichen, d.h. von allen *FIX*-Objekten besitzen sie bzw. ihre Elemente, die Felder, die engste Verbindung zu den Datenelementen, auf denen die Anwendung operiert.

Ehe auf die Einsatzmöglichkeiten von Masken eingegangen wird, sind zunächst einige detaillierte Bemerkungen zu dem Umgang mit den Daten nötig.

3 Änderung von Feldwerten

Anders als bei der interaktiven Eingabe von Feldwerten durch den Anwender legt *FIX* deren programmgesteuerte Modifikation allein in die Verantwortung des Entwicklers. Der Wert wird in diesem Fall weder auf Übereinstimmung mit den Feldeigenschaften überprüft noch werden die mit dem Feld verknüpften Prüfungen durchgeführt. Einzige Ausnahmen sind die Behandlung von NULL-Werten (vgl. “NULL-Werte” auf Seite 40) und Zuweisungen an Datum-Felder ohne Tageskomponente.

Gewöhnlich wird der Wert von Feldern (f->wrkp) auf eine der folgenden Arten geändert:

1. durch direkte Modifikation des Inhalts der Feld-Hostvariablen, z.B. durch die Ausführung einer SQL-Anweisung, im Anwendungsprogramm:

   Hier kann *FIX* - da nicht beteiligt - die Konsistenz des Wertes mit den Feldeigenschaften nicht sicherstellen; ein nicht zulässiger NULL-Wert wird von *FIX* erst beim nächsten Aufruf von wrktoinfo() geeignet ersetzt.

2. durch Löschen des Feldes (→ ferase()):

   Die Konsistenz des neuen Wertes - dies ist NULL bzw. “ ”/0 - mit den Feldeigenschaften wird von *FIX* nicht geprüft.

3. durch Übernahme eines Wertes von einer anderen Speicherstelle (→ fputval()):

   Die Konsistenz des neuen Wertes mit den Feldeigenschaften wird hier von *FIX* ebenfalls nicht geprüft.

4. durch Ableitung aus einer Zeichenfolge (→ fputstring()):

   Das Argument wird in einen Wert des Feldtyps zu konvertieren versucht, wobei bei allen Typen außer FXCHARTYPE und FXGRAPHICSTYPE ein Leerstring zu NULL konvertiert wird. Argumente, die nicht konvertiert werden können, weist *FIX* zurück, wobei der bisherige Feldwert erhalten bleibt.

5. durch Selo-Auswahl (→ fputselo()):

   Numerische Werte werden zunächst nach dec_t und dann in den Feldtyp zu konvertieren versucht. Schlägt die Konvertierung fehl, bleibt der bisherige Feldwert erhalten.

   Bei den Typen FXDTIMETYPE und FXINVTYPE muss der Qualifier der Spalte mit dem des Feldes übereinstimmen.

6. durch interaktive Erfassung (→ fx_accept()):

   Die Konsistenz des *vorgefundenen* Wertes mit den Feldeigenschaften wird von *FIX* nicht geprüft, ein nicht zulässiger NULL-Wert aber geeignet ersetzt.

   Die Korrektheit der Eingabe des Anwenders wird hingegen teils direkt, teils erst beim Beenden der Felderfassung geprüft. Bei der Eingabe werden die Feldeigenschaften berücksichtigt. Der Anwender kann das Feld erst verlassen, wenn es einen syntaktisch korrekten Wert enthält.

   Ein leer verlassenes Feld ergibt den Wert NULL bzw. “ ”/0.

Mit Ausnahme des ersten Falles sorgt *FIX* anschließend durch einen Aufruf von wrktoinfo() dafür, dass

- nicht zulässige NULL-Werte geeignet ersetzt werden,
- sofern das Feld zu einer Mehrsatz-Maske gehört, der Wert in die zugehörige Spalte des aktuellen Satzes der Memory Relation übertragen wird (zu Sonderfällen siehe Seite 182 und Seite 186),
- sofern das Feld Entscheidungsfeld einer Tabellenmaske mit Zeilentypen ist, der Zeilentyp neu bestimmt wird,
- die Felddarstellung neu erzeugt wird (jedoch nicht unbedingt ausgegeben; vgl. Parameter *display*).

Immer, wenn *FIX* prüft, ob ein NULL-Wert zu ersetzen ist, wird außerdem

- bei einem CHARACTER-Feld ein Wert ungleich NULL ggf. mit Leerzeichen aufgefüllt,
- bei einem Semigrafik-Feld die Anzahl der Zeichen geprüft,
- bei Datum-Feldern ohne Tageskomponente ein Wert ungleich NULL auf den Monatsanfang bzw. das Monatsende gewandelt,
- bei einem Zeitpunkt- oder Zeitspannen-Feld die Übereinstimmung des Qualifiers der Hostvariablen mit dem des Feldes verifiziert.

Das TOUCHED-Attribut (vgl. Seite 210) wird durch die Konversion eines unzulässigen Wertes nicht berührt.

4 Feld-Links

Feld-Links stellen eine praktische Möglichkeit dar, in einer Maske auf den Wert eines Feldes einer anderen Maske zuzugreifen oder ihn zu ändern, erfordern allerdings ein hohes Maß an Sorgfalt. Ein Feld-Link wird als ein Feld definiert, das dem, auf das es Bezug nimmt - dem *Root-Feld* -, in logischer Hinsicht wie dem Wertebereich gleich ist, sich in optischen Eigenschaften aber von ihm unterscheiden kann. Es können mehrere Links auf das gleiche Root-Feld eingerichtet werden. Ein Link auf einen Feld-Link wird auf dessen Root-Feld umgelenkt.

Wird beim Laden einer Maske ein Feld-Link vorgefunden (**link** #*maskenname.feldname*), so muss zu diesem Zeitpunkt (genau) eine Maske mit Namen *maskenname* von der Anwendung geladen sein und diese ein Feld mit Namen *feldname* enthalten. Dabei kann es sich auch um die einbettende Maske handeln, da *FIX* eine Maske bereits katalogisiert, ehe die darin eingebetteten Masken geladen werden.

Die Definitionen von Feld-Link und Root-Feld müssen übereinstimmen bzgl.

- Datentyp
- Art der NULL-Unterstützung
- Format bzw. Länge sowie ggf. Anzahl Nachkommastellen

Die Stimmigkeit weiterer Werteinschränkungen durch Wertvorgabe, reguläre Ausdrücke, DBMUST und Checkroutine liegt in der Verantwortung des Entwicklers.

Feld-Link und Root-Feld teilen sich

- die Feld-Hostvariable (*wrkp*),
- die Eigenschaft TOUCHED

sowie

- den Puffer, der die Literaldarstellung enthält (*info*)[1].

Dies hat zur Folge, dass Zuweisungen an einen Feld-Link (→ fputval(), fputstring(), fputselo(), wrktoinfo()) die gleiche Wirkung haben, als würden sie auf das Root-Feld angewendet; insbesondere verändern sie, wenn das Root-Feld zu einer Mehrsatz-Maske gehört, deren Memory Relation (vgl. Seite 186).

Eine Reihe von Operationen auf einem Feld-Link werden von *FIX* automatisch auf das Root-Feld abgebildet und von dort an alle weiteren Links propagiert, u.a.:

- das Setzen/Wegsetzen der Eigenschaft TOUCHED (→ f_state())
- die Darstellung (→ fdisp())

bzw. bei Feld-Links unterdrückt:

- Restaurieren des Defaultwertes beim Neustart der Maske.

1. Daher erbt z.Z. ein Feld-Link die Eigenschaft ZERO_VALUED von seinem Root-Feld.

Zur Bindung von Feld-Links vgl. Seite 180, zu Feld-Links in Mehrsatz-Masken Seite 186.

5 Videoattribute für Felder

I. Allg. werden Felder ohne spezielle Videoattribute dargestellt und nur während des Besuchs invertiert. Durch das Setzen kontextbezogener Videoattribute ist es aber möglich, Felder besonders hervorzuheben. Insbesondere in Kombination mit der Zuordnung von Farben zu Videoattributen ergeben sich für den Entwickler vielfältige Möglichkeiten zur optischen Bereicherung seiner Anwendung.

Besonderheit hierbei ist, dass pro Feld vier Videoattribute gesetzt werden können (→ f_color()):

F_ACTIVE	Videoattribut während der Felderfassung
F_INACTIVE	Videoattribut, wenn das Feld zum aktuellen Satz gehört, aber nicht im Zustand F_ACTIVE ist.
F_DISPLACED	Videoattribut, wenn das Feld weder im Zustand F_ACTIVE noch im Zustand F_INACTIVE ist. Dieser Fall kann nur bei Tabellenmasken auftreten.
F_NOENT	Videoattribut für Felder mit der Eigenschaft NOENT

Die Standard-Zuordnung von Videoattributen zu den kontextbezogenen Videoattributen ist:

- F_ACTIVE - INVERS
- F_INACTIVE - NORMAL
- F_DISPLACED - NORMAL
- F_NOENT - LOW

6 Dynamische Änderung von Feldformaten

Es ist möglich, das Format eines Feldes in Abhängigkeit von bestimmten Gegebenheiten einzustellen. Für Masken und Rollmasken kann das durch Ersetzen des Formates (f_AssignFormat()) oder durch Entfernen (f_RemoveFormat()) und Wiederherstellen (f_RestoreFormat()) des Formates innerhalb der Anwendungslogik erfolgen.

Bei einer Tabelle ändert sich in diesem Fall jedoch das Format der kompletten Spalte. Um das Format für jede Tabellenzelle einzeln zu bestimmen, muss wie folgt vorgegangen werden:

An der globalen Variablen S_SetFormatHook muss eine Funktion der Anwendung hinterlegt werden. Diese Funktion wird von *FIX* jedesmal aufgerufen, bevor ein Format zur Darstellung eines Feldwertes verwendet wird. Die Anwendung hat somit die Möglichkeit, das Format vorher zu ändern. Dazu sind die Funktionen f_AssignFormat(), f_RemoveFormat() und f_RestoreFormat() zu verwenden. Diese Funktionen besitzen einen Parameter display, der bestimmt, ob ein Feld nach der Formatänderung neu dargestellt werden soll. Dieser Parameter muss beim Aufruf aus der Funktion in S_SetFormatHook mit FALSE besetzt werden, sonst würde auf Grund der Neudarstellung die Funktion in S_SetFormatHook in einer Endlosrekursion aufgerufen.

Bekannte Probleme:

Bei der Verwendung der Funktionen sm_forall() und sm_restore() kann es zu Problemen kommen, wenn diese auf das Format des Feldes zugreifen. *FIX* stellt das Format der Felder vor Aufruf von sm_restore() und vor Aufruf der durch sm_forall() definierten Funktion nicht neu ein. Zugriffe auf das Format liefern damit nicht das Format des betreffenden Satzes, sondern das Format des aktuellen Satzes.

7 Umrechnung von Feldwerten

Zur Darstellung und Erfassung eines Feldwertes kann eine andere Maßeinheit verwendet werden. Dazu muss eine Umrechnung zwischen dargestelltem und erfasstem Wert und dem Wert in der Feld-Hostvariable stattfinden. Zur Definition einer Umrechnungsfunktion ist die globale Variable S_ConvertFieldValue mit einem geeigneten Funktionszeiger zu belegen.

Wenn an der Adresse S_ConvertFieldValue eine Funktion hinterlegt wird, dann wird diese Funktion von *FIX* bei der Felderfassung und bei der Darstellung von Feldwerten aufgerufen. Sie hat die Aufgabe, den abgespeicherten Wert (wrkp) oder den angezeigten Wert (info) umzurechnen. Wird als dir der Wert DIR_INFO_TO_WRK angegeben, dann muss eine Umrechnung des angezeigten Wertes in den abgespeicherten Wert erfolgen. Bei dem Wert DIR_WRK_TO_INFO muss eine Umrechnung in der umgekehrten Richtung erfolgen. Der Originalwert steht in dem Parameter from. Der Datentyp ergibt sich aus dem Datentyp des Feldes (f). An der Adresse to muss der umgerechnete Wert in dem gleichen Datentyp hinterlegt werden. *FIX* stellt an der Adresse to vor Aufruf der Funktion eine dem Datentyp entsprechende Speicherfläche bereit. Das Feld f kann zur Ermittlung weiterer Informationen verwendet werden (ob_props). Die Funktion wird für alle Felder aufgerufen. Soll für ein Feld keine Umrechnung vorgenommen werden, dann kann der Wert an der Adresse to unbesetzt bleiben und mit FALSE zurückgekehrt werden. *FIX* verwendet in diesem Fall den unveränderten Wert an der Adresse from.

8 Tooltips auf Feldern

Wird *FIX/Win* als Frontend benutzt, können auf Feldern Tooltips benutzt werden. *FIX/Win* sendet dazu eine Aufforderung an *FIX*, den Text eines Tooltips für ein Feld an einer bestimmten Bildschirmposition zu senden. Wenn ein Text empfangen wird, dann wird der Tooltip dargestellt. Zur Bestimmung des Textes muss *FIX* feststellen, welches Feld an der übergebenen Position liegt. Für Objekte vom Typ MASK, ROLLMASK und SUBMASK müssen dazu keine Funktionen der Anwendung aufgerufen werden. Bei dem Typ TABLE ist die Bestimmung wesentlich schwieriger, da die Felder einer Tabelle nicht für jede Zeile vorhanden sind. Hat die Tabelle Zeilenvarianten so muss die mit *t_discriminate()* definierte Funktion zur Bestimmung der Zeilenvariante aufgerufen werden, um die Variante der Zeile zu ermitteln, auf die geklickt wurde. Erst dann kann des passende Feld ermittelt werden.

Wenn das Feld, zu dem ein Tooltip dargestellt werden soll, gefunden wurde, dann wird geprüft, ob die Komponente

```
char* tooltiptext
```

einen Wert besitzt. Die Komponente kann von der Anwendung durch Aufruf der Funktion

```
void fset_tooltiptext(field* f, char* text)
```

belegt werden. Wenn sie einen Wert != NULL besitzt, dann wird der Text als Tooltip zu *FIX/Win* gesendet.

Ist das nicht der Fall wird geprüft, ob es sich um ein Feld vom Typ FXCHARTYPE handelt und ob die Länge des Feldinhalts (f->info) größer ist, als die Breite des Feldes (f->displen). In diesem Fall wird der vollständige Feldwert als Text für den Tooltip zu *FIX/Win* gesendet. Handelt es sich dabei um ein Feld einer Tabelle, dann muss der Wert in f->info durch Aufruf von wrktoinfo() für die entsprechende Tabellenzeile vorher erzeugt werden. Das hat zur Folge, dass die Funktion zur Umrechnung (S_ConvertFieldValue) und zur Bestimmung des Formates (S_SetFormatHook), die beide von der Anwendung definiert werden, aufgerufen werden.

In allen anderen Fällen wird kein Text für einen Tooltip an *FIX/Win* gesendet und die Anzeige eines Tooltips bleibt aus.

Tooltips dürfen Newlines enthalten. Dadurch wird das Tooltip-Fenster mehrzeilig dargestellt. Die Breite richtet sich nach dem längsten Teilstück des Textes.

9 Feldbuttons

Mittels dieses Features ist es möglich, rechts neben einem Feld bei der Darstellung durch *FIX/Win* einen Button anzuzeigen. Dazu wird der rechte Feldbegrenzer durch ein anderes Grafikzeichen ersetzt und dahinter noch ein zusätzliches Grafikzeichen gezeichnet. Das erste Grafikzeichen stellt den Feldbegrenzer schmaler als üblich dar, so dass der übrige

Platz zum Zeichnen des ersten Teils des Buttons genutzt wird. Dabei ist zu beachten, dass die Bitmap für dieses Grafikzeichen für die vier Feldzustände NORMAL, AKTIV, SELEKTIERT und INAKTIV (je nach Videoattribut des Feldes - hier ist zu beachten, dass das Videoattribut für einen bestimmten Zustand mit f_color() geändert werden kann.) vorhanden sein muss. Für das zweite Grafikzeichen, das den zweiten Teil des Buttons darstellt, genügt es jedoch, dass die Bitmap nur für das Videoattribut NORMAL vorhanden ist.

9.1 Definition der möglichen Buttons

Neben einem Feld kann ein Button dargestellt werden. Die Grafikzeichen, die für die Buttons verwendet werden, können *FIX* mit der Funktion

```
void defineButtonSet(char* cset)
```

bekannt gemacht werden. Der Prototyp befindet sich in der Headerdatei `proto/mask_pto.h`. Als Parameter bekommt die Funktion einen Zeiger auf beliebig viele Zeichen, von denen jeweils zwei die Grafikzeichen eines Buttons definieren. Unterbleibt ein Aufruf dieser Funktion, dann werden die Zeichen "sScCiI" als Grafikzeichen verwendet.

Beispiel

Als Grafikzeichen für die Buttons neben Feldern sollen die Zeichen "ABCDEF" verwendet werden. Dafür sind folgende Grafikzeichen anzulegen (Das jeweils erste Zeichen eines Paares ist der rechte Feldbegrenzer, also A C E)

- A - Code 65 - Bitmapdateien: 065-0.bmp, 065-1.bmp, 065-2.bmp, 065-3.bmp
- B - Code 66 - Bitmapdateien: 066-0.bmp
- C - Code 67 - Bitmapdateien: 067-0.bmp, 067-1.bmp, 067-2.bmp, 067-3.bmp
- D - Code 68 - Bitmapdateien: 068-0.bmp
- E - Code 69 - Bitmapdateien: 069-0.bmp, 069-1.bmp, 069-2.bmp, 069-3.bmp
- F - Code 70 - Bitmapdateien: 070-0.bmp

9.2 Anzeigen eines Buttons

In der Variablen S_DefineFieldButton kann die Adresse einer selbst erstellten Funktion hinterlegt werden, die beim Zeichnen des Feldes aufgerufen wird und als Parameter den Zeiger auf das Feld und ein Flag bekommt, das besagt ob das Feld gerade aktiv ist. Der Rückgabewert dieser Funktion wird als Index des Buttons ausgewertet. Ist dieser -1, wird kein Button dargestellt. Ansonsten werden die Zeichencodes verwendet, die mittels defineButtonSet() definiert wurden. Für das obige Beispiel wären das für den Index 0 "AB", für 1 "CD" und für 2 "EF".

9.3 Anklicken eines Buttons

Beim Anklicken eines Buttons läuft der gleiche Mechanismus ab, wie beim Anklicken eines Feldes. D.h. die Anwendungslogik wird mit dem Maustasten-Event aufgerufen und kann mittels o_RetrieveButtonClickedOn() (definiert in `proto/obj_pro.h`) den Index des angeklickten Buttons abrufen. An der Adresse des Parameters int* buttonIdx_adr wird der Index des angeklickten Buttons hinterlegt oder -1, wenn das Feld selbst und nicht der Button angeklickt wurde. Aufgrund dieser Information kann in der Anwendungslogik die dem Button zugeordnete Aktion ausgeführt werden.

Wenn mit der Struktur `struct clickinfo` gearbeitet wird, dann kann der Index des Buttons aus der Komponente buttonIdx dieser Struktur gelesen werden.

10 Mehrsatz-Masken

10.1 Wertetransport

Beim Laden einer Mehrsatz-Maske erzeugt *FIX* aus den Feldbeschreibungen eine Memory Relation, die zunächst leer ist und später die Sätze der Maske aufnimmt.

Feld-Links und eingebettete Masken bleiben in der Memory Relation unberücksichtigt. Ihre Werte werden also nicht im Tupel abgelegt (daher erhält bei deren Änderung der aktuelle Satz auch nicht die Eigenschaft TOUCHED, vgl. Seite 211) und sie sind als Argument für satzmengenbezogene Operationen wie sm_sum() oder sm_selfield() nicht verwendbar.

Bei Feldern werden deren Hostvariablen mit den Spalten der Memory Relation gebunden (vgl. Seite 150).

FIX geht i. Allg. davon aus, dass die Werte der Variablen und der Spalten des aktuellen Tupels übereinstimmen. Hierzu überträgt die Funktion wrktoinfo(), die von allen *FIX*-Funktionen gerufen wird, die einem Feld einen Wert zuweisen (vgl. Seite 181), diesen Wert unmittelbar in das aktuelle Tupel der Memory Relation. Hierbei tritt jedoch eine Anomalie auf: auch wenn die Mehrsatz-Maske keine Sätze enthält, haben die Feld-Hostvariablen einen i. Allg. aber ungültigen Wert (dies ist vergleichbar mit Datenbank-Hostvariablen vor dem oder nach einem nicht erfolgreichen Datenbankzugriff).

FIX erlaubt die Zuweisung von Feldwerten, auch ohne dass ein Satz existiert: in diesem Fall werden nur die Feld-Hostvariablen geändert (z.B. bei einem Neustart).

Solange die Maske aktiv ist, wird dieser Zustand allerdings zu vermeiden gesucht. Dazu legt *FIX* automatisch einen Satz an, wenn eine leere Mehrsatz-Maske betreten wird (statt *0 von 0* erscheint in der Satzanzeige *1 von 1*). Tupelspalten und die ihnen entsprechenden Variablen nehmen hierbei die Defaultwerte der Felder an; die Werte von Feld-Links und der Inhalt eingebetteter Masken bleiben hingegen unberührt. In gleicher Weise wird verfahren, wenn die Satzmenge im Verlauf der Bearbeitung leer wird, ohne dass die Maske anschließend verlassen wird.

Umgekehrt löscht *FIX* beim Verlassen der Maske alle *Leersätze*, d.h. solche Sätze, bei denen alle Tupelspalten mit den Defaultwerten der jeweiligen Felder übereinstimmen, sofern der Entwickler nicht besondere Vorkehrungen trifft (→ sm_install_cleanup()).

Da Feld-Links und eingebettete Masken keinen Eingang in die Memory Relation finden und die Nummerierung von Maskenelementen bei 0, die Spaltennummerierung von Memory Relations hingegen bei 1 beginnt, stellt *FIX* Funktionen zur Umrechnung zwischen Element-Position und Spaltennummer zur Verfügung:

```
int  m_fnr_to_colno(mask *mskp, int fnr)
```

liefert die Nummer der Spalte, die dem Element mit Element-Position *fnr* entspricht,

```
int  m_colno_to_fnr(mask *mskp, int colno)
```

liefert die Element-Position des Feldes, zu dem die *colno*-te Spalte gehört.

Benötigt wird die Spaltennummer bei einigen Operationen, die auf Ebene der Memory Relation ablaufen (vgl Beispiel auf Seite 187).

Ein Satzwechsel hinterlässt die Feldeigenschaften i.d.R. unverändert; dies gilt auch bei der Neuanlage eines Satzes als Standardreaktion auf die Events L_ADD und L_INSERT (vgl. Seite 206). Die wichtigste Ausnahme hiervon stellt der Wechsel des Zeilentyps bei Tabellenmasken dar. Es ist Aufgabe der Anwendung, modifizierte Feldeigenschaften ggf. geeignet zurückzusetzen (→ m_restore_properties(), restart(), m_untouch()). Zum Sonderfall TOUCHED vgl. Seite 211.

10.2 Darstellung von Tabellenmasken

Zu Problemen bei der Darstellung von Tabellenmasken am Bildschirm kann es führen, wenn die zugrunde liegende Memory Relation programmgesteuert - beispielsweise durch die Anwendungslogik - manipuliert worden ist. Zu deren Vermeidung geht *FIX* folgendermaßen vor:

- *FIX* vermerkt strukturändernde Operationen wie Austausch und Sortierung der Menge, Einfügen und Löschen von Tupeln in der Memory Relation.
- *FIX* vermerkt eine Positionierung innerhalb der Satzmenge, die die Menge der sichtbaren Sätze verändert, in der Masken-Struktur.
- Bei satzbezogenen Ausgabe-Operationen wird der komplette Maskeninhalt und nicht bloß der aktuelle Satz neu ausgegeben, wenn eine strukturändernde oder scrollende Operation stattgefunden hat, und die entsprechenden Markierungen werden zurückgesetzt.

Dieses Vorgehen bietet gegenüber einer sofortigen Bildschirmanpassung insbesondere dann Vorteile, wenn mehrere derartige Operationen aufeinander folgen.

10.3 Clipboard

Mehrsatz-Masken verfügen über ein *Clipboard*, das automatisch den zuletzt gelöschten Satz aufnimmt. Die Attribute des Satzes (TOUCHED, “alt” etc.), die Werte von Feld-Links und die Sätze eingebetteter Masken werden davon allerdings nicht erfasst. Mittels der Funktion

```
int sm_paste(submask *smsp)
```

kann der aktuelle Satz durch den Inhalt des Clipboards überschrieben werden.

10.4 ROWID

Die ROWID der Tupel einer Memory Relation (vgl. “Memory Relations” auf Seite 149) kann man auch auf Maskenebene nutzen:

sm_rowid(*smsp*)

liefert die ROWID des aktuellen Satzes,

sm_find(*smsp*, BY_ROWID, (char *)*rowid*)

macht den Satz mit ROWID *rowid* zum aktuellen Satz.

Dieser Mechanismus kann beispielsweise anstelle von sm_hold()/sm_restore(), die auf der Position eines Satzes beruhen, benutzt werden, um auch dann zum ursprünglichen Satz zurückzukehren, wenn die Sätze umgeordnet oder andere Sätze gelöscht wurden.

Beispiel:

Die Tabellenmaske *smsp* soll gemäß dem Element an Position *fnr* aufsteigend sortiert werden:

```
/* bestimme Spalte zu Element fnr */
col = m_fnr_to_colno((mask *)smsp, fnr);
/* rette Kennung des aktuellen Satzes, um nach der Sortierung wieder darauf
   positionieren zu koennen */
row = sm_rowid(smsp);
```

```
/* falls zu fnr eine Spalte existiert und (mindestens) ein Satz vorhanden ist ... */
if (col > 0 && row > 0) {
    /* sortiere Memory Relation (erstes Tupel wird zum aktuellen Tupel) */
    mr_qsort(smsp->sm_mr_cb, col, ASCENDING, 0);
    /* gehe zurueck zum urspruenglichen Satz */
    sm_find(smsp, BY_ROWID, (char *)row);
    /* stelle Objekt neu dar (Zeile in Tabelle kann möglicherweise nicht beibehalten
       werden) */
    o_rebuild(smsp);
}
```

10.5 Satz-Historie

FIX unterscheidet “alte” und “neue” Sätze. Erstere sind durch eine *FIX*-eigene Methode zur Datenbeschaffung gelesen worden, Letztere programmgesteuert oder vom Anwender im Zuge der Bearbeitung angelegt.

Zum Erkennen dient die Funktionen

```
BOOLEAN     sm_old(submask *smsp)
```

(liefert TRUE genau dann, wenn der aktuelle Satz “alt” ist) und das Makro

```
sm_new(submask *smsp)
```

(liefert TRUE genau dann, wenn der aktuelle Satz “neu” ist), zum Ändern die Funktionen

```
int  sm_mark_old(submask *smsp)
```

(lässt den aktuellen Satz als “alt” erscheinen) und

```
int  sm_mark_new(submask *smsp)
```

(lässt den aktuellen Satz als “neu” erscheinen).

10.6 Darstellung von Feldern als Tabellenzellen

Durch Verwendung eines anderen Zeichenverfahrens können Tabellen in *FIX/Win* Windows-ähnlicher gezeichnet werden. Die Voraussetzungen dafür sind:

- Alle Spalten der Tabelle stehen in einer Zeile.
- Keine der Spalten besitzt die SPECIAL-Feldeigenschaft.

Da nicht alle Tabellen diese Bedingungen erfüllen, sollten nur bestimmte Tabellen (mit bestimmtem Wert in PrivatePropertyType) mit dieser Methode dargestellt werden.

Es ist notwendig, dass alle Spalten unmittelbar nebeneinander angeordnet sind, damit keine Löcher in der Tabelle entstehen. Um dies zu erreichen, können die Feldlängen nach dem Laden der Maske angepasst werden. Dazu wird die Display-Länge erhöht. Die Größe des Eingabebereichs, der zum Datentyp passt, bleibt erhalten. Die folgende Funktion vergrößert alle Felder einer Tabelle, so dass sie unmittelbar nebeneinander liegen:

```
void expandTableFields(table* tp)
{
    int i,j;
    int next_x;
#define MAX_X 99999
    for (i = 0; i < tp->ms_anz_f; i++) {
        if ( tp->f[i].ob_class != FIELD ) {
```

```
                continue;
            }
            next_x = MAX_X;

            for (j = 0; j < tp->ms_anz_f; j++) {
                if (tp->f[j].s <= tp->f[i].s)
                    continue;
                if (tp->f[j].s < next_x)
                    next_x = tp->f[j].s;
            }
            if (next_x != MAX_X) {
                tp->f[i].displen = next_x - tp->f[i].s - 2;
                if (tp->f[i].f_sep != NULL || tp->f[i].f_choice != NULL )
                tp->f[i].displen -= 1;
            }
        }
    }
```

Sie ermittelt zu einem Feld (tp->f[i]) das am nächsten rechts neben ihm liegende Feld und verwendet dessen Position (next_x) zur Bestimmung der Display-Länge. Dabei werden zwei Zeichen für die Feldbegrenzer abgezogen, die zusätzlich zur Display-Länge gezeichnet werden. Besitzt das Feld eine Choice oder ein Selo, dann wird nochmal eine Position abgezogen, damit die Feldbuttons dargestellt werden können[1].

10.6.1 Darstellen von Tabellenfeldern

Ein Feld hat in *FIX/Win* bezüglich der Darstellung folgenden schematischen Aufbau:

```
[tttt___^^^]
```

Es besteht aus folgenden Teilen:

- einem Grafikzeichen, das als linker Feldbegrenzer verwendet wird: `[`
- dem Text, der den Feldinhalt darstellt: `t`
- Grafikzeichen, die als Füllzeichen für nicht eingegebene Positionen verwendet werden: _
- Grafikzeichen, die als Füllzeichen verwendet werden, wenn die Display-Länge größer ist als die Länge des darstellbaren Textes: ^
- einem Grafikzeichen, das als rechter Feldbegrenzer verwendet wird: `]`

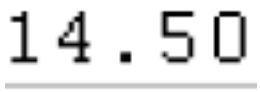
Als Ziel soll eine Tabellen der folgenden Art dargestellt werden:

Art	Einzel	Gesamt
1	4.50	90.00
3	12.49	249.80
4	11.90	238.00
6	14.50	290.00

1. Das ist nur in diesem Beispiel so. In einer Anwendung kann das anders sein.

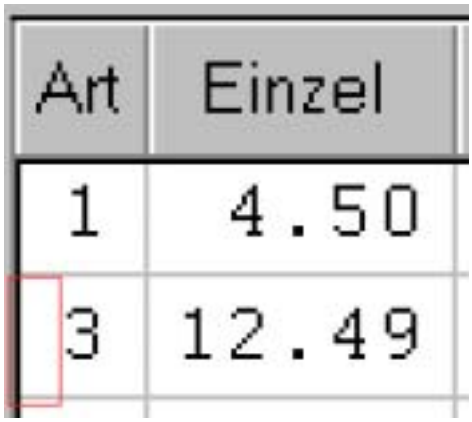
Sie bestehen aus einer eingerahmten Fläche, in der die einzelnen Zellen durch eine hellgraue Linie getrennt werden. Der obere Rand wird durch den Tabellenkopf mit den Überschriften gebildet. Der linke Rand besteht aus einer dunkelgrauen und einer schwarzen Linie. Der rechte und der untere Rand besteht aus einer hellgrauen und einer weißen Linie.

Soll ein Tabellenfeld gezeichnet werden, dann muss für den Text ein anderes Attribut verwendet werden, so dass *FIX/Win* eine andere Darstellung wählt. Statt oben und unten eine Linie in der Hintergrundfarbe (BACKGROUND) und eine Linie in der Feldfarbe (FIELD_LINE) zu zeichnen, wird der Hintergrund komplett in der Hintergrundfarbe des Feldes gezeichnet. Zusätzlich wird am unteren Rand eine hellgraue Linie und eine weiße Linie gezeichnet. Diese beiden Linien bilden die Trennlinie zur darunterliegenden Zeile oder den unteren Rand der Tabelle bei der letzten Tabellenzeile. In diesem Fall ist die weise Linie Bestandteil des unteren Tabellenrandes. In dem anderen Fall ist sie Bestandteil der weißen Fläche des darunterliegenden Feldes.

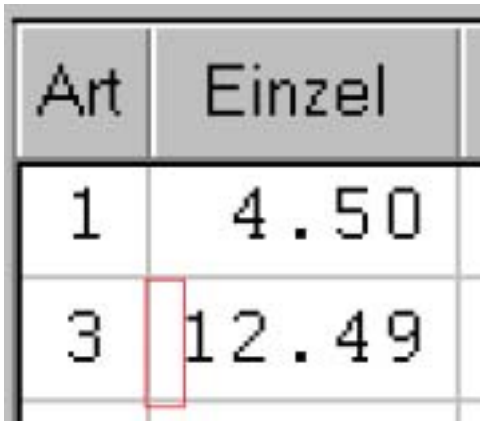


Für die Feldbegrenzer müssen je nach Position des Feldes andere Zeichen verwendet werden:

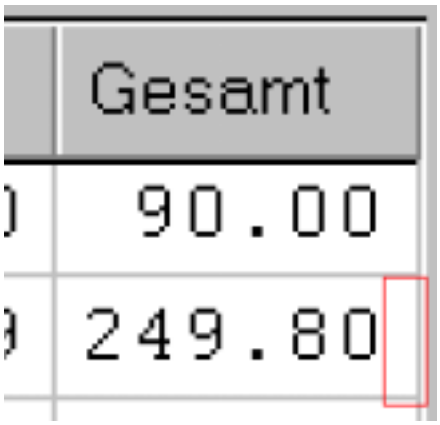
- Der linke Feldbegrenzer des ersten Feldes bildet den linken Rand der Tabelle. Er besteht aus einer vertikalen dunkelgrauen und einer schwarzen Linie, die den Tabellenrand bilden. Zusätzlich enthält er am unteren Rand eine hellgraue Linie, die den Übergang zur unteren Tabellenzelle markiert.



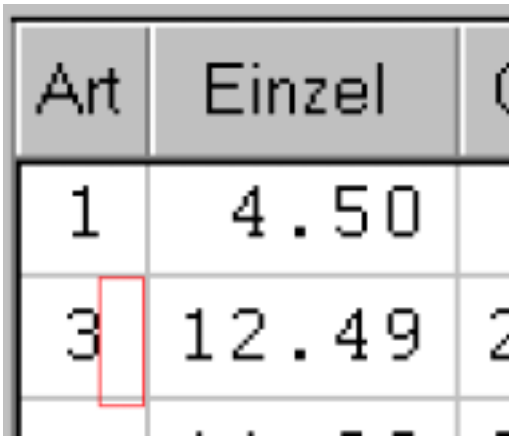
- Der linke Feldbegrenzer aller anderen Felder besteht aus einem Grafikzeichen mit einer hellgrauen vertikalen Linie, die den Übergang zum vorherigen Feld bildet. Auch hier ist die untere hellgraue Linie als Übergang zum darunter liegenden Feld zu zeichnen.



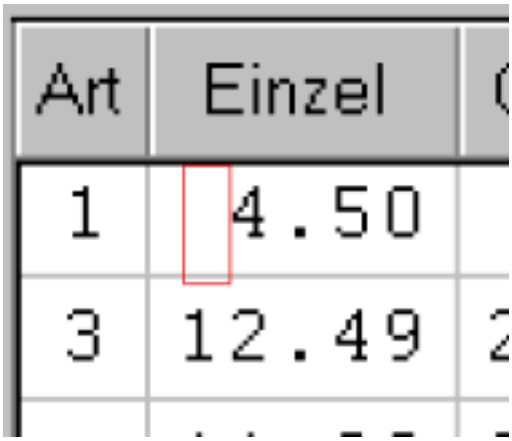
- Der rechte Feldbegrenzer des letzten Feldes bildet den rechten Rand der Tabelle. Er besteht aus einer vertikalen hellgrauen und einer weißen Linie, die den Tabellenrand bilden. Zusätzlich enthält er am unteren Rand eine hellgraue Linie, die den Übergang zur unteren Tabellenzelle markiert.



- Der rechte Feldbegrenzer aller anderen Felder besteht nur aus einer hellgrauen Linie am unteren Rand, die den Übergang zur unteren Tabellenzelle markiert.



- Die beiden Füllzeichen werden auf die gleiche Weise gezeichnet, wie der rechte Feldbegrenzer. Dafür kann das gleiche Grafikzeichen verwendet werden. Um die Flexibilität zu erhalten, Feldfüllzeichen anders darzustellen, wird dafür jedoch ein anderer Code verwendet.



Damit *FIX* weiß, welche Grafikzeichen für Feldbegrenzer und Füllzeichen zu verwenden sind und ob ein Feld als Tabellenfeld dargestellt werden soll, wird vor der Darstellung eines Feldes eine Funktion zur Ermittlung dieser Werte aufgerufen. Die Adresse der Funktion ist in der globalen Variablen

```
BOOLEAN (*S_GetTableFieldCodes)(field *, unsigned char* delim)
```

zu hinterlegen. Sie bekommt als Parameter das darzustellende Feld und einen Zeiger auf einen 5 Zeichen langen Bereich, in dem die Codes für die Grafikzeichen abzulegen sind. Der Bereich hat folgenden Aufbau:

```
[]_^0
```

wobei die Zeichen die folgende Bedeutung haben:

[- linker Feldbegrenzer
] - rechter Feldbegrenzer
_ - Feldfüllzeichen
^ - Feldfüllzeichen für Display-Länge
0 - 0-Byte

Wird der Bereich nicht mit eigenen Codes gefüllt, dann werden die von *FIX* standardmäßig definierten Begrenzer verwendet. Über den Rückgabewert steuert die Funktion, ob das Feld als Tabellenfeld oder als normales Feld gezeichnet werden soll. Ein Wert von TRUE definiert ein Tabellenfeld. Die in delim definierten Codes werden jedoch unabhängig davon immer verwendet. Der folgende Code legt die Zeichencodes für die Tabellenfelder fest:

```
BOOLEAN getTableFieldCodes(field* f, unsigned char* delim)
{
    if (f->ob_parent != 0 && f->ob_parent->ob_class == TABLE && f->ob_class == FIELD) {
        table* tb = (table*)f->ob_parent;
        delim[0] = hasLeftField(tb, f) ? '/' : '{';
        delim[1] = hasRightField(tb, f) ? '!' : '}';
        delim[2] = '°';
        delim[3] = '!';
        return 1;
    }
    return 0;
}
```

Wenn das Feld zu einer Tabelle gehört, dann wird geprüft, ob sich rechts und links neben dem Feld noch ein anderes sichtbares Feld befindet. Je nachdem werden andere Codes für die Feldbegrenzer verwendet.

Zur Überprüfung, ob sich neben einem Feld ein anderes befindet, können die folgenden beiden Funktionen verwendet werden:

```
BOOLEAN hasLeftField(table* tb, field* f)
{
    int i;

    for (i = 0; i < tb->ms_anz_f; i++) {
        if (f_state(&(tb->f[i]), P_GET, (NODISPL)))
            continue;
        if (f->z != tb->f[i].z)
            continue;
        if (f->s > tb->f[i].s)
            return TRUE;
    }
    return FALSE;
}

BOOLEAN hasRightField(table* tb, field* f)
{
    int i;

    for (i = 0; i < tb->ms_anz_f; i++) {
        if (f_state(&(tb->f[i]), P_GET, (NODISPL)))
            continue;
        if (f->z != tb->f[i].z)
            continue;
```

```
            if (f->s < tb->f[i].s)
                return TRUE;
        }
        return FALSE;
    }
```

Damit wird das erste Tabellenfeld auf die folgende Art gezeichnet (`ttt` stellt den Text dar):

{ttt°°°!!!

Ein mittleres Feld wird mit folgenden Codes dargestellt:

/ttt°°°!!!

Das letzte Feld einer Tabellenzeile enthält folgende Codes:

/ttt°°°!!}

Wenn die verwendeten Codes in einer Anwendung bereits belegt sind, dann ist die Funktion abzuändern, so dass andere Codes verwendet werden. Weiterhin ist sie um eine Prüfung zu ergänzen (PrivatePropertyType) die nur bestimmte Tabellen berücksichtigt.

Für *FIX/Win* sind die entsprechenden Bitmaps für alle verwendeten Textattribute bereitzustellen. Weiterhin sind in der Farbzuordnungstabelle die Farbwerte für Tabellenfelder zu definieren.

10.7 Darstellen der Feldbuttons

Für Felder, die ein Selo oder eine Choice besitzen, die über einen Feldbutton geöffnet werden, ist ein Zeichen mehr Platz zu reservieren. Um das Layout der Tabelle nicht zu zerstören, sollte der Feldbutton immer gezeichnet werden (auch dann, wenn das Feld nicht aktiv ist). Es besteht jedoch die Möglichkeit, den Feldbutton aus Zeichen zusammenzusetzen, die aussehen wie Feldfüllzeichen, so dass der Benutzer ihn nicht als solchen erkennen kann.

Im ersten Schritt sollte die Menge der Feldbuttons definiert werden.

```
defineButtonSet("sScCiI§$%&²?");
```

Die ersten drei Buttons (erste 6 Zeichen) sind für normale Felder vorgesehen und entsprechen den von *FIX* definierten Standardwerten. Die anderen drei Buttons sollen für Tabellenfelder verwendet werden. Die in S_DefineFieldButton hinterlegte Funktion sollte folgenden Aufbau haben.

```
int defineFieldButton(field* f, BOOLEAN active)
{
    if ((f->f_sep != NULL || f->f_choice != NULL ) && STYP(NOENT) != NOENT) {
        if (f->ob_parent != 0 && f->ob_parent->ob_class == TABLE) {
            if (active)
                return 3;
            else if (((mask *)f->ob_parent)->ms_dispmode == MS_SHADOW_MODE)
                /* Zeile nicht selektiert */
                return 4;
            else
                return 5;
        } else {
            return 0;
        }
    }
    return -1;
}
```

Damit werden für ein aktives Tabellenfeld die Zeichen "§$" für den Button verwendet. Für diese Zeichen sind Bitmaps bereitzustellen, die aussehen wie ein Button in einer Tabelle. Ein nicht aktives Feld, das sich nicht in der aktuellen Zeile befindet, werden die Zeichen "%&" für den Button verwendet. Dafür sind Bitmaps bereitzustellen, die aussehen wie die Feldfüllzeichen im nicht selektiertem Zustand Liegt das Feld in der aktuellen Zeile, dann werden die Zeichen "²?" verwendet. Für diese Zeichen sind Bitmaps bereitzustellen, die aussehen wie die Feldfüllzeichen im selektiertem Zustand.

Damit beim Anklicken eines Buttons, der aussieht wie ein Feldfüllzeichen, keine Aktion ausgeführt wird, ist beim Eintreten des Events BT_LEFT unbedingt der Index des Buttons (o_RetrieveButtonClickedOn() oder buttonIdx in struct clickinfo) auszuwerten. Dabei ist zu beobachten, dass beim Anklicken des aktiven Buttons der Index 4 ist. Dies liegt daran, dass das Event BT_LEFT die Felderfassung beendet und der Button erst danach ermittelt wird. Vorher erfolgt jedoch die Darstellung des Feldes im nicht aktiven Zustand, wozu der Button mit dem Index 4 verwendet wird.

10.8 Darstellen des Tabellenkopfes

Der Tabellenkopf besteht aus Paintareas des Typs PA_TABLEHEADER. Zur Markierung der Paintareas kann folgende Funktion in einer *FIX*-Anwendung verwendet werden:

```
void demoMarkTableHeaders(table *objp)
{
    struct layout *ptr;
    int x, y , i, len;
    unsigned short *wi_c;
    char label[PA_MAX_TEXTLEN];
    int labelIdx;
    unsigned char label_attr;

    if (!objp->ms_layout) {
        return;
    }

    ptr = (struct layout *)objp->ms_layout;

    for (i = 0; i < objp->ms_anz_f; i++) {
        if (f_state(&(objp->f[i]), P_GET, (NODISPL)))
            continue;
        x = objp->f[i].s-objp->ob_box.bx_b_x - 1;
        y = objp->f[i].z-objp->ob_box.bx_b_y - objp->tb_screen_row;
        len = objp->f[i].displen + 2;
        if (objp->f[i].f_sep != NULL || objp->f[i].f_choice != NULL )
            len++;
        wi_c = (unsigned short *)ptr->data;
        wi_c += ((y * objp->ob_box.bx_cols) + x);
        labelIdx = 0;
        while(len > 0) {
            label_attr = ATTR(wi_c);
            label[labelIdx] = CHAR(wi_c);
            ATTR(wi_c) |= PAINTAREA;
            labelIdx++;
            wi_c++;
            len--;
        }
        label[labelIdx] = '\0';
        if (i == objp->ms_anz_f - 1 ) {
```

```
                pa_declare_tableheader((obj *)objp, y, x,
                    label, NULL, label, label_attr, PA_ALGN_LEFT, PA_BT_LEFT | PA_BT_RIGHT,
                    PA_TABLEHEADER_LAST, 0L, 0L, 0L, NULL, NULL);
            } else {
                pa_declare_tableheader((obj *)objp, y, x,
                    label, NULL, label, label_attr, PA_ALGN_LEFT, PA_BT_LEFT | PA_BT_RIGHT,
                    PA_TABLEHEADER_FIRST, 0L, 0L, 0L, NULL, NULL);
            }
        }

    }
```

Sie sollte von der in S_pa_mark hinterlegten Funktion aufgerufen werden, wenn diese erkennt, dass es sich um eine Tabelle handelt, die Windows-ähnlich dargestellt werden soll (Auswertung von PrivatePropertyType). Die Funktion geht über alle Felder und sucht den Text, der unmittelbar über dem Feld steht. Als linker Rand wird dabei der Feldbegrenzer gewählt. Deshalb ist es notwendig, von der Feldpostion 1 abzuziehen. Als Länge für die Paintarea wird die Länge des Feldes + 2 (wegen der Feldbegrenzer) gewählt. Besitzt das Feld ein Selo oder eine Choice, wird die Länge wegen des Feldbuttons nochmal um eins erhöht. Anhand der so bestimmten Fläche wird der Text der Paintarea aus dem Layout gelesen und mit dem Attribut PAINTAREA versehen. Bei der Deklaration der Paintarea werden die Werte PA_TABLEHEADER_FIRST und PA_TABLEHEADER_LAST für longval1 verwendet. Die Zeichenroutine erkennt daran, ob es sich um einen Tabellenkopf für eine der ersten Spalten oder für die letzte Spalte handelt.

FIX/Win zeichnet Paintareas vom Typ PA_TABLEHEADER in der folgenden Art und Weise. Für jede Paintarea des Tabellenkopfes wird oben und unten eine schwarze Linie gezeichnet. Die untere Linie bildet den Rand der darunter liegenden Tabelle. Dann wird links und unten eine dunkelgraue Linie gezeichnet. Im nächsten Schritt wird dann noch links und oben eine weiße Linie gezeichnet.

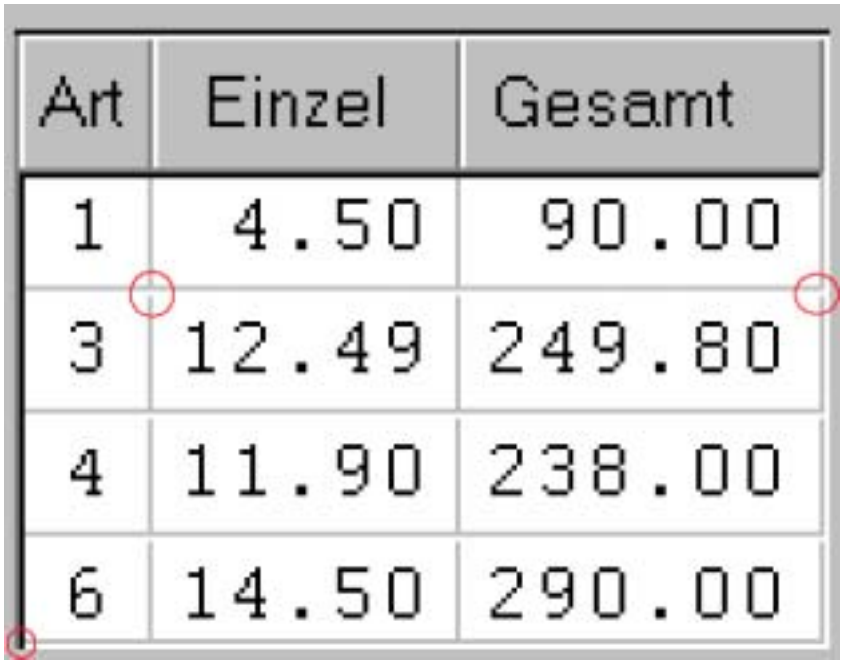
Der rechte Rand wird nur für die letzte Paintarea mit dem Wert PA_TABLEHEADER_LAST gezeichnet. Er besteht aus einer dunkelgrauen und einer weißen Linie. Der rechte Rand der übrigen Tabellenköpfe wird aus dem linken Rand der nächsten Paintarea gebildet.

10.9 Optionen

Damit ist die Tabelle noch nicht ganz perfekt. Bei genauerer Betrachtung sieht man, dass für die unterste Tabellenzeile andere Grafikzeichen verwendet werden müssen, als für die anderen Tabellenzeilen.

Art	Einzel	Gesamt
1	4.50	90.00
3	12.49	249.80
4	11.90	238.00
6	14.50	290.00

Um die Menge der Grafikzeichen einzuschränken kann es jedoch dabei bleiben, beide Zeilen gleich zu zeichnen. Bei dem rechten Feldbegrenzer der mittleren Felder sollte jedoch der unterste graue Punkt der vertikalen Linie entfernt werden, damit am unteren Rand eine durchgezogene Linie entsteht. Die Tabelle hat somit folgendes Aussehen:



Art	Einzel	Gesamt
1	4.50	90.00
3	12.49	249.80
4	11.90	238.00
6	14.50	290.00

Sollen jedoch unterschiedliche Grafikzeichen für die letzte und alle anderen Zeilen verwendet werden, dann ist die Funktion `getTableFieldCodes()` entsprechend anzupassen.

31 Maskenbearbeitung aus der Sicht des Entwicklers

Zur Bearbeitung von Masken durch den Anwender bietet *FIX* die Funktionen perform() und m_present() an. Beide unterstützen eine anwendungsspezifische Logik, wie sie auf Seite 163 beschrieben ist. *FIX* ruft die Anwendungslogik bei einer Vielzahl von Zustandsänderungen auf, die sich aus den Aktionen des Anwenders ergeben und für die Programmlogik relevant sein könnten. Nicht mitgeteilt werden hingegen Vorgänge wie das Eingeben von Zeichen oder das bloße Bewegen der Schreibmarke in einem Feld.

Zustandsänderungen im obigen Sinne sind z.B.:

- eine Maske ist aktiv geworden (L_ENTER_OBJECT)
- ein Wechsel des Datensatzes hat stattgefunden (L_ENTER_RECORD)
- ein Maskenelement steht zur Bearbeitung an (L_ENTERFIELD)
- die Bearbeitung eines Maskenelements ist durch Drücken einer Sondertaste beendet worden (Tasten-Event)
- ein Variantenwechsel hat stattgefunden (L_MSWAP)
- der Datensatz soll gewechselt werden (L_LEAVE_RECORD)
- die Maskenbearbeitung soll beendet werden (L_LEAVE_OBJECT)

Die Anwendungslogik muss die Form

```
Event  ev_control(obj *objp, Event ev)
```

haben. Neben dem Event, das die Art der Änderung angibt, wird ihr beim Aufruf auch das Objekt mitgegeben, auf das es sich bezieht, wodurch dieselbe Funktion auch für mehrere Masken verwendbar wird.

Innerhalb der Anwendungslogik kann der Entwickler vielfältig reagieren:

- Er kann das Event vollständig selbst behandeln und die Standardreaktion von *FIX* unterdrücken.
- Er kann eigene Aktionen mit dem Event verbinden, eine Restbehandlung aber der *FIX*-Logik überlassen; i. Allg. muss die Funktion dann dasjenige Event zurückliefern, auf das *FIX* reagieren soll.
- Er kann die Standardreaktion von *FIX* wirksam werden lassen; dann sollte die Funktion das Event selbst (oder NOTHING) zurückliefern.

Der Entwickler ist weitgehend frei in dem, was er als Reaktion auf ein Event vorsieht, und muss lediglich sicherstellen, dass zu dem Zeitpunkt, an dem *FIX* wieder die Kontrolle übernimmt, ein plausibler Zustand vorliegt: beispielsweise muss das bearbeitete Objekt noch/wieder aktiv und am Bildschirm dargestellt sein. Unter Umständen muss der Entwickler selbst für eine Neuausgabe in Mitleidenschaft gezogener Elemente sorgen, etwa, wenn er im Hintergrund dargestellte Objekte manipuliert hat.

FIX unterstützt zwei Arten, Objekt und Logik miteinander in Verbindung zu bringen:

1. Die Funktion kann als Attribut an die Maske “geheftet” werden (vgl. Seite 163).
2. Die Funktion kann beim Aufruf von perform() und m_present() als Parameter mitgegeben werden.

Eine auf die erste Art bestimmte Anwendungslogik hat Vorrang.

Wird keine Anwendungslogik vorgegeben ((Event (*)(obj *, Event))0), verwendet *FIX* eine eingebaute Logik, die stets die Standardreaktion von *FIX* wirksam werden lässt.

1 perform()

perform() bietet die allgemeinste Form der Maskenbearbeitung an. Die Felder eines Satzes können betreten und eingebettete Masken bearbeitet werden.

Bei einer Maske mit Varianten im Restart-Modus (vgl. Seite 91) wird beim Aufruf von perform() zunächst zur Hauptmaske zurückgekehrt (→ m_setvariant()). Anschließend lässt *FIX* die Maske am Bildschirm erscheinen (→ o_appear()) bzw. bringt sie in den Vordergrund (→ foreground()) und ersetzt den vorhandenen Prompttext durch den der Maske. Die Maske erhält für die Dauer der Bearbeitung die Eigenschaft ACTIVE, die erkennen lässt, dass die Maske in Bearbeitung ist. Die für die Bearbeitung der Maske gültige Anwendungslogik wird bestimmt und der Turbomodus deaktiviert (siehe “Besuch eines Feldes” auf Seite 208).

Wo nicht ausdrücklich ausgenommen, kann die Anwendungslogik davon ausgehen, dass die globale Variable

```
short  *_fnr;
```

auf eine Speicherstelle zeigt, deren Wert das aktuelle Maskenelement mittels seiner Element-Position bestimmt (== &*mskp*->fnr), und dass der Zeiger

```
field  *_f;
```

auf das aktuelle Maskenelement zeigt.[1]

FIX erwartet, dass während des interaktiven Teils der Maskenbearbeitung mittels perform() stets mindestens ein Satz vorhanden ist. Um dies sicherzustellen, legt perform() bei einer Mehrsatz-Maske ohne Sätze zunächst einen Leersatz an, der zum aktuellen Satz wird und ausgegeben wird.

Erst dann wird die Anwendungslogik mit dem Event L_ENTER_OBJECT gerufen (Achtung: hierbei sind *_fnr und _f noch undefiniert). Gibt die Anwendungslogik L_END zurück, wird der Besuchsmodus (vgl. Abb. 35) übersprungen; Ergebnis der Maskenbearbeitung ist L_PRVFIELD.

Bei jedem anderen Ergebnis wird jetzt das Element (des aktuellen Satzes) zum aktuellen Element, dessen Element-Position in der Komponente *mskp*->ms_startfield vermerkt ist, sofern nicht die Anwendungslogik bereits bei der Behandlung von L_ENTER_OBJECT durch Zuweisung an *_fnr ein erstes aktuelles Element bestimmt hat. Alle Felder der Maske verlieren die Eigenschaft TOUCHED.

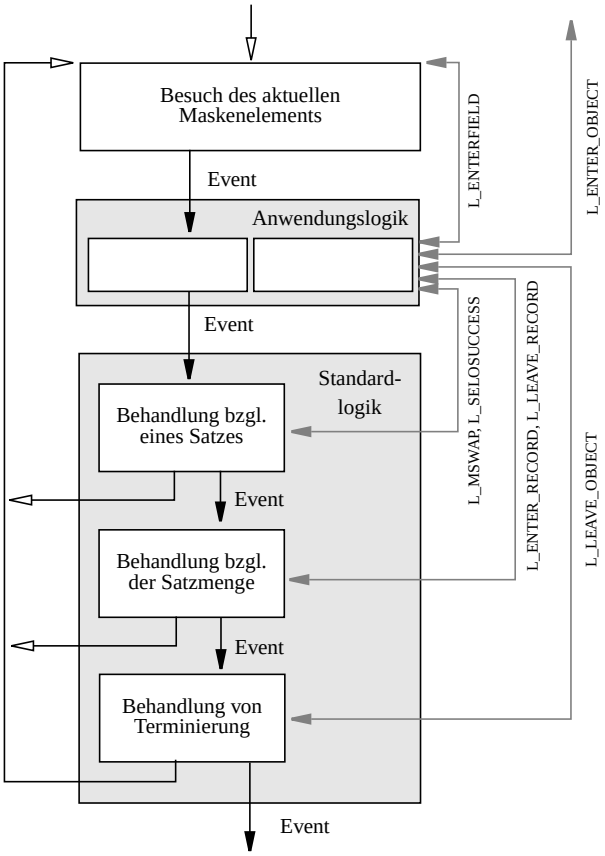
Bei einer Mehrsatz-Maske wird die Anwendungslogik nun mit dem Event L_ENTER_RECORD gerufen.

Anschließend werden die Elemente (des aktuellen Satzes) nacheinander besucht.[2] Vor jedem Besuch wird die Anwendungslogik mit dem Event L_ENTERFIELD gerufen; wird hierbei durch Zuweisung an *_fnr ein anderes aktuelles Element bestimmt, wird erneut L_ENTERFIELD generiert, bis das aktuelle Element stabil bleibt.

1. Dies gilt natürlich nur, wenn die Maske überhaupt Elemente besitzt. Handelt es sich bei dem Maskenelement um eine eingebettete Maske, muss ein geeigneter Cast des Zeigers _f erfolgen. *FIX* leitet _f aus *_fnr her; daher darf _f vom Entwickler nicht verändert werden.

2. Besucht werden können alle Maskenelemente unabhängig von ihren Eigenschaften; diese bestimmen nur den Verlauf des Besuchs.

Abb. 35 Besuchsmodus



1.1 Schritt 1: Besuch des aktuellen Maskenelements

Bei Mehrsatz-Masken hat *FIX* sichergestellt, dass zum Zeitpunkt eines Besuchs ein aktueller Satz existiert.

Besuch eines Feldes

Zunächst wird der Feldwert neu aufbereitet und ausgegeben (→ wrktoinfo()) und die Fortdauer des Turbomodus (Taste f3) geprüft: wenn der Feldwert nicht darstellbar ist (“***” im Feld) oder das Feld betretbar ist und die Eigenschaft NO-JUMP besitzt oder das gleiche Feld wieder besucht wird, in dem der Turbomodus ausgelöst wurde, so wird er abgeschaltet.

Fand unmittelbar vor dem Besuch aufgrund der Standardlogik eine Auswahl mittels eines Selos oder einer Choice statt, bei der das gleiche Feld aktuell war, stellt *FIX* die Taste RT in die Eingabe zurück. Dieses Verhalten fußt auf der Annahme, dass dabei ein Wert in das Feld übertragen wurde (vgl. hierzu Seite 214).

Anschließend wird die *FIX*-Funktion fx_accept() aufgerufen, die den eigentlichen Feldbesuch realisiert (ab Seite 208 beschrieben). fx_accept() gibt ein Event zurück, das (bis auf die im nächsten Abschnitt beschriebene Ausnahme) das Resultat des Besuchs bildet.

Im Anschluss an fx_accept() merkt sich *FIX*, welches Feld soeben besucht wurde, und schaltet, sofern das Feld mit der Taste f3 verlassen wurde (fx_accept() liefert in diesem Fall L_JMPFIELD), den Turbomodus ein; dabei merkt sich *FIX* das Element, bei dem er eingeschaltet wurde, und ersetzt L_JMPFIELD durch L_NXTFIELD.[1]

1. Die Anwendungslogik kann also nicht mehr unterscheiden, ob ein Feld mit der Taste RT oder f3 verlassen wurde.

Besuch einer eingebetteten Maske

Ist die Maske nicht betretbar (vgl. Seite 143), wird der Besuch sofort beendet. Das Ergebnis des Besuchs ist je nach Durchlaufrichtung (vgl. Seite 201) entweder L_PRVFIELD oder L_NXTFIELD.

Anderenfalls wird die eingebettete Maske durch einen rekursiven Aufruf von perform() abgearbeitet, dem die eingebettete Maske und dieselbe Anwendungslogik, die für die einbettende Maske benutzt wird, als Parameter mitgegeben werden. Ergebnis des Besuchs ist das vom perform()-Aufruf zurückgegebene Event (L_PRVFIELD oder L_NXTFIELD).

Nach dem Besuch bringt *FIX* automatisch wieder die einbettende Maske in den Vordergrund.

Achtung

Solange der rekursive Aufruf andauert, beziehen sich _fnr und _f auf die eingebettete Maske.

1.2 Schritt 2: Anwendungslogik

Anschließend wird die Anwendungslogik mit dem Event gerufen, das vom Besuch zurückgeliefert wurde. ∗_fnr enthält die Element-Position des besuchten Maskenelements, _f zeigt auf es.

Die Anwendungslogik kann auf das Event beliebig reagieren; sie muss lediglich die Maske in einem konsistenten Zustand als aktives Objekt hinterlassen.

Erst das von der Anwendungslogik zurückgegebene Event (bzw. das ihr übergebene Event, wenn sie NOTHING zurückgibt) wird dann von *FIX* interpretiert.

Hinweis zu klick-sensitiven Masken

1. BT_LEFT darf nur dann zurückgegeben werden, wenn die Anwendungslogik auch mit BT_LEFT aufgerufen wurde.
2. Maßgeblich bei der Standardreaktion auf BT_LEFT sind die Positionen der Feld*darstellungen*, wie sie zum Zeitpunkt des Klickens vorlagen. Eine durch die Anwendungslogik veranlasste andere Maskendarstellung bleibt unberücksichtigt. Es wird empfohlen, in diesem Falle BT_LEFT nicht weiterzureichen.

1.3 Schritt 3: Standardlogik

Die von der Standardlogik behandelten Events können in drei Gruppen eingeteilt werden:

- solche, die weder auf einen Wechsel des Satzes noch ein Verlassen der Maske abzielen,
- solche, die einen Wechsel des aktuellen Satzes bezwecken, und
- solche, die ein Beenden der Bearbeitung auslösen.

Jeder Gruppe entspricht eine Phase der weiteren Eventbehandlung.

Schritt 3a: Event-Behandlung bzgl. des Satzes

Das von der Anwendungslogik bestimmte Event wird zunächst lokal zu dem derzeit aktuellen Satz und Element zu behandeln versucht. Bei vollständig behandelten (✓) oder als unzulässig erkannten (*Error*) Events wird anschließend mit Schritt 1 fortgefahren, sonst ein Event (unten hinterlegt dargestellt) an die nächste Phase weitergereicht.

Event	Reaktion (alle Arten von Masken)	Wirkung
L_STAY	✓	Turbomodus beenden
K_MD, L_MODE	✓ [1]	Variante wechseln

K_f9, L_KEYSELECTION	✓ [1]	Auswahl durchführen
K_kh, L_FIRSTFIELD	✓ / L_FIRSTFIELD	zum Startelement, sofern nicht bereits dort
K_RT, L_NXTFIELD	✓ / L_NXTFIELD	zum nachfolgenden Element, sofern vorhanden
K_PL, L_PRVFIELD	✓ / L_PRVFIELD	zum vorangehenden Element, sofern vorhanden
BT_LEFT [2]	✓ / BT_LEFT	zu diesem Feld, sofern im gleichen Satz[3]
K_f8, L_KEYFIRST	L_KEYFIRST	
K_f0, L_KEYLAST	L_KEYLAST	
K_PD, L_FORWBROWSE	L_FORWBROWSE	
K_PU, L_BACKBROWSE	L_BACKBROWSE	
K_kd, L_CUR_DOWN	L_CUR_DOWN	
K_ku, L_CUR_UP	L_CUR_UP	
K_f7, L_DELETE	L_DELETE	
K_f6, L_ADD	L_ADD	
K_f4, L_INSERT	L_INSERT	
L_COPY_RECORD	L_COPY_RECORD	
L_CUT_RECORD	L_CUT_RECORD	
L_ADD_CLIPBOARD	L_ADD_CLIPBOARD	
L_INSERT_CLIPBOARD	L_INSERT_CLIPBOARD	
K_EN, L_END	L_END	
sonstige	*Error*	

Eine besondere Bedeutung kommt hierbei dem Event L_PRVFIELD zu. *FIX* merkt sich jeweils, ob dieses oder ein anderes Event behandelt wird, und leitet daraus eine *Durchlaufrichtung* für die Maske ab. Wird im Anschluss an eine Behandlung von L_PRVFIELD ein nicht betretbares Maskenelement besucht, liefert *FIX* für dessen Besuch L_PRVFIELD, wurde hingegen ein anderes Event behandelt, L_NXTFIELD als Ergebnis. Wird also ein Element rückwärts verlassen, werden anschließend unmittelbar folgende nicht betretbare Elemente ebenfalls rückwärts verlassen, bis ein Event ungleich L_PRVFIELD die Durchlaufrichtung wieder umkehrt.

Schritt 3b: Event-Behandlung bzgl. der Satzmenge

Weiter-Modus

Nur beim Maskentyp **submask** und nur, falls L_NXTFIELD zu behandeln ist, wird nun ein Zwischenschritt ausgeführt: um dem Anwender das Verlassen eines Satzes deutlich zu machen, wird die Maske in den Modus Weiter versetzt. Hierbei wird kurzzeitig ein Standard-Prompttext angezeigt, auf Eingabe gewartet und das entsprechende Event interpretiert:

Event	Reaktion	Wirkung
K_HP	✓	ein Hilfetext, der den Weiter-Modus erklärt, wird aufgeblendet
K_g5	✓	die Tastenhilfe wird aufgeblendet
K_RT	✓	Rückkehr ins Startelement des aktuellen Satzes
K_f8	L_KEYFIRST	

1. Hat ein Variantenwechsel stattgefunden, ruft *FIX* vor der Rückkehr zu Schritt 1 die Anwendungslogik mit dem Event L_MSWAP. Das Ergebnis dieses Aufrufs wird von *FIX* nicht ausgewertet.

1. Ist eine Auswahl getroffen worden, merkt sich *FIX* das aktuelle Maskenelement und ruft vor der Rückkehr zu Schritt 1 die Anwendungslogik mit dem Event L_SELOSUCCESS. Das Ergebnis dieses Aufrufs wird von *FIX* nicht ausgewertet.

2. bei klick-sensitiver Maske und Feld an der Position des Mauszeigers; bei Masken ohne Angabe zur Maus-Unterstützung wird BT_LEFT behandelt wie unter *sonstige* beschrieben.

3. Ist das Feld nicht betretbar, bleibt das aktuelle Feld unverändert.

K_f0	L_KEYLAST
K_f6	L_ADD
K_f4	L_INSERT
K_f2	L_COPY_RECORD
K_g7	L_INSERT_CLIPBOARD
K_f3	L_END
K_PL	L_PRVFIELD
K_kh	L_PRVFIELD
sonstige	*Error*

Bei K_RT wird in den Besuchsmodus zurückgekehrt, wobei die TOUCHED-Eigenschaft der Felder unverändert bleibt und das Startelement des aktuellen Satzes besucht wird. Bei allen übrigen vollständig behandelten oder als unzulässig erkannten Events wird im Weiter-Modus verblieben. Sonst wird das aufgeführte Event behandelt wie unten beschrieben.

Achtung:

Der Weiter-Modus macht es dem Anwender möglich, das von der Anwendungslogik vorgegebene Event L_NXTFIELD per Tastendruck durch eines der oben hinterlegt dargestellten Events zu überschreiben.

Das erhaltene Event wird nun bzgl. aller Sätze zu behandeln versucht. Bei vollständig behandelten oder als unzulässig erkannten Events wird wiederum mit Schritt 1 fortgefahren, sonst ein Event an die dritte Phase weitergereicht.

Event		Reaktion		Wirkung
	einfache Maske	Sub-/Rollmaske	Tabellenmaske	
L_KEYFIRST	*Error*	✓	✓	erster Satz
L_KEYLAST	*Error*	✓	✓	letzter Satz
L_FORWBROWSE	*Error*	✓	✓	nächste(r) Satz/ Seite
L_BACKBROWSE	*Error*	✓	✓	vorige(r) Satz/ Seite
L_CUR_DOWN	*Error*	*Error*	✓	nächster Satz
L_CUR_UP	*Error*	*Error*	✓	voriger Satz
BT_LEFT	*Error*	*Error*	✓	entspr. Satz, dort in entspr. Feld [1]
L_FIRSTFIELD	*Error*	L_PRVFIELD	✓/ L_PRVFIELD	oberster Satz, wenn nicht bereits dort
L_DELETE	*Error*	✓	✓	Satz löschen
L_INSERT	*Error*	✓	✓	Leersatz einfügen
L_ADD	*Error*	✓	✓	Leersatz anfügen
L_COPY_RECORD	*Error*	✓	✓	Satz auf Clipboard kopieren
L_CUT_RECORD	*Error*	✓	✓	Satz auf Clipboard kopieren und anschl. löschen
L_ADD_CLIPBOARD	*Error*	✓	✓	Satz auf dem Clipboard anfügen
L_INSERT_CLIPBOARD	*Error*	✓	✓	Satz auf dem Clipboard einfügen

1. Ist das Feld nicht betretbar, bleibt das aktuelle Feld unverändert.

L_NXTFIELD	L_NXTFIELD	✓	✓	nächster Satz
L_PRVFIELD	L_PRVFIELD	L_PRVFIELD	L_PRVFIELD	
L_END	L_NXTFIELD	L_NXTFIELD	L_NXTFIELD	

Bei Einzelsatz-Masken werden Events, die sich auf Satzmengen beziehen, abgewiesen. Bei Mehrsatz-Masken erfolgt die Behandlung der Events stets nach dem gleichen Schema:

1. Der Turbomodus wird deaktiviert.
2. Besitzt die Maske Sätze, ruft *FIX* die Anwendungslogik mit dem Event L_LEAVE_RECORD und macht so darauf aufmerksam, dass ein Wechsel des Satzes ansteht:

   Hierbei ist der Satz noch nicht gewechselt. Während der Behandlung von L_LEAVE_RECORD kann das auslösende Event mittels der Funktion get_triggering_event() abgefragt werden. Gibt die Anwendungslogik nicht L_LEAVE_RECORD (oder NOTHING) zurück[1], verbleibt *FIX* im alten Satz und bricht die Durchführung der Aktion ab, wobei drei Fälle unterschieden werden[2]:

zurückgegebenes Event	Durchlaufrichtung
L_PRVFIELD	die Durchlaufrichtung wird “rückwärts”
L_NXTFIELD	die Durchlaufrichtung wird “vorwärts”
sonstiges “sperrendes” Event	die Durchlaufrichtung bleibt unverändert

3. *FIX* versucht nun, zu dem durch das Event bestimmten Satz zu wechseln; wenn dies fehlschlägt (weil es z.B. keinen entsprechenden Satz gibt), wird, sofern die Maske überhaupt Sätze enthält, der dem beabsichtigten am nächsten kommende Satz zum aktuellen Satz (u.U. derselbe wie zuvor). Alle Felder verlieren die Eigenschaft TOUCHED.

und in den mit ✓ markierten Fällen:

4. Es wird sichergestellt, dass ein Satz existiert, wozu ggf. ein Leersatz angelegt wird. Anschließend ruft *FIX* die Anwendungslogik mit dem Event L_ENTER_RECORD und macht so darauf aufmerksam, dass ein Satzwechsel stattgefunden hat:

   Hierbei hat *FIX* im neuen Satz bereits ein erstes zu besuchendes Maskenelement bestimmt. Den Wert, den die Anwendungslogik zurückgibt, wertet *FIX* nicht aus.

Einzige Ausnahme hiervon bildet die Behandlung des Events L_COPY_RECORD.

Achtung

In bestimmten Situationen, z.B. L_FORWBROWSE im letzten Satz, wird die Anwendungslogik nacheinander mit L_LEAVE_RECORD und L_ENTER_RECORD für denselben Satz gerufen.

Bei der Behandlung von L_INSERT oder L_ADD geschieht Folgendes:

- Vor dem aktuellen bzw. hinter dem letzten Satz wird ein neuer Satz angelegt. Schlägt dies fehl, bleibt der Satz aktuell, in dem die Operation angestoßen wurde.
- Der neue Satz enthält in nicht gelinkten Feldern den in der Beschreibungsdatei angegebenen Defaultwert bzw. den sonst von *FIX* gesetzten Initialwert; besitzt das Feld die Eigenschaft COPY, stattdessen den Wert aus dem Satz, der aktuell war, als die Operation angestoßen wurde.
- Der neue Satz wird in den sichtbaren Bereich gebracht und zum aktuellen Satz gemacht. Aktuelles Element ist das Startelement.

Schritt 3c: Behandlung von terminierenden Events

Erreicht ein Event diese Phase (dies kann nur L_PRVFIELD oder L_NXTFIELD sein), ruft *FIX* die Anwendungslogik mit dem Event L_LEAVE_OBJECT. Während der Behandlung von L_LEAVE_OBJECT kann das auslösende Event mittels der Funktion get_triggering_event() abgefragt werden.

1. Das Durchreichen des Arguments (oder die Rückgabe von NOTHING) ist das typische Default-Verhalten für eine Anwendungslogik.

2. Diese mit Version 2.9.1 eingeführte Differenzierung erleichtert die Programmierung der Anwendungslogik, wenn vor dem Wechsel ein nicht betretbares Maskenelement besucht wird.

Gibt die Anwendungslogik nicht L_LEAVE_OBJECT (oder NOTHING) zurück, wird im Besuchsmodus verblieben; bei Mehrsatz-Masken wird hierbei der 4. Punkt aus Schritt 3b nachgeholt. Anderenfalls wird das Ende der Maskenbearbeitung eingeleitet.

1.4 Terminierung der Maskenbearbeitung

Masken-Aktion

Die mit der Maske assoziierte Aktion wird nur ausgeführt, wenn es sich bei dem aktuellen Element um das letzte Maskenelement handelt und das terminierende Event L_NXTFIELD ist. Bei einer Mehrsatz-Maske wird die Aktion auch dann ausgeführt, wenn die Maske keine Sätze enthält; wird hierbei auf Felder zugegriffen, ist deren Wert undefiniert (zufälliger Wert in den Feld-Hostvariablen).

Cleanup

Vor Verlassen einer Mehrsatz-Maske löscht *FIX alle* vorhandenen Sätze, die *leer* sind, es sei denn, der Entwickler hat ein anderes Verhalten spezifiziert (→ sm_install_cleanup()).

Nach der Bearbeitung entzieht *FIX* der Maske die Eigenschaft ACTIVE und entfernt die Maske vom Bildschirm (→ o_disappear()), sofern sie nicht die Eigenschaft STEADY besitzt. Bei einer Maske mit Varianten im Restart-Modus wird zur Hauptmaske zurückgekehrt.

Ergebnis des Aufrufs von perform() ist - mit Ausnahme des auf Seite 198 beschriebenen Sonderfalls - das Event, das L_LEAVE_OBJECT ausgelöst hat.

2 m_present()

m_present() realisiert eine eingeschränkte “Maskenbearbeitung”, die im Wesentlichen nur das Blättern durch die vorhandenen Sätze und das Verlassen der Maske auf einem definierten Satz erlaubt. Im Gegensatz zu perform() existiert zu keiner Zeit der Bearbeitung ein aktuelles Maskenelement (d.h. _fnr und _f sind stets undefiniert). Die Funktion ist zwar auch auf Einzelsatz-Masken anwendbar, doch macht dies selten Sinn.

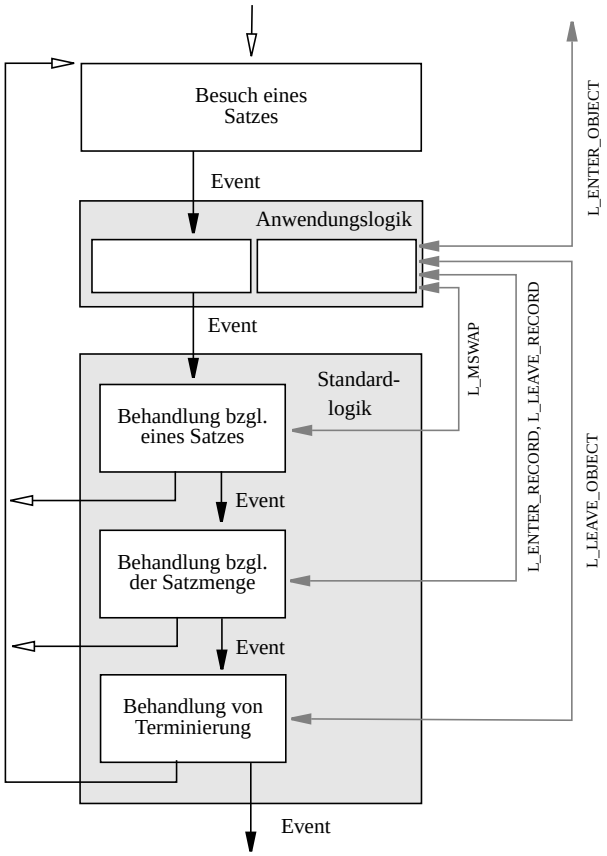
Bei einer Maske mit Varianten im Restart-Modus wird zunächst zur Hauptmaske zurückgekehrt (→ m_setvariant()). Anschließend lässt *FIX* die Maske am Bildschirm erscheinen (→ o_appear()) bzw. bringt sie in den Vordergrund (→ foreground()) und ersetzt den vorhandenen Prompttext durch den der Maske. Die Maske erhält für die Dauer der Bearbeitung die Eigenschaft ACTIVE, die erkennen lässt, dass die Maske in Bearbeitung ist. Die für die Bearbeitung der Maske gültige Anwendungslogik wird bestimmt und der Turbomodus deaktiviert.

Im Unterschied zu perform() erhält die Maske allerdings während der Bearbeitung einen speziellen Modus (Anzeige) und, sofern sie keinen eigenen Prompttext besitzt, einen Standard-Prompttext zugewiesen. Auch wird, dem readonly-Konzept folgend, bei einer Mehrsatz-Maske ohne Sätze kein Leersatz angelegt (vgl. Seite 198), und die TOUCHED-Eigenschaft der Felder bleibt unberührt.

Erst dann wird die Anwendungslogik mit dem Event L_ENTER_OBJECT gerufen. Gibt dieser Aufruf L_END zurück, wird der “Besuchsmodus” (vgl. Abb. 36) übersprungen; Ergebnis der Maskenbearbeitung ist L_END.

Bei jedem anderen Ergebnis wird jetzt der aktuelle Satz besucht - der Optik wegen bei einer Mehrsatz-Maske ohne Sätze ein *Phantomsatz* (real nicht vorhandener Leersatz). Beim Besuch eines (existenten) Satzes einer Mehrsatz-Maske wird die Anwendungslogik mit dem Event L_ENTER_RECORD gerufen.

Abb. 36 Besuchsmodus



Sofern nicht ausdrücklich ausgenommen, kann “aktueller Satz” im Folgenden auch Phantomsatz bedeuten.

2.1 Schritt 1: Besuch des aktuellen Satzes

Beim Besuch eines Satzes wird die Schreibmarke hinter den Satz positioniert - bei einer Tabellenmaske, die nicht leer ist, wird der Satz zusätzlich hervorgehoben dargestellt - und dort auf die Eingabe einer Sondertaste oder, außer bei klick-insensitiven Masken, ein Klicken gewartet. Zeichen-Tasten werden abgewiesen.

Hinweis zu klick-sensitiven Masken

Erkennt *FIX* das Event BT_LEFT, so ermittelt *FIX* die Bildschirmposition, an der der Klick erfolgt ist, baut die entsprechende Maske am Bildschirm neu auf und prüft dabei, ob die Position zu einem sichtbaren Feld der bearbeiteten Maske gehört. Ist dies nicht der Fall, wird BT_LEFT ignoriert und auf das nächste Event gewartet.

2.2 Schritt 2: Anwendungslogik

Nachdem eine Sondertaste oder ein Klicken erkannt wurde, wird die Anwendungslogik mit dem korrespondierenden Event gerufen.

Die Anwendungslogik kann auf das Event beliebig reagieren; sie muss lediglich die Maske in einem konsistenten Zustand als aktives Objekt hinterlassen. Ein Verändern der Datenmenge sollte unterbleiben, da dies dem Konzept der Funktion zuwiderläuft.

Erst das von der Anwendungslogik zurückgegebene Event (bzw. das ihr übergebene Event, wenn sie NOTHING zurückgibt) wird dann von *FIX* interpretiert.

Hinweis zu klick-sensitiven Masken

1. BT_LEFT darf nur dann zurückgegeben werden, wenn die Anwendungslogik auch mit BT_LEFT aufgerufen wurde.
2. Maßgeblich bei der Standardreaktion auf BT_LEFT sind die Positionen der Feld*darstellungen*, wie sie zum Zeitpunkt des Klickens vorlagen. Eine durch die Anwendungslogik veranlasste andere Maskendarstellung bleibt unberücksichtigt. Es wird empfohlen, in diesem Falle BT_LEFT nicht weiterzureichen.

2.3 Schritt 3: Standardlogik

Die von der Standardlogik behandelten Events können in die gleichen Gruppen wie bei perform() eingeteilt werden.

Schritt 3a: Event-Behandlung bzgl. des Satzes

Das von der Anwendungslogik bestimmte Event wird zunächst lokal zu dem derzeit aktuellen Satz zu behandeln versucht. Bei vollständig behandelten (✓) oder als unzulässig erkannten (*Error*) Events wird anschließend mit Schritt 1 fortgefahren, sonst ein Event an die nächste Phase weitergereicht.

Event	Reaktion (alle Arten von Masken)	Wirkung
L_STAY	✓	keine
K_MD, L_MODE	✓ [1]	Variante wechseln
K_g1, L_REBUILD	✓	Bildschirm neu aufbauen
K_sh	✓	Shell aufrufen
K_HP, L_HELP	✓	Standard-Hilfetext anzeigen
K_g5, L_T_HELP	✓	Tastenhilfe anzeigen
BT_LEFT [2]	✓ / BT_LEFT	keine, sofern angeklicktes Feld im aktuellen Satz
K_f8, L_KEYFIRST	L_KEYFIRST	
K_f0, L_KEYLAST	L_KEYLAST	
K_PD, L_FORWBROWSE	L_FORWBROWSE	
K_PU, L_BACKBROWSE	L_BACKBROWSE	
K_kd, L_CUR_DOWN	L_CUR_DOWN	
K_ku, L_CUR_UP	L_CUR_UP	
K_kh, L_FIRSTFIELD	L_FIRSTFIELD	
K_RT, L_NXTFIELD	L_NXTFIELD	
K_EN, L_END	L_END	
sonstige	*Error*	

Schritt 3b: Event-Behandlung bzgl. der Satzmenge

Ein Weiter-Modus wie bei perform() existiert hier nicht. Das erhaltene Event wird nun bzgl. aller Sätze zu behandeln versucht. Bei vollständig behandelten oder als unzulässig erkannten Events wird wiederum mit Schritt 1 fortgefahren, sonst ein Event an die dritte Phase weitergereicht.

---

1. Hat ein Variantenwechsel stattgefunden, ruft *FIX* vor der Rückkehr zu Schritt 1 die Anwendungslogik mit dem Event L_MSWAP. Das Ergebnis dieses Aufrufs wird von *FIX* nicht ausgewertet.
2. vgl. Fußnote auf Seite 201.

Event	Reaktion			Wirkung
	einfache Maske	Sub-/Rollmaske	Tabellenmaske	
BT_LEFT	*Error*	*Error*	✓	entspr. Satz
L_KEYFIRST	*Error*	✓	✓	erster Satz
L_KEYLAST	*Error*	✓	✓	letzter Satz
L_FORWBROWSE	*Error*	✓	✓	nächste(r) Satz/ Seite
L_BACKBROWSE	*Error*	✓	✓	vorige(r) Satz/ Seite
L_CUR_DOWN	*Error*	*Error*	✓	nächster Satz
L_CUR_UP	*Error*	*Error*	✓	voriger Satz
L_FIRSTFIELD	*Error*	L_END	✓ / L_END	oberster Satz, wenn nicht bereits dort
L_NXTFIELD	L_NXTFIELD	L_NXTFIELD	L_NXTFIELD	
L_END	L_END	L_END	L_END	

Das Vorgehen bei Mehrsatz-Masken entspricht dem bei perform() (siehe Seite 203) mit folgenden Einschränkungen:

- Der Turbomodus bleibt unberührt.
- Beim Wechsel des Satzes wird weder bei einer Mehrsatz-Maske ohne Sätze ein Leersatz angelegt noch wird ein zu besuchendes Maskenelement bestimmt. Auch bleibt die TOUCHED-Eigenschaft der Felder unberührt.

Schritt 3c: Behandlung von terminierenden Events

Erreicht ein Event diese Phase (dies kann nur L_END oder L_NXTFIELD sein), ruft *FIX* die Anwendungslogik mit dem Event L_LEAVE_OBJECT. Während der Behandlung von L_LEAVE_OBJECT kann das auslösende Event mittels der Funktion get_triggering_event() abgefragt werden.

Gibt die Anwendungslogik nicht L_LEAVE_OBJECT (oder NOTHING) zurück, wird im Besuchsmodus verblieben; bei Mehrsatz-Masken wird, sofern ein aktueller Satz existiert, der Aufruf der Anwendungslogik mit dem Event L_ENTER_RECORD nachgeholt. Anderenfalls wird das Ende der Maskenbearbeitung eingeleitet.

2.4 Terminierung der Maskenbearbeitung

Masken-Aktion

Die mit der Maske assoziierte Aktion wird nur ausgeführt, wenn das terminierende Event L_NXTFIELD ist. Bei einer Mehrsatz-Maske wird die Aktion auch dann ausgeführt, wenn die Maske keine Sätze enthält; wird hierbei auf Felder zugegriffen, ist deren Wert undefiniert (zufälliger Wert in den Feld-Hostvariablen).

Cleanup

Die Cleanup-Routine (siehe Seite 204) kommt entsprechend dem Konzept von m_present() nicht zur Ausführung.

Modus und Prompttext der Maske werden restauriert, wie sie vor Beginn des Aufrufs von m_present() gültig waren.

Nach der Bearbeitung entzieht *FIX* der Maske die Eigenschaft ACTIVE und entfernt die Maske vom Bildschirm (→ o_disappear()), sofern sie nicht die Eigenschaft STEADY besitzt. Bei einer Maske mit Varianten im Restart-Modus wird zur Hauptmaske zurückgekehrt.

Ergebnis des Aufrufs von m_present() ist - mit Ausnahme des auf Seite 204 beschriebenen Sonderfalls - das Event, das L_LEAVE_OBJECT ausgelöst hat.

3 Besuch eines Feldes

3.1 Ablauf

Der Besuch eines Feldes wird durch die Funktion fx_accept() realisiert. Ihr Verhalten hängt sowohl von den Feldeigenschaften als auch vom Umfeld des Besuchs ab: beim ersten Besuch eines Elements (im aktuellen Satz) oder wenn zuvor ein Element mit kleinerer Element-Position besucht wurde, wird von einem *Betreten von vorne* gesprochen, wurde zuvor ein Element mit größerer Element-Position besucht, von einem *Betreten von hinten*. Als dritte Möglichkeit kommt das *sofortige Wiederbetreten* hinzu, ohne dass zwischendurch ein anderes Element der Maske besucht wird. Zum Kontext des Besuchs gehört weiter die Durchlaufrichtung (vgl. Seite 201).

fx_accept() unterscheidet - in dieser Reihenfolge - folgende Fälle:

*Fall 1: Feld besitzt obligatorische Choice (**use**)*

Betreten von vorne: Sofern der Turbomodus nicht aktiv ist (vgl. hierzu Abschnitt 1.1), bietet *FIX* die Choice zur Auswahl an, nachdem er sich zuvor den aktuellen Feld*inhalt* gemerkt hat. Findet eine Auswahl statt, existiert eine Wertübernahme-Routine und liefert deren Aufruf keinen Fehler, so bleibt die Durchlaufrichtung erhalten, ansonsten wird sie auf “rückwärts” umgekehrt und der ursprüngliche Feldwert wird restauriert.

Betreten von hinten: *FIX* bietet die Choice zur Auswahl an, nachdem er sich zuvor den aktuellen Feld*inhalt* gemerkt hat. Findet eine Auswahl statt, existiert eine Wertübernahme-Routine und liefert deren Aufruf keinen Fehler, so wird die Durchlaufrichtung nach “vorwärts” geändert, ansonsten bleibt sie erhalten und der ursprüngliche Feldwert wird restauriert.

Sofortiges Wiederbetreten: Die Choice wird nicht angeboten.

Das Feld erhält die Eigenschaft TOUCHED, wenn sich der neue Wert von dem ursprünglichen Wert unterscheidet (vgl. “TOUCHED auf Feldebene” auf Seite 210). Anschließend wird wie bei Fall 2 weiterverfahren.

Achtung:

Für das Anbieten der Choice ist es unerheblich, ob das Feld die Eigenschaft NOENT besitzt oder nicht.

Fall 2: Feld nicht betretbar (NOENT)

Gehört zu dem Feld eine am Bildschirm dargestellte Choice, so wird die darin vorgenommene Auswahl an den Feldwert angepasst. Die Durchlaufrichtung bleibt erhalten, die Felderfassung wird ohne irgendwelche Prüfungen bei Durchlaufrichtung “vorwärts” mit dem Event L_NXTFIELD, bei Durchlaufrichtung “rückwärts” mit dem Event L_PRVFIELD beendet.

*Fall 3: Feld besitzt bevorzugte Choice (**offer**) und ist leer, erfordert aber Wert (REQUIRED)*

Betreten von vorne: Sofern der Turbomodus nicht aktiv ist, bietet *FIX* die Choice zur Auswahl an, nachdem er sich zuvor den aktuellen Feld*inhalt* gemerkt hat. Findet eine Auswahl statt, existiert eine Wertübernahme-Routine und liefert deren Aufruf keinen Fehler, so stellt *FIX* die Taste RT in die Eingabe zurück (→ undcgetch()), die dann nachfolgend gelesen wird; anderenfalls restauriert *FIX* den ursprünglichen Feldwert.

Betreten von hinten: *FIX* bietet die Choice zur Auswahl an, nachdem er sich zuvor den aktuellen Feld*inhalt* gemerkt hat. Findet keine Auswahl statt, stellt *FIX* die Taste PL in die Eingabe zurück, die dann nachfolgend gelesen wird. Findet eine Auswahl statt, existiert eine Wertübernahme-Routine und liefert deren Aufruf keinen Fehler, so stellt *FIX* stattdessen die Taste RT in die Eingabe zurück; anderenfalls restauriert *FIX* den ursprünglichen Feldwert.

Sofortiges Wiederbetreten: Die Choice wird nicht angeboten.

Das Feld erhält die Eigenschaft TOUCHED, wenn sich der neue Wert von dem ursprünglichen Wert *hinsichtlich der Darstellung* unterscheidet. Anschließend wird wie in Fall 4 weiterverfahren.

Hinweis:

Ist das Feld nicht leer oder besitzt es nicht die Eigenschaft REQUIRED, wird die Choice nur auf Anforderung (Taste f9) angeboten.

Fall 4: Feld ist betretbar

Zunächst merkt *FIX* sich den zur Realisierung von “Undo” und zur Bestimmung der Eigenschaft TOUCHED benötigten aktuellen Feld*inhalt*. Bei einem Feld mit Wertvorgabe (**values**) wird der Aufsetzpunkt für den Blätter-Mechanismus bestimmt.[1] Außerdem wird der vorhandene Prompttext durch den zu dem Feld gehörigen Prompttext (bzw. einen nur temporär gültigen Standardtext wie “Auswahlfeld: …”) ersetzt.

Nun kann der Feldinhalt editiert werden, wobei zwischendurch auch die *innerhalb der Felderfassung* unterstützten Sondertasten zum Bewegen der Schreibmarke, Anzeigen des Hilfetextes usw. benutzt werden können (vgl. Seite 21).

Bei Sondertasten, die fx_accept() selbst nicht unterstützt, oder, außer bei klick-insensitiven Masken, einem Klicken wird die Felderfassung zu beenden versucht (vgl. Hinweis auf Seite 210). Hierzu müssen die im Feld stehenden Zeichen einen syntaktisch korrekten Wert ergeben.

Erkennt *FIX* die Taste RT (im Turbomodus simuliert *FIX* den Erhalt dieses Tasten-Events, statt Eingabe von der Tastatur zu erwarten) oder die Taste f3, so wird der Wert den für das Feld definierten Prüfungen unterzogen und zwar in folgender Weise:

1. Besitzt das Feld die Eigenschaft REQUIRED, so darf es nicht leer sein.
2. Besitzt es die Eigenschaft DBMUST und ist nicht leer, so muss die Suche mittels Selo (→ dbmust()) oder Choice (→ chmust()) erfolgreich sein (Ergebnis == SUCCESS).
3. Ist ein Muster (regulärer Ausdruck) für das Feld definiert und ist das Feld nicht leer, so muss der Feldwert dem regulären Ausdruck entsprechen.
4. Ist eine Prüfroutine für das Feld definiert und ist das Feld nicht leer, so muss deren Aufruf erfolgreich sein (Ergebnis != 0).

Die Prüfungen werden beim ersten Fehlschlag beendet; es wird dann in der Felderfassung verblieben. Verlaufen alle Feldprüfungen erfolgreich (und besitzt das Feld nicht die Eigenschaft NOMOD), wird dem Feld der in dem lokalen Puffer gespeicherte Wert zugewiesen (→ fputval()). Die Durchlaufrichtung wird auf “vorwärts” gesetzt.

Bei anderen Tasten oder bei Klicken wird, sofern das Feld nicht die Eigenschaft NOMOD besitzt, der Wert dem Feld ohne weitere Feldprüfungen zugewiesen.

Das Feld erhält die Eigenschaft TOUCHED, wenn sich der neue Wert von dem ursprünglichen Wert *bzgl. der Darstellung* unterscheidet.

Gehört zu dem Feld eine am Bildschirm dargestellte Choice, so wird die darin vorgenommene Auswahl an den Feldwert angepasst. Bei Scrollfeldern wird der Ausschnitt wieder an den linken Rand zurückgeschoben.

Das Feld wird neu ausgegeben, wobei die Schreibmarke sich wieder wie ausgangs am linken Rand des Feldes befindet.

---

1. *FIX* verwaltet die Bestandteile der **values**-Angabe in einer zirkulären Struktur, die stets ein aktuelles Element enthält, von dem aus weitergeblättert wird. Sofern ein Element existiert, dessen Darstellung (bis auf Füllzeichen am Ende) mit dem Feld*inhalt* übereinstimmt, wird dies zum aktuellen Element; anderenfalls bleibt der vorgefundene Aufsetzpunkt erhalten.

Abschließend erfolgt eine Bildschirmaktualisierung (→ sync_refresh()), um den neuen Zustand des Feldes sichtbar zu machen.

Die Felderfassung wird mit dem Event verlassen, das der zuletzt erkannten Taste entspricht.

Hinweis zu klick-sensitiven Masken

Erkennt *FIX* bei der Felderfassung das Event BT_LEFT, so ermittelt *FIX* die Bildschirmposition, an der der Klick erfolgt ist, baut die entsprechende Maske am Bildschirm neu auf und prüft dabei, ob die Position zu einem sichtbaren Feld der bearbeiteten Maske gehört. Ist dies nicht der Fall, wird BT_LEFT ignoriert und auf das nächste Event gewartet, anderenfalls das Feld mit dem Event BT_LEFT verlassen.

3.2 Aufbau von Routinen zur Feldprüfung

Die C-Funktion, die der Entwickler einem Feld als Prüfroutine zuordnet, erhält beim Aufruf als Argument den Zeiger auf das Feld, dessen Erfassung *FIX* sich anschickt abzuschließen, und muss einen Wert vom Typ int zurückliefern. Ist der Wert ungleich 0, gilt der Feldwert als akzeptiert und die Felderfassung wird beendet; anderenfalls wird sie fortgesetzt.

Die Komponente *wrkp* des Feldes zeigt für die Dauer des Aufrufs auf einen internen Puffer, in dem *FIX* den Wert (datentypgerecht) hält, der im Falle des Akzeptierens dem Feld zugewiesen würde. Daher dürfen innerhalb der Prüfroutine keine “höheren” objekt- oder feldbezogene Funktionen aus der *FIX*-Library verwendet werden: sie würden unzulässigerweise auf den noch gar nicht installierten neuen Wert zugreifen. Triviale Testfunktionen wie f_isnull(), fis_empty() etc. sind allerdings erlaubt.

Der Wert darf nicht modifiziert werden.

4 Die Eigenschaft TOUCHED

4.1 TOUCHED auf Feldebene

Die Eigenschaft TOUCHED dient dazu, dem Entwickler die Chance zu geben zu erkennen, ob der Anwender den Wert eines Feldes *interaktiv* verändert hat.

Ein Feld erhält die Eigenschaft TOUCHED automatisch, wenn bei der Bearbeitung einer Maske die *normierte textuelle Darstellung* des Feldwertes zu Beginn und am Ende

- eines Feldbesuchs (→ fx_accept()) oder
- der Standardbehandlung des Events L_KEYSELECTION durch *FIX* (Auswahl aus Selo oder Choice)

nicht übereinstimmt.[1]

FIX entzieht den Feldern einer Maske die Eigenschaft TOUCHED

- beim Rücksetzen der Feldeigenschaften auf die in der Beschreibungsdatei angegebenen Eigenschaften (→ restart(), m_restore_properties()) und
- bei der Bearbeitung einer Maske mittels perform()

1. Bei Gleitkommawerten bietet TOUCHED keine Gewähr dafür, dass sich der Feld*wert* nicht geringfügig geändert hat: Enthält beispielsweise ein float-Feld anfangs den Wert 1.285, wird aber mit nur 2 Nachkommastellen dargestellt, also als 1.29, so wird am Ende der Felderfassung 1.29 zurückgeschrieben.

- vor dem ersten stattfindenden Besuch
- bei der Behandlung eines satzübergreifenden Events, allerdings nur, wenn die Behandlung des Events L_LEAVE_RECORD ein Verlassen des Satzes nicht von vornherein verhindert (vgl. Seite 203).

Da es sich weniger um eine Eigenschaft des Feldes als vielmehr des Feldwertes handelt, erhält bzw. verliert mit einem Feld-Link auch das Root-Feld die Eigenschaft; mit dem Root-Feld erhalten/verlieren alle darauf bestehenden Feld-Links die Eigenschaft ebenfalls.

Funktionen

Zum Manipulieren der Eigenschaft TOUCHED bietet *FIX* an:

```
long    f_state(field *f, int mode, unsigned long state)
```

(void)f_state(*f*, P_ADD, TOUCHED)	setzt Eigenschaft
(void)f_state(*f*, P_DEL, TOUCHED)	entzieht Eigenschaft
f_state(*f*, P_GET, TOUCHED) == TOUCHED	prüft Eigenschaft

Hinweise

Die Benutzung eines Feldes zur Startwerterfassung, als Ziel eines Wertexports aus einem Selo oder ein Ändern des Feldwertes per Programm ändern das TOUCHED-Attribut nicht.

Beim Variantenwechsel oder Umschalten der Zeilenvariante bleibt die Eigenschaft TOUCHED erhalten.

4.2 TOUCHED auf Satzebene

Der aktuelle Satz einer Mehrsatz-Maske bekommt die Eigenschaft TOUCHED automatisch verliehen, sobald

- ein Feld der Maske, das zu einer Spalte der Satzmenge in Beziehung steht, aufgrund eines der in *4.1* beschriebenen Fälle die Eigenschaft TOUCHED erhält,
- bei der Bearbeitung der Maske mittels perform() ein Satz als Standardreaktion auf L_ADD (bzw. L_NXTFIELD im letzten Element des letzten Satzes), L_INSERT, L_ADD_CLIPBOARD oder L_INSERT_CLIPBOARD angelegt wird (vgl. Seite 201f).

Automatisch verliert ein Satz die Eigenschaft TOUCHED nie.

Funktionen

Zum Manipulieren der Eigenschaft TOUCHED bietet *FIX* an:

für den aktuellen Satz der Mehrsatz-Maske *smsp*

```
long    mr_rowstate((MemRelType mrp, int mode, long state)
```

(void)mr_rowstate(*smsp*->sm_mr_cb, P_ADD, TOUCHED)	setzt Eigenschaft
(void)mr_rowstate(*smsp*->sm_mr_cb, P_DEL, TOUCHED)	entzieht Eigenschaft
mr_rowstate(*smsp*->sm_mr_cb, P_GET, TOUCHED) == TOUCHED	prüft Eigenschaft

bzw. für einen beliebigen Satz im Tupel *tp*

```
unsigned long    tp_state(TupelType tp, int mode, unsigned long state);
```

(void)tp_state(*tp*, P_ADD, TOUCHED)	setzt Eigenschaft

(void)tp_state(*tp*, P_DEL, TOUCHED)	entzieht Eigenschaft
tp_state(*tp*, P_GET, TOUCHED) == TOUCHED	prüft Eigenschaft

Hinweise

Wird einem Feld die Eigenschaft TOUCHED per Programm verliehen, so sollte dies ggf. auch für den Satz geschehen.

4.3 TOUCHED auf Objektebene

Auf Objektebene wird TOUCHED in der Form unterstützt, dass mittels Funktionen abgefragt werden kann, ob zurzeit

- eines der Maskenfelder die Eigenschaft TOUCHED besitzt
  BOOLEAN m_touched(mask *mskp)

- der aktuelle Satz die Eigenschaft TOUCHED besitzt (nur bei Mehrsatz-Masken, siehe Abschnitt 4.2 auf Seite 211).

Um allen Feldern einer Maske die Eigenschaft TOUCHED zu entziehen, bietet *FIX* die Funktion

```
void    m_untouch(mask *mskp)
```

an.

m_touched() und m_untouch() operieren auch auf Feld-Links.

32 Programmieren mit Selos und Choices

1 Eigenständiger Gebrauch

Selos und Choices können als selbständige Objekte geladen (→ loadselo(), loadchoice()) und dem Anwender zur Bearbeitung angeboten werden (→ perform()).

Liefert die Bearbeitung eines Selos als Ergebnis L_NXTFIELD, wurde ein Satz ausgewählt. Auf seine Spaltenwerte kann bis zu einem erneuten Gebrauch des Selos mit Library-Routinen zugegriffen werden (→ fx_selo_fld_adr(), fputselo()).

Bei Choices kann nach der Bearbeitung, die bei Abbruch als Ergebnis L_STAY, sonst L_NXTFIELD liefert, die getroffene Selektion ebenfalls mittels Library-Routinen ermittelt werden (→ tp_get_mark()), jedoch nur solange, bis entweder ein Abgleich mit einem Feldwert oder eine Neubestimmung der Datenmenge erfolgt (vgl Seite 109).

FIX bricht die Bearbeitung einer Choice automatisch ab, wenn einer der folgenden Fälle eintritt:

- die Choice besitzt keine Datenmenge,
- die Datenmenge ist leer,
- die Auswahlkriterien (**range**) sind für die gegebene Datenmenge nicht erfüllbar, weil sie z.B. die Selektion von mehr Elementen fordern, als in der Datenmenge vorhanden sind.

2 Interaktiver Gebrauch in Verbindung mit einem Feld

Obgleich Selos und Choices eigenständig benutzt werden können, werden sie meist in Verbindung mit einem Feld eingesetzt. Ist in einer Maskenbeschreibung bei der Definition eines Feldes ein Selo oder eine Choice aufgeführt, wird dieses Objekt im Anschluss an die Maske automatisch geladen und diesem Feld zugeordnet.

Wird bei der Bearbeitung einer Maske die Taste f9 (KEYSELECTION) betätigt und hat der Entwickler keine eigene Logik für diese Taste definiert, prüft *FIX*, ob dem gerade bearbeiteten Feld ein Selo oder eine Choice zugeordnet ist. Diese werden in der Regel dazu dienen, dem Anwender die Bestimmung eines neuen Feldwertes durch Auswahl zu ermöglichen.

2.1 Selo

Ist dem Feld ein Selo zugeordnet, wird zunächst der bisherige Feldinhalt gerettet und dann das Selo zur Bearbeitung angeboten (→ perform()).

Wenn keine Auswahl erfolgt (perform() liefert L_STAY), bleiben die im Selo aufgeführten Wertübernahme-Beziehungen unausgewertet und *FIX* betritt das Feld, das unberührt geblieben ist, von neuem.

Anderenfalls (perform() liefert L_NXTFIELD) werden die Wertübernahme-Beziehungen ausgewertet. Scheitert eine der Wertübernahmen - Achtung: erfolgreich verlaufene werden hierdurch nicht wieder zurückgesetzt -, betritt *FIX* das Feld ebenfalls von neuem.

Nur bei erfolgreichem Auswerten aller Wertübernahme-Beziehungen wird die TOUCHED-Eigenschaft des Feldes neu bestimmt und das Event L_SELOSUCCESS generiert, auf das die Anwendungslogik reagieren kann (vgl Abb. 35). Wird hierbei das aktuelle Feld nicht gewechselt, stellt *FIX* die Taste RT in die Eingabe zurück, ehe das Feld von neuem betreten wird. Dieses Verhalten fußt auf der Annahme, dass eine Spalte der Ergebnismenge in einer Wertübernahme-Beziehung zu dem Feld steht und der übernommene Wert der fortan gültige Wert des Feldes sein soll. Ist dies nicht vorgesehen, muss bei der Reaktion auf L_SELOSUCCESS ein anderes aktuelles Feld bestimmt oder das Ereignis L_KEYSELECTION vollständig in der Anwendungslogik behandelt werden.

2.2 Choice

Ist dem Feld eine Choice zugeordnet, wird zunächst der bisherige Feldinhalt gerettet und die Choice zur Auswahl angeboten (→ perform()). Bei einer dynamischen Choice wird hierzu die Datenmenge neu beschafft. Bevor eine Interaktion mit dem Anwender oder ein automatischer Abbruch (vgl. Seite 213) erfolgt, wird, sofern eine entsprechende Routine vorgegeben ist (**import**), ein Abgleich der Selektion mit dem Feldwert durchgeführt.

Wenn keine Selektion vorgenommen wird (perform() liefert Wert ungleich L_NXTFIELD), bleibt die Wertübernahme-Funktion unausgewertet und *FIX* betritt das Feld von neuem.

Anderenfalls (perform() liefert L_NXTFIELD) wird, sofern vorgesehen, die Wertübernahme-Funktion aufgerufen (**export**). Ist keine vorgesehen, ist sie *FIX* nicht bekannt oder gibt ihr Aufruf einen Fehler zurück (in den beiden letzten Fällen erfolgt eine Meldung), betritt *FIX* das Feld ebenfalls von neuem.

Nur bei erfolgreichem Aufruf der Wertübernahme-Funktion wird die TOUCHED-Eigenschaft des Feldes neu bestimmt und das Event L_SELOSUCCESS generiert, auf das die Anwendungslogik reagieren kann. Wird hierbei das aktuelle Feld nicht gewechselt, stellt *FIX* die Taste RT in die Eingabe zurück, ehe das Feld von neuem betreten wird. Dieses Verhalten fußt auf der Annahme, dass die Wertübernahme-Funktion dem Feld einen Wert gegeben hat, der fortan der gültige Wert des Feldes sein soll. Ist dies nicht vorgesehen, muss bei der Reaktion auf L_SELOSUCCESS ein anderes aktuelles Feld bestimmt oder das Ereignis L_KEYSELECTION vollständig in der Anwendungslogik behandelt werden.

Zum Anschluss von Choices an Felder mittels **use** und **offer** vgl. “Besuch eines Feldes” auf Seite 208.

3 Benutzung zur Feldprüfung und Datenbeschaffung

Einen weiteren Einsatz finden Selos und Choices bei der Prüfung von Feldeingaben (→ dbmust(), chmust(), DBMUST).

3.1 Selo

Hier muss im Selo eine Wertübernahme-Beziehung zwischen einer Spalte der Ergebnismenge und dem Feld definiert sein. Durch dbmust() kann festgestellt werden, ob die durch die Grundanweisung oder den SPL-Prozeduraufruf definierte Ergebnismenge einen Satz enthält, der in dieser Spalte den gleichen Wert wie das Feld aufweist. Dieser Satz wird gelesen; ggf. können Spaltenwerte dieses Satzes in weitere Felder übernommen werden.

select-Anweisung

Hier ist zu beachten, dass IBM Informix ESQL/C keine dynamische Einzelsatz-Suche (execute) erlaubt und - im ungünstigsten Fall, wenn der Zugriff nicht eindeutig ist - bei jedem Aufruf eine Folge von Operationen der Form prepare - declare - open - fetch - close - free durchlaufen werden muss.

Zur Datenbeschaffung erweitert *FIX* die im Selo beschriebene Grundanweisung um die Bedingung

<expr> = #<maskname>.<fieldname>

wobei

<fieldname> Name des Feldes, an das das Selo gebunden ist,

<maskname> Name der Maske, zu der dieses Feld gehört,

<expr> Ausdruck, der die Spalte definiert, die in einer Wertübernahme-Beziehung zum Feld steht

und versucht, einen Satz zu lesen. Liefert die Anfrage mehr als einen Satz, ist der gelesene Satz zufällig.

Beispiel:

```
select auftr.aufnr, auftr.datum, auftr.bearb, auftr.kunr, kunde.name from auftr, kunde
where auftr.aufnr = #AUFTR.AUFNR and auftr.kunr = kunde.kunr and auftr.bearb > " "
```

execute procedure-Anweisung

Im Falle der Datenbeschaffung mittels SPL-Prozedur erfolgt ein Aufruf der Prozedur, zu deren Argumenten #<maskname>.<fieldname> gehören sollte. Für das Argument # wird der Wert 3 und für die mit # eingeleitete Feldreferenzen werden die jeweiligen Feldwerte eingesetzt. Der Aufruf muss im Erfolgsfall einen Datensatz zurückgeben und sonst den Fehler SQLNOTFOUND erzeugen (RAISE EXCEPTION 100).

Wird ein Satz gefunden, werden die Werte von Spalten, zu denen Wertübernahme-Beziehungen existieren, in die entsprechenden Felder übernommen. Der Suchwert sollte in diesem Fall Schlüsselfunktion haben.

Aus dem Ergebnis der Funktion dbmust() geht hervor, ob ein Satz gefunden wurde oder nicht.

3.2 Choice

Hierzu muss in der Datenmenge (Memory Relation) der Choice eine ausgezeichnete (“auserkorene”) Spalte existieren (→ mr_select_column()). Bei aufgrund der Datenbeschaffungsvorschrift beschafften Datenmengen zeichnet *FIX* regelmäßig zunächst die erste Spalte aus. Ausgezeichnete Spalte und Feld müssen den gleichen Typ haben.

chmust() versucht, einen Satz zu finden und zum aktuellen Satz zu machen, der in der ausgezeichneten Spalte mit dem Feldwert übereinstimmt. Bei dynamischen Choices wird aus diesem Anlass die Datenmenge zuvor neu beschafft, was zum Verlust der bestehenden Selektion führt.

Existiert mehr als ein solcher Satz, ist der aktuelle Satz zufällig.

Aus dem Ergebnis der Funktion chmust() geht hervor, ob ein Satz gefunden wurde oder nicht.

3.3 Selos als Tabellen darstellen

Auf ähnliche Weise wie Tabellen, können Selos windowsähnlicher gezeichnet werden. Dazu ist die Funktion

```
enableSeloAsTable("{!/!/}", PA_TABLEHEADER_FIRST, PA_TABLEHEADER_LAST)
```

aufzurufen. Sie bekommt als Parameter eine Zeichenkette, mit den Codes, die für die Tabellenzellen des Selos als Feldbegrenzer verwendet werden. Die Zeichenkette muss genau 6 Zeichen lang sein. Die ersten beiden Zeichen werden für das erste Feld, die letzen beiden für das letzte Feld und die mittleren beiden für die mittleren Felder verwendet. In dem obigen Beispiel werden die gleichen Zeichen wie für die Tabelle verwendet. Die beiden anderen Parameter dienen zum Zeichnen der Paintareas für die Überschriften. Die erste dient für die vorderen Paintareas der Überschrift, die zweite

für die letzte Paintarea. In dem Beispiel werden auch hier die gleichen Werte wie für die Tabelle verwendet. Da die Felder wegen den beiden Feldbegrenzern um zwei Zeichen länger werden, wurden die Überschriften ebenfalls um zwei Leerzeichen verlängert. Dies ist bei der Übersetzungsfunktion für Paintareas zu beachten. Zur Darstellung der Auswahlbuchstaben werden ebenfalls Paintareas verwendet. Hier wird als Wert für longval1 die erste ID verwendet, so dass diese auf die gleiche Weise wie die Überschrift dargestellt werden. Erkennt die Funktion enableSeloAsTable() einen Fehler, dann liefert sie FALSE zurück. Das kann einen der folgenden Gründe haben:

- Die Länge der Zeichenkette in delim ist ungleich 6.
- Die Anwendung wird nicht von *FIX/Win* bedient.

Tritt kein Fehler auf, liefert die Funktion TRUE zurück. Soll die Darstellung wieder abgeschaltet werden, dann ist die Funktion nochmals aufzurufen, wobei mindestens einer der Parameter mit 0 zu belegen ist:

```
enableSeloAsTable(NULL, 0L, 0L);
```

33 Programmgesteuertes Aufblenden von Hilfetexten

Für Hilfetexte zum gerade besuchten Element oder aktiven Objekt sollte der von *FIX* gebotene Mechanismus benutzt werden, der keinerlei Programmierung erfordert. Falls ein Hilfetext aber situationsbezogen oder nicht aus dem Element- oder Objektnamen ableitbar ist, kann er im Programm auch explizit angesprochen werden. Hierzu dienen die Funktionen

```
void    entry_help(void)
void    help(field *f)
void    n_help(field *f, char *name)
void    chelp(field *f)
void    t_help(field *f)
```

entry_help() zeigt den Hilfetext zum aktuellen Menüpunkt aus dem Verzeichnis $HLPPATH (Achtung: beim Aufruf muss ein Menü aktives Objekt sein!).

help() zeigt den Hilfetext zum Feld *f* aus dem Verzeichnis $HLPPATH.

n_help() zeigt den Hilfetext *name* aus dem Verzeichnis $HLPPATH. Das Feld *f* dient nur zur Positionierung des Windows; ggf. kann ein Hilfsfeld verwendet werden.

chelp() zeigt den *privaten* Hilfetext zum Feld *f* aus dem Verzeichnis $CHLPPATH.

t_help() zeigt den Hilfetext, der üblicherweise beim Drücken der Taste g5 (T_HELP) erscheint und die Standard-Bedeutung der Tasten erklärt. Auch hier dient *f* nur zur Positionierung des Windows.

Hinweis:

Die Seiten eines Hilfetextes werden nicht gespeichert, sondern jedes Mal neu aus der entsprechenden Datei gelesen; Hilfetexte können also vom Programm selbst (möglichst im Verzeichnis $CHLPPATH) generiert oder modifiziert werden.

Für Tastenbezeichnungen eingesetzte Tastenbeschriftungen werden stets mit Leerzeichen auf 5 Zeichen aufgefüllt, um eine vernünftige Formatierung des Hilfetextes zu erlauben.

Überlagern der "Hilfe"-Funktionalität

Um dem Entwickler die Möglichkeit zu bieten, unter einer grafischen Oberfläche Hilfen in anspruchsvollerer Form zu präsentieren, kann das Verhalten der entsprechenden *FIX*-Routinen vom Entwickler überlagert werden.

Näheres hierzu findet sich in der Beschreibung der globalen Variable S_ApplicationHelpHandler auf Seite 441.

34 Programmieren mit Paintareas

1 Datenstrukturen von Paintareas

Die Makros und Strukturen, die für eine Paintareas benötigt werden, sind in der Datei `fix/fixwin.h` definiert, die über `fix/fix.h` eingebunden wird. Die Funktionsprototypen werden in `proto/fixwin_pto.h` definiert. Diese Datei wird über `fix/libfix.h` eingebunden.

Da Paintareas mit einem Cache arbeiten, existiert eine *FIX*-interne Struktur und eine externe Struktur. Die externe Struktur dient zur Übergabe der Daten an die Funktion des Interfaces und zum Auslesen von Werten über Funktionen des Interfaces. Sie hat folgenden Aufbau:

```
typedef struct {
        short pa_type;                  /* type of paintarea */
        obj* pa_objp;                   /* object which contains the paintarea */
        short pa_variant;               /* variant which contains the paintarea */
        int pa_pos_y;
        int pa_pos_x;                   /* position in object */
        char* pa_token;                 /* token of the area */
        int pa_len;                     /* length of the area */
        char* pa_text;                  /* translated text of the area */
        char* pa_tooltiptext;           /* text of the tooltip */
        unsigned char pa_attr;          /* video attribute */
        unsigned char pa_align;         /* alignment of text */
        unsigned char pa_bt_mask;       /* possible mouse buttons */
        PrivatePropertyType pa_props;   /* private properties */
} paintarea;
```

Je nach Funktion und Typ werden alle oder nur einige der Komponenten der Struktur genutzt. Die Komponente pa_type beinhaltet den Typ der Paintarea. Als Wert kommen folgende Konstanten in Betracht:

- PA_TEXTLABEL - ein Textlabel.
- PA_TABLEHEADER - ein Text als Kopf einer Tabellenspalte.
- PA_BITMAP - eine Bitmap. Dieser Wert gilt als reserviert. Bitmaps werden von *FIX/Win* zur Zeit noch nicht unterstützt.
- PA_BUTTON - ein Button.
- PA_VARBUTTON - ein Button zur Umschaltung von Varianten.
- PA_USERDEFINED - benutzerdefiniert. Alle Paintareas, die diesen Wert oder einen größeren Wert benutzen, werden von *FIX/Win* nicht dargestellt. Die Darstellung bleibt der Benutzerbibliothek von *FIX/Win* überlassen.

Die Werte pa_objp und pa_variant definieren das Objekt und die Nummer der Variante (0-n) des Objekts, in der die Paintarea platziert wird. Die Position innerhalb des Objekts steht in pa_pos_y (Zeile) und pa_pos_x (Spalte).

In der Komponente pa_token wird ein Kurztext hinterlegt. Dieser Text wird in den virtuellen Bildschirm geschrieben und als Basis für die Übersetzungsfunktion verwendet. Da der Text einer Paintarea durch die Verwendung von Proportionalschrift häufig wesentlich mehr Zeichen umfassen kann, kann die Übersetzungsfunktion zu einem kurzen Text einen längeren liefern. Die sprach- oder umgebungsabhängige Übersetzung ist ein Nebeneffekt dieser Funktion. Der Wert pa_len enthält die Länge in Zeichen, die die Paintarea im Layout belegt.

Durch das Belegen der Komponente pa_text kann der Aufruf der Übersetzungsfunktion umgangen werden. Wenn die Komponente einen anderen Wert als NULL besitzt, wird dieser Wert als übersetzter Text zur Darstellung in der Paintarea verwendet.

Wenn die Paintarea einen Tooltip besitzen soll, dann ist dessen Text in pa_tooltiptext zu hinterlegen. Soll die Paintarea keinen Tooltip besitzen, dann ist dort eine leere Zeichenkette einzutragen. Alternativ ist die Angabe von NULL zulässig. Dieser Wert wird jedoch vor der Verwendung von *FIX* in eine leere Zeichenkette umgesetzt.

Das Videoattribut, mit dem der entsprechende Bereich im Layout gezeichnet wird, ist in der Komponente pa_attr zu hinterlegen.

In der Komponenten pa_align kann die Ausrichtung des Textes definiert werden. Folgende Werte sind zulässig:

- PA_ALGN_LEFT - Der Text wird linksbündig ausgerichtet.
- PA_ALGN_CENTER - Der Text wird zentriert.
- PA_ALGN_RIGHT - Der Text wird rechtsbündig ausgerichtet.

Die Komponente pa_bt_mask bestimmt die Anklickbarkeit einer Paintarea. Als Wert kann eine Kombination (oder-Verknüpfung) der folgenden Werte verwendet werden:

- PA_BT_LEFT - Die Paintarea kann mit der linken Maustaste angeklickt werden.
- PA_BT_MIDDLE - Die Paintarea kann mit der mittleren Maustaste angeklickt werden.
- PA_BT_RIGHT - Die Paintarea kann mit der rechten Maustaste angeklickt werden.

Wenn die Paintarea nicht anklickbar sein soll, dann kann für pa_bt_mask der Wert PA_BT_NONE verwendet werden.

Die Werte in pa_props können für eigene Zwecke genutzt werden.

Zum Zugriff auf die interne Struktur definiert *FIX* ein Handle vom Typ

```
HPAINTAREA
```

Ein Handle dieses Typs wird mit einer bestimmten Paintarea im Cache gleichgesetzt. Die Funktionen des Interfaces nutzen ein Handle zum schnellen Zugriff auf dessen Cache-Eintrag. Da ein Cache-Eintrag mit dem Entfernen des Objekts vom Bildschirm (disappear) freigegeben wird, werden damit auch alle Handles ungültig, die auf diesen Eintrag verweisen. *FIX* verwendet zusammen mit den Cache-Einträgen auch die Werte für die Handles wieder, so dass es vorkommen kann, dass ein altes Handle auf eine andere Paintarea zeigt. Deshalb dürfen die Handles aller Paintareas eines Objektes nach dem Entfernen des Objekts vom Bildschirm nicht mehr benutzt werden.

Ein Handle kann zwei spezielle Werte annehmen, die als Makros in `fixwin.h` definiert werden:

```
HPAINTAREA_INVALID
```

dient als Fehlerwert, der von einer Funktion zurückgegeben wird, wenn ein Fehler aufgetreten ist.

```
HPAINTAREA_EMPTY
```

definiert eine leere Paintarea und wird zum Beispiel dann zurückgegeben, wenn keine Paintarea gefunden werden konnte. Wenn diese Werte als Ergebnis oder Argument in Betracht kommen, dann enthält die Beschreibung der jeweiligen Funktion eine entsprechende Anmerkung.

2 Erzeugen von Paintareas

Paintareas werden unmittelbar vor dem Anzeigen des Objektes erzeugt und nach dem Entfernen des Objektes vom Bildschirm als ungültig markiert. Zur Verminderung des Netzwerkverkehrs wird mit einem Cache gearbeitet, so dass bei der erneuten Anzeige des Objekts die in *FIX* und *FIX/Win* bereits bestehenden Strukturen wiederverwendet werden können. Der Grund, dass Paintareas beim Anzeigen und nicht beim Erzeugen des Objektes angelegt und beim Entfernen vom Bildschirm freigegeben werden, liegt darin, dass viele Anwendungen die Objekte nicht unmittelbar nach der

Benutzung freigeben. *FIX/Win* müsste in diesem Fall die Daten zu den Paintareas aller geladenen Masken halten, was den Speicherbedarf von *FIX/Win* stark erhöhen würde.

Zum Erzeugen von Paintareas für ein Objekt wird unmittelbar vor der Anzeige die in der Variablen

```
extern void (*S_pa_mark)(obj *)
```

hinterlegte Funktion aufgerufen.

Zur Anlage einer Paintarea ist ein Bereich im Layout des Objektes zu markieren und ein Eintrag im Paintarea-Cache vorzunehmen. Dies ist über zwei unterschiedliche Verfahren möglich:

- Definition durch die Analyse des Layouts eines Objektes.
- Definition durch das Schreiben in das Layout eines Objektes.

In ein und der selben Anwendung können beide Verfahren verwendet werden.

Definition durch die Analyse des Layouts

Bei der Definition durch die Analyse des Layouts eines Objektes sind die Bereiche des Layouts, die als Paintareas dienen sollen mit dem Attribut PAINTAREA zu markieren. Damit das ursprüngliche Attribut nicht verloren geht, muss das neue Attribut aus einer oder-Verknüpfung von PAINTAREA und dem alten Attribut bestehen. Die Verwendung von verschiedenen Attributen für die einzelnen Zellen einer Paintarea ist dabei nicht möglich. Danach muss die Funktion

```
HPAINTAREA pa_declare(paintarea* pa)
```

zur Definition der übrigen Daten der Paintarea aufgerufen werden. Sie legt wenn notwendig einen neuen Cache-Eintrag an oder verwendet einen alten wieder. Dabei werden bis auf eine Ausnahme alle Werte der Struktur, auf die pa zeigt, in den Cache-Eintrag übertragen.

Der Wert, der in der Komponenten pa->pa_variant steht, wird beim Übertragen in den Cache ignoriert. Statt dessen wird die aktuelle Variante der Maske verwendet. Denn das ist die Variante, die gerade dargestellt werden soll und für die die Funktion in S_pa_mark aufgerufen wird. Wenn für das Objekt keine Varianten möglich sind (also wenn es einen anderen Objekttyp als MASK, SUBMASK, ROLLMASK oder TABLE besitzt), dann wird der Wert -1 für die aktuelle Variante verwendet. Die Komponente pa_variant der Struktur paintarea dient nur zum Auslesen der Variantennummer eines Cache-Eintrags mittels pa_get() (siehe weiter unten).

Stimmen bei der Wiederverwendung eines Cache-Eintrags die neuen Werte mit den alten überein, dann werden die Daten nicht nochmals an*FIX/Win* übertragen. Nur wenn ein neuer Cache-Eintrag erzeugt wird, oder wenn die Werte eines bestehenden Eintrags verändert werden, dann werden die Werte an*FIX/Win* übertragen. Dabei wird als Text nicht der Wert in pa->pa_token verwendet. Stattdessen wird ein übersetzter Text, der wesentlich länger sein darf als der Text in pa_token, an *FIX/Win* übertragen.

Der Text wird entweder aus der Komponenten pa_text gelesen oder wenn diese Komponente den Wert NULL besitzt, durch Aufruf der Übersetzungsfunktion ermittelt. Der Zeiger auf diese Funktion ist in der Variablen

```
extern void (*S_pa_translate_token)(paintarea* pa, char* transText)
```

zu hinterlegen. Die Funktion bekommt den Parameter pa als erstes Argument. Sie hat die Aufgabe aufgrund dieses Parameters, der auch den Kurztext in der Komponente pa_token enthält, einen Text für *FIX/Win* zu ermitteln und in dem Speicherbereich abzulegen, auf den der Parameter transText zeigt. Die Länge des Textes darf dabei höchstens PA_MAX_TEXTLEN (=256) Bytes betragen.

Wird in der Variablen S_pa_translate_token keine Funktion hinterlegt (und ist pa_text == NULL), dann wird der in pa->pa_token vorhandene Text an *FIX/Win* gesendet.

Wenn die Markierungsfunktion zurückkehrt und alle Paintareas definiert hat, stellt *FIX* das Objekt auf dem Bildschirm dar. Dabei werden alle mit dem Attribut PAINTAREA markierten Flächen im Layout des Objektes als Paintarea-Bereiche an *FIX/Win* übertragen. *FIX/Win* ist damit in der Lage, zur Darstellung dieser Flächen die entsprechenden Zeichenfunktionen aufzurufen. Die Verknüpfung zu den mit pa_declare() definierten Daten wird dabei auf der Seite von *FIX* über das Objekt, die Variantennummer, die Position und die Länge einer Paintarea hergestellt.

Definition durch das Schreiben in das Layout

Zur Definition einer Paintarea durch das Schreiben in das Layout eines Objektes ist die Funktion

```
HPAINTAREA pa_put(paintarea* pa)
```

aufzurufen. Diese Funktion kann sowohl innerhalb der Funktion zur Markierung, als auch nach dem Anzeigen des Objekts aufgerufen werden. Deshalb wird die Paintarea sowohl im Layout des Objektes als auch in dem Teil des virtuellen Bildschirm angelegt, der von dem Objekt belegt wird. Als Text wird dabei der in pa->pa_token hinterlegte Text verwendet. Ist der Text kürzer als in pa->pa_len angegeben, dann wird der Rest des Bereichs mit Leerzeichen aufgefüllt. Das Textattribut wird aus pa->pa_attr gelesen. Nach dem Übertragen des Textes in das Layout und auf den virtuellen Bildschirm wird ein Eintrag im Paintarea-Cache durch Aufruf der Funktion pa_declare() angelegt. Dabei wird der Text, wie oben beschrieben, durch die Übersetzungsfunktion umgesetzt und an *FIX/Win* gesendet.

Befindet sich in dem durch pa->pa_pos_y, pa->pa_pos_x und pa->pa_len definierten Bereich bereits eine andere (eventuell überlappende) Paintarea, dann wird diese vor der Anlage der neuen Paintarea entfernt.

Wird als pa->pa_token der Wert NULL angegeben, dann wird geprüft, ob an der Position pa->pa_pos_y, pa->pa_pos_x bereits der Teil einer anderen Paintarea steht. Wenn ja, wird diese komplett entfernt. Die Anlage einer neuen Paintarea unterbleibt. Damit besteht die Möglichkeit, durch Aufruf von pa_put() mit pa->pa_token=NULL bestehende Paintareas zu löschen.

Neben den Funktionen pa_declare() und pa_put() können Paintareas auch mit einer der Funktionen

```
HPAINTAREA pa_declare_type(short type, obj *objp, int pos_y, int pos_x,
    char* token, char* text, char* tooltiptext, int attr, int align, int bt_mask,
    long longval1, long longval2, long longval3, long longval4, char* ptrval1, char* ptrval2)

HPAINTAREA pa_put_type(short type, obj *objp, int pos_y, int pos_x,
    char* token, char* text, char* tooltiptext, int attr, int align, int bt_mask,
    long longval1, long longval2, long longval3, long longval4,char* ptrval1, char* ptrval2)
```

definiert werden. Diese Funktionen bekommen statt eines Zeigers auf eine paintarea Struktur die einzelnen Werte für die Komponenten. Sie erzeugen intern eine pa-Struktur und kopieren die Werte in die entsprechenden Komponenten. Die Länge der Paintarea (pa_len) wird dabei aus der Länge der Zeichenkette token gebildet. Danach rufen sie pa_declare() bzw. pa_put() auf.

Zum Erzeugen einer Paintarea eines bestimmten Typs, werden in `fixwin.h` Makros definiert, die den Parameter type belegen, die übrigen Parameter durchreichen und mit der so gebildeten Parameterliste die Funktion pa_declare_type() oder pa_put_type() aufrufen. Im Namen des Makros wird dabei type durch den Namen des Typs ersetzt. Dadurch ergeben sich folgende Macros:

```
pa_declare_textlabel(objp, pos_y, pos_x, token, text, tooltiptext, attr, align, bt_mask,
    longval1, longval2, longval3, longval4, ptrval1, ptrval2)

pa_declare_tableheader(objp, pos_y, pos_x, token, text, tooltiptext, attr, align, bt_mask,
    longval1, longval2, longval3, longval4, ptrval1, ptrval2)

pa_declare_bitmap(objp, pos_y, pos_x, token, text, tooltiptext, attr, align, bt_mask,
    longval1, longval2, longval3, longval4, ptrval1, ptrval2)

pa_declare_button(objp, pos_y, pos_x, token, text, tooltiptext, attr, align, bt_mask,
    longval1, longval2, longval3, longval4, ptrval1, ptrval2)

pa_put_textlabel(objp, pos_y, pos_x, token, text, tooltiptext, attr, align, bt_mask,
    longval1, longval2, longval3, longval4, ptrval1, ptrval2)

pa_put_tableheader(objp, pos_y, pos_x, token, text, tooltiptext, attr, align, bt_mask,
    longval1, longval2, longval3, longval4, ptrval1, ptrval2)
```

```
pa_put_bitmap(objp, pos_y, pos_x, token, text, tooltiptext, attr, align, bt_mask,
    longval1, longval2, longval3, longval4, ptrval1, ptrval2)

pa_put_button(objp, pos_y, pos_x, token, text, tooltiptext, attr, align, bt_mask,
    longval1, longval2, longval3, longval4, ptrval1, ptrval2)
```

3 Beschaffen von Informationen zu einer Paintarea

Die Funktionen pa_declare() und pa_put() liefern ein Handle auf einen Eintrag des Paintarea-Caches. Durch Aufruf der Funktion

```
BOOLEAN pa_get(HPAINTAREA h_pa, paintarea* pa)
```

können die Werte aus diesem Cache-Eintrag gelesen werden. h_pa enthält das Handle des Caches und pa zeigt auf eine Struktur, in der die Informationen abgelegt werden. Danach kehrt die Funktion mit dem Rückgabewert TRUE zurück. Ist der zugeordnete Cache-Eintrag ungültig geworden, weil das Objekt vom Bildschirm entfernt wurde, dann liefert die Funktion FALSE zurück. Es ist jedoch auch möglich, dass sie TRUE zurückliefert, obwohl das Objekt vom Bildschirm entfernt wurde. Dieser Fall kommt dann vor, wenn der Cache-Eintrag mittlerweile durch ein anderes Element belegt wird. Wie bei der Beschreibung der Handles bereits erwähnt, dürfen die Handles der Paintareas eines Objekts nach dem Entfernen des Objekts vom Bildschirm nicht mehr benutzt werden.

Der Speicherbereich an den Komponenten pa_token, pa_text und pa_tooltiptext wurde von *FIX* passend zur Größe des Inhalts allokiert. Diese Speicherbereiche dürfen nicht verändert werden, da sie auf interne Strukturen von *FIX* verweisen. Wenn diese Werte geändert werden sollen, dann muss den entsprechenden Komponenten ein Zeiger auf einen eigenen Speicherbereich zugewiesen werden.

Wenn das Handle zu einem Eintrag nicht bekannt ist, dann kann es ermittelt werden. Dazu können alle Handles zu den Einträgen eines Objektes ermittelt werden, die bestimmten Anforderungen entsprechen. Hierzu ist die Funktion

```
HPAINTAREA pa_next(HPAINTAREA h_paStart, paintarea* pa_check, unsigned long validmask)
```

zu verwenden. Sie ermittelt zu dem Handle in h_paStart das nächste Handle des Eintrags, dessen Werte mit denen in pa_check übereinstimmen. Dabei werden nicht alle Werte in pa_check geprüft. Der Wert für validmask bestimmt, welche Werte geprüft werden. Er ist durch eine oder-Verknüpfung aus den folgenden Werten zu bilden:

- PA_TYPE - Der Wert in pa_check->pa_type wird auf Übereinstimmung geprüft.
- PA_POS - Die Position wird auf Überlappung geprüft. Der in Frage kommende Eintrag muss in seinem Bereich eine Zelle besitzen, die an der Position pa_check->pa_pos_y, pa_check->pa_pos_x liegt.
- PA_TOKEN - Der Wert in pa_check->pa_token wird auf Übereinstimmung geprüft.
- PA_ATTR - Der Wert in pa_check->pa_attr wird auf Übereinstimmung geprüft.
- PA_ALIGN - Der Wert in pa_check->pa_align wird auf Übereinstimmung geprüft.
- PA_BT_MASK - Der Wert in pa_check->pa_bt_mask wird auf Übereinstimmung geprüft.
- PA_LONGVAL1 - Der Wert in pa_check->pa_props.longval1 wird auf Übereinstimmung geprüft.
- PA_LONGVAL2 - Der Wert in pa_check->pa_props.longval2 wird auf Übereinstimmung geprüft.
- PA_LONGVAL3 - Der Wert in pa_check->pa_props.longval3 wird auf Übereinstimmung geprüft.
- PA_LONGVAL4 - Der Wert in pa_check->pa_props.longval4 wird auf Übereinstimmung geprüft.
- PA_PTRVAL1 - Der Wert in pa_check->ptrval1 wird auf Übereinstimmung geprüft.
- PA_PTRVAL2 - Der Wert in pa_check->ptrval2 wird auf Übereinstimmung geprüft.

Soll auf keinen der Werte geprüft werden, dann ist PA_NONE als Wert für den Parameter validmask zu verwenden. Darüber hinaus werden nur Paintareas berücksichtigt, die sich in der aktuellen (und damit zur Zeit angezeigten) Variante des Objektes in pa_check->pa_objp befinden.

Wenn zu Beginn kein Handle für h_paStart bekannt ist, dann kann HPAINTAREA_EMPTY angegeben werden. In diesem Fall beginnt die Suche bei dem ersten Cache-Eintrag des Objekts. Wird ein konkreter Wert für h_paStart angege-

ben, dann beginnt die Suche hinter dem Eintrag, der durch h_paStart definiert wird. Wird ein passender Eintrag gefunden, dann wird das Handle zu diesem Eintrag zurückgeliefert. Bei einem Fehler liefert die Funktion HPAINTAREA_INVALID zurück. Wenn kein weiterer Eintrag gefunden wird, der zu den Bedingungen passt, dann wird HPAINTAREA_EMPTY zurückgeliefert. Die Reihenfolge, in der pa_next() die gefundenen Handles liefert ist zufällig.

Beispiel

Der folgende Code gibt die Positionen aller Buttons in der Maske AUFTR_mskp aus:

```
HPAINTAREA h_pa;
paintarea pa_check;
paintarea pa_result;

pa_check.pa_objp = (obj*)AUFTR_mskp;
pa_check.pa_type = PA_BUTTON;

h_pa = HPAINTAREA_EMPTY;
while((h_pa = pa_next(h_pa, &pa_check, PA_TYPE)) != HPAINTAREA_EMPTY) {
    if (h_pa == HPAINTAREA_INVALID) {
        rt_msg(ACHTUNG, "Fehler");
        break;
    }
    pa_get(h_pa, &pa_result);
    rt_msg(ACHTUNG, "Button %s at %d,%d",
        pa_result.pa_token,
        pa_result.pa_pos_y,
        pa_result.pa_pos_x);
}
```

4 Aktualisieren einer Paintarea

Durch Aufruf der Funktion

```
BOOLEAN pa_update(HPAINTAREA h_pa, paintarea *pa, unsigned long validmask)
```

können die Daten einer Paintarea aktualisiert werden. In h_pa ist dazu das Handle des Cache-Eintrags anzugeben. pa zeigt auf eine Struktur mit den neuen Daten. Dabei werden nicht alle Daten dieser Struktur genutzt. Der Wert in validmask bestimmt, welche Daten verwendet werden. Er ist durch eine oder-Verknüpfung aus den folgenden Werten zu bilden:

- PA_TYPE - Der Typ wird auf pa->type geändert.
- PA_TOKEN - Der Wert in pa->pa_token wird als neuer Kurztext verwendet. Dadurch ändert sich jedoch nicht automatisch der übersetzte Text. Es wird der bei pa_declare() festgesetzte Text verwendet. Wenn der übersetzte Text geändert werden soll, dann ist zusätzlich der Wert PA_TEXT anzugeben.
- PA_TEXT - Der Wert in pa->pa_text wird als neuer Text verwendet. Beinhaltet die Komponente den Wert NULL, dann wird der Text durch Aufruf der Übersetzungsfunktion ermittelt.
- PA_TOOLTIPTEXT - Der Wert in pa->pa_tooltiptext wird als neuer Text für den Tooltip verwendet.
- PA_ATTR - Der Wert in pa->pa_attr wird als neues Bildschirmattribut verwendet.
- PA_ALIGN - Der Wert in pa->pa_align wird übernommen.
- PA_BT_MASK - Der Wert in pa->pa_bt_mask wird übernommen.
- PA_LONGVAL1 - Der Wert in pa->pa_props.longval1 wird übernommen.

- PA_LONGVAL2 - Der Wert in pa->pa_props.longval2 wird übernommen.
- PA_LONGVAL3 - Der Wert in pa->pa_props.longval3 wird übernommen.
- PA_LONGVAL4 - Der Wert in pa->pa_props.longval4 wird übernommen.
- PA_PTRVAL1 - Der Wert in pa->ptrval1 wird übernommen.
- PA_PTRVAL2 - Der Wert in pa->ptrval2 wird übernommen.

Nach der Übernahme der Werte in den Cache-Eintrag, wird geprüft, ob das Layout und der virtuelle Bildschirm von der Änderung betroffen sind. Dies ist bei der Angabe von PA_TOKEN oder PA_ATTR der Fall. Die geänderten Werte werden dann ins Layout und in den virtuellen Bildschirm übernommen. Bei der Angabe von PA_TYPE, PA_TOKEN, PA_TEXT, PA_ALIGN, PA_TOOLTIPTEXT oder PA_LONGVAL1, PA_LONGVAL2, PA_LONGVAL3, PA_LONGVAL4 müssen die neuen Werte auch an *FIX/Win* übertragen werden. Bei der Angabe von PA_TEXT wird zur Bestimmung des Textes die Übersetzungsfunktion aufgerufen, wenn eine solche definiert wurde und der übersetzte Text nicht in der Komponenten pa_text mitgegeben wurde.

Beispiel

Der folgende Code setzt das Attribut aller Buttons in der Maske AUFTR_mskp auf GRAYED:

```
HPAINTAREA h_pa;
paintarea pa_check;
paintarea pa_result;

pa_check.pa_objp = (obj*)AUFTR_mskp;
pa_check.pa_type = PA_BUTTON;

h_pa = HPAINTAREA_EMPTY;
while((h_pa = pa_next(h_pa, &pa_check, PA_TYPE)) != HPAINTAREA_EMPTY) {
    if (h_pa == HPAINTAREA_INVALID) {
        rt_msg(ACHTUNG, "Fehler");
        break;
    }
    pa_get(h_pa, &pa_result);
    pa_result.pa_attr = GRAYED;
    pa_update(h_pa, &pa_result, PA_ATTR);
}
```

5 Löschen einer Paintarea

Zum Löschen einer Paintarea ist die Funktion

```
BOOLEAN pa_delete(HPAINTAREA h_pa)
```

aufzurufen. Sie löscht die Paintarea, indem sie den entsprechenden Bereich mit Leerzeichen im Attribut NORMAL überschreibt. Danach liefert sie TRUE zurück. Wenn h_pa keinem gültigen Cache-Eintrag zugeordnet ist, dann liefert die Funktion FALSE zurück.

6 Freigeben von Paintareas

Paintareas werden bis auf eine Ausnahme dann freigegeben, wenn das zugehörige Objekt vom Bildschirm entfernt wird. Sie werden beim Darstellen des Objektes wieder durch Aufruf von (*S_pa_mark)() erzeugt. Dies führt zu Proble-

men bei Maskenvarianten, da nach einem Wechsel zu einer anderen Variante und wieder zurück alle mit pa_put_... erzeugten Paintareas nicht mehr vorhanden sind.

Das Freigeben der Paintareas beim Entfernen des Objekts vom Bildschirm hatte folgenden Grund: Viele Anwendungen geben die Masken bewußt nicht mit o_free() frei. Damit werden dann auch die Paintareas nicht freigegeben, wodurch es schnell zu einem Überlauf des Caches kommen kann.

Da das Umschalten von Varianten ein eher temporäres Entfernen vom Bildschirm ist, wurde in *FIX* folgendes Verhalten zur Behebung des Problems implementiert:

- Die Paintareas der aktuellen Variante werden beim Aufblenden der Maske erzeugt, die diese Variante enthält.
- Beim Wechsel auf eine andere Variante werden die Paintareas der aktuellen Variante nicht freigegeben.
- Die Paintareas der neuen Variante werden nur dann erzeugt, wenn die Variante seit dem Aufblenden der Maske noch nicht sichtbar war, also beim ersten Anzeigen der Variante. Beim zweiten Anzeigen der Variante werden die bestehenden Paintareas verwendet, da sie beim Wechsel nicht freigegeben wurden.
- Beim Entfernen der Maske vom Bildschirm werden die Paintareas aller Varianten freigegeben.

Dieses Verhalten hat zur Folge, dass die Funktion (*S_pa_mark)() nicht mehr bei jedem Variantenwechsel aufgerufen wird, sondern nur noch beim ersten Wechsel auf eine Variante nach dem Anzeigen der Maske.

7 Mausbedienbarkeit von Paintareas

Generell sind alle Typen von Paintareas mit der Maus anklickbar. Der Wert in der Komponente pa_bt_mask bestimmt, ob und mit welcher Maustaste eine Paintarea angeklickt werden kann. Beim Anklicken wird von *FIX/Win* je nach Maustaste eines der Events

- BT_LEFT_PA - Paintarea wurde mit linker Maustaste angeklickt.
- BT_MIDDLE_PA - Paintarea wurde mit mittlerer Maustaste angeklickt.
- BT_RIGHT_PA - Paintarea wurde mit rechter Maustaste angeklickt.

an *FIX* gesendet. *FIX* versucht daraufhin die Felderfassung abzuschließen. Wenn das gelingt, dann wird das Event an die Anwendungslogik weitergegeben. Die Anwendung kann dann mit Hilfe der Funktion

```
HPAINTAREA pa_get_clicked(void)
```

ein Handle der Paintarea ermitteln, die angeklickt wurde. Mit pa_get() können weitere Informationen zu der Paintarea ausgelesen werden. Soll mit dem Anklicken eine bestimmte Aktion verbunden werden, dann ist es sinnvoll, bei der Definition einer anklickbaren Paintarea einen Wert in der Komponenten longval1, longval2, longval3, longval4, ptrval2 oder ptrval1 zu hinterlegen, in der die Aktion kodiert wird. longval1 könnte beispielsweise die ID einer Aktion enthalten oder ptrval1 könnte einen Funktionszeiger enthalten. Die konkrete Realisierung wird der Anwendung überlassen. Bei der Behandlung des BT_.._PA Events kann die Information ausgewertet werden.

Bei der Mausbedienbarkeit hat der Typ PA_BUTTON eine spezielle Bedeutung. Bei diesem Typ besteht die Möglichkeit, dem Benutzer eine Bedienbarkeit zu visualisieren. Dies wird dadurch erreicht, dass je nach Zustand von *FIX/Win* ein anderes Bildschirmattribut verwendet wird und der Button von der Benutzer-DLL (oder *FIX/Win*) anderes gezeichnet werden kann. Dabei wird zwischen den Zuständen "Normal", "Mauszeiger über der Paintarea" und "Button gedrückt" unterschieden. Da damit dem Benutzer schon vor dem Anklicken visualisiert wird, dass beim Anklicken eine Aktion ausgeführt wird, muss *FIX/Win* wissen, in welchen Objekten Paintareas zu einem bestimmten Zeitpunkt anklickbar sind. Wird beispielsweise ein Selo aufgeblendet, dann darf ein Button in der darunter liegenden Maske nicht in dem Zustand "Mauszeiger über der Paintarea" angezeigt werden. Weiterhin darf kein Event an *FIX* gesendet werden (das gilt für alle Typen von Paintareas), wenn die Paintarea angeklickt wird. Andererseits muss es möglich sein, einen Button in einer eingebetteten Maske anklicken zu können, auch wenn die übergeordnete Maske gerade den Eingabefokus besitzt. In diesem Fall wird das Event zwar an die Anwendungslogik der übergeordneten Maske gesendet, jedoch kann auch diese Anwendungslogik so beschaffen sein, dass sie das Event verarbeiten kann. Möglicherweise wird das Event bereits von der globalen Anwendungslogik verarbeitet. In diesem Fall muss der Button in der eingebetteten Maske den Zustand "Mauszeiger über der Paintarea" annehmen können und beim Anklicken ein Event senden, auch wenn die eingebettete Maske nicht aktiv ist. Da weder *FIX* noch *FIX/Win* den Aufbau der Anwendung kennen und wissen, ob

sie das Anklicken von Paintareas in nicht aktiven Masken behandelt, muss die Anwendung *FIX/Win* mitteilen, in welchen Masken Paintareas angeklickt werden können. Dazu ist die Funktion

```
void set_activeobj_list(obj* objp, obj** obj_list)
```

aufzurufen. Sie definiert für das Objekt objp eine Liste von zusätzlichen Objekten, die als aktiv gelten sollen, wenn die Maske objp aktiv ist. Die Liste ist als Array in obj_list zu übergeben. Das letzte Element des Arrays muss als Endemarkierung den Wert NULL besitzen. Dabei ist zu beachten, dass *FIX* sich den übergebenen Zeiger auf das Array zu dem Objekt merkt und wiederverwendet (zB. bei einem erneuten perform() Aufruf). Deshalb genügt es, die Liste einmal (zB. nach dem Laden der Maske) zu definieren. Das Array sollte auf keinen Fall als automatische Variable auf dem Stack angelegt werden oder für andere Objekte umdefiniert werden. Erfolgt für ein Objekt kein Aufruf von set_activeobj_list() oder wird der Wert obj_list mit NULL übergeben, dann ist die Liste leer und die Buttons in einem Objekt sind nur anklickbar, wenn das Objekt selbst aktiv ist.

Befindet sich eine Paintarea in einem Objekt, das in der Liste angegeben wurde, dann ist sie auch anklickbar, wenn die Maske mskp aktiv ist und das Objekt selbst nicht. Handelt es sich dabei um eine Paintarea vom Typ PA_BUTTON, dann wird sie in dem Zustand "Mauszeiger über der Paintarea" dargestellt, wenn sich der Mauszeiger im Bereich der Paintarea befindet.

Über diesen Mechanismus hinaus, ist eine Paintarea nur dann anklickbar, wenn sie nicht das Attribut GRAYED besitzt und wenn die Komponente pa_bt_mask einen Wert ungleich PA_BT_NONE besitzt. Das Attribut GRAYED ist dazu gedacht, eine bestimmte Paintarea (und nicht alle Paintareas des Objekts) zu deaktivieren (im Sinne nicht anklickbar) und dies dem Benutzer auch zu visualisieren.

Beispiel

```
static Event AUFTR_event_control(obj *objp, Event event)
{
    switch (event) {
    ...
    case BT_LEFT_PA:
        hClickedButton = pa_get_clicked();
        pa_get(hClickedButton, &paTemp1);
        rt_msg(PROTO, "BT_LEFT_PA: %s", paTemp1.pa_token);
        break;
    ...
    }
}

static obj* list[2];

int AUFTRb_proc(void)
{
    ...
    list[0] = AUFTR_mskp;
    list[1] = NULL;
    set_activeobj_list(POS_mskp, list);
    ...
}
```

Beim Anklicken einer Paintarea mit der linken Maustaste (Event BT_LEFT_PA) in der Auftragsmaske, wird der Text der Paintarea über rt_msg() ausgegeben. Der Aufruf von set_activeobj_list() nach dem Laden der Positionsmaske POS_mskp und der Auftragsmaske AUFTR_mskp sorgt dafür, dass die Buttons in AUFTR_mskp auch dann aktiv sind, wenn die Maske POS_mskp das aktuelle Objekt ist. Damit auf das Anklicken der Buttons reagiert wird, muss die Behandlung des Events BT_LEFT_PA auch in die Anwendungslogik der Maske POS_mskp oder in die globale Anwendungslogik aufgenommen werden.

8 Konfiguration des Paintarea-Caches

Um den Netzwerkverkehr zwischen *FIX* und *FIX/Win* zu reduzieren, arbeiten Paintareas mit einem Cache. Der Cache besteht aus einem Array, bei dem jeder Eintrag Informationen zu einem *FIX*-Objekt (Maske, Selo, ..) und einen Zeiger auf ein Array mit Informationen zu den Paintareas enthält. Beim Anzeigen (map) des Objektes wird ein Arrayeintrag belegt. Beim Entfernen (unmap) wird der Eintrag als frei markiert. Bei nochmaligem Anzeigen wird versucht, den eventuell noch vorhandenen Eintrag zu nutzen, wenn dieser noch nicht von einem anderen *FIX*-Objekt belegt wurde. Beim Neubelegen eines Eintrags werden zuerst die noch nie belegten Einträge verwendet. Erst wenn von diesen keine mehr vorhanden sind, wird einer der als frei markierten Einträge verwendet. Dazu wird ein Zähler mitgeführt, der bei jeder Benutzung erhöht wird. Der am wenigsten benutzte und noch freie Eintrag wird verwendet.

Bei der Wiederverwendung eines Eintrags wird versucht, die einzutragende Paintarea anhand ihrer Parameter wiederzufinden. Gelingt dies, dann wird der Eintrag als verwendet markiert. Da die Parameter bereits bei der letzten Verwendung an *FIX/Win* gesendet wurden, unterbleibt das Senden der Parameter über das Netzwerk.

Als Vergleichswert wird u.a. der nicht übersetzte Text (pa_token) verwendet. Nur wenn dieser sich unterscheidet, dann wird der Text durch Aufruf der Übersetzungsfunktion übersetzt. Der Cache wirkt sich also nicht nur auf den Netzwerkverkehr aus, sondern minimiert auch den Übersetzungsaufwand. Diese Vorgehensweise setzt jedoch voraus, dass zu einem Text in einer bestimmten Maske bei erneuter Anzeige der gleiche Text als Übersetzung geliefert wird, wie bei der vorherigen Anzeige. Es ist also z.B. nicht möglich, die Maske auszublenden, die Sprache der Übersetzungsroutine umzudefinieren und die Maske wieder (in der dann anderen Sprache) einzublenden.

Parameter

```
int S_pa_max_tabs
```

Diese globale Variable bestimmt die Größe des Arrays mit den Einträgen zu den *FIX*-Objekten. Je größer S_pa_max_tabs ist, desto mehr Speicher wird verwendet und desto größer ist die Wahrscheinlichkeit, dass ein bereits vorhandener Eintrag wieder verwendet werden kann. Die Mindestgrenze für S_pa_max_tabs ergibt sich aus der maximalen Anzahl der gleichzeitig angezeigten Objekte einer Anwendung.

Die Variable sollte möglichst frühzeitig im Programm gesetzt werden, spätestens jedoch vor dem Anzeigen des ersten *FIX*-Objektes. Danach darf der Wert nicht mehr verändert werden. Der Defaultwert für diesen Parameter ist 20.

```
int S_pa_tab_size
```

Diese globale Variable bestimmt die Anzahl der Paintareas pro Objekt. Für diese Anzahl wird beim Belegen eines Eintrags Speicher angefordert. Übersteigt die Anzahl der Paintareas eines Objektes diesen Wert, dann wird ein weiterer Block Speicher dieser Größe angefordert. Es kann bei einem zu kleinen Wert also nicht zu einem Überlauf der Tabelle kommen, sondern höchstens zu immer wieder neuen Speicheranforderungen auf Kosten der Performance.

Die Variable sollte möglichst frühzeitig im Programm gesetzt werden, spätestens jedoch vor dem Anzeigen des ersten *FIX*-Objektes. Danach darf der Wert nicht mehr verändert werden. Der Defaultwert für diesen Parameter ist 20.

9 Ausgabe von Informationen zu Debugzwecken

Mit Hilfe der Funktion

```
void pa_dump_for_obj(FILE *fout, obj* objp)
```

können alle Paintareas zu dem Objekt objp auf dem Kanal fout ausgegeben werden. Wenn das Objekt zur Zeit nicht sichtbar ist und der Cache-Eintrag ungültig ist, dann wird eine Meldung ausgegeben und es erfolgen keine weiteren Ausgaben. Ansonsten wird für die aktuelle Variante des Objekts eine Tabelle ausgegeben. Im Kopf der Tabelle wird die ID des Objekts, der Name und die Nummer der Variante angegeben.

Darauf folgt eine Zeile für jede Paintarea dieser Variante mit deren Daten. Die Angabe der Werte attr, longval1, longval2, longval3 und longval4 in der ausgegebenen Tabelle erfolgt hexadezimal. Die übrigen Zahlenwerte werden dezimal angegeben. Da die Angabe von ptrval1 und ptrval2 als Zahl keine Aussagekraft hat, kann in der Variablen

```
extern char* (*S_pa_ptrval_text)(char *ptrval, int pttype)
```

eine Funktion hinterlegt werden, die zu dem Wert einen Text zurückgibt. Der Wert in pttype bestimmt, ob es sich um ptrval1 (pttype==PA_PTRVAL1) oder ptrval2 (pttype==PA_PTRVAL2) handelt. Wenn in ptrval1 beispielsweise ein zugeordnetes Feld abgelegt wird, dann kann die Funktion den Feldnamen zurückgeben. Wenn die Funktion definiert

wird, dann muss sie auch den Fall ptrval == NULL behandeln und dann eine leere Zeichenkette zurückgeben. Die zurückgegebene Zeichenkette wird auf 37 Zeichen gekürzt bzw. erweitert, damit die Tabelle formatiert ausgegeben wird.

Um die Daten der Paintareas aller Objekte auszugeben, für die ein gültiger Eintrag im Cache existiert, ist die Funktion

```
void pa_dump_all(FILE *fout)
```

aufzurufen. Das Format der Ausgabe entspricht dem von pa_dump_for_obj().

10 Verwendung von Paintareas in Selos

Durch Setzen der Ressource

```
UseLabelsInSelo
```

auf den Wert TRUE können Selos umgeschaltet werden, so dass Textlabels zur Beschriftung verwendet werden. Auch das Darstellen der Überschriften als Tabellenspalten benutzt bereits intern Paintareas vom Typ PA_TABLEHEADER, wenn diese über

```
BOOLEAN enableSeloAsTable(delim, id1, id2)
```

eingeschaltet werden. Dabei ist es unbedingt notwendig, dass die Funktion FwownDrawPaintArea() diesen Typ behandelt oder durch Rückgabe von "" das Darstellen eines Kopfes für eine Tabellenspalte *FIX/Win* überlässt.

11 Besonderheiten bei bestimmten Typen von Paintareas

11.1 PA_TABLEHEADER

Bei dem Typ PA_TABLEHEADER bestimmt der Wert in longval1 die Art des Kopfes der Tabellenspalte. Eine Tabellenspalte am Ende der Tabelle muss anders gezeichnet werden, als eine zu Beginn oder in der Mitte der Tabelle. Für das Argument sind deshalb folgende Werte zulässig:

- PA_TABLEHEADER_FIRST - Spaltenkopf am Anfang oder in der Mitte der Tabelle.
- PA_TABLEHEADER_LAST - Spaltenkopf am Ende der Tabelle.
- PA_TABLEHEADER_HEADER - Marker an der Seite eines Selos (Auswahlbuchstaben a-z).

11.2 PA_BUTTON

Bei dem Typ PA_BUTTON darf als Wert für pa_attr nicht der Wert BOLD oder INVERS angegeben werden. Diese Werte werden bereits von *FIX/Win* für die Darstellung des Buttons in verschiedenen Zuständen verwendet. Wird eines dieser Attribute übergeben, dann wird NORMAL verwendet.

Da ein Button anklickbar ist, muss der Wert in pa_bt_mask mindestens eine Maustaste definieren. Ist das nicht der Fall, dann wird PA_BT_LEFT verwendet.

Beispiel

```
hButtons[0] = pa_put_button(AUFTR_mskp, 5, 40, "Button 1", NULL, NULL,
    NORMAL, PA_ALGN_LEFT, PA_BT_LEFT, 1L, 1L, 1L, 1L, NULL, NULL);
hButtons[1] = pa_put_button(AUFTR_mskp, 5, 48, "Button 2", NULL, NULL,
    NORMAL, PA_ALGN_LEFT, PA_BT_LEFT | PA_BT_RIGHT, 1L, 1L, 0L, 0L, NULL, NULL);
```

```
hButtons[2] = pa_put_button(AUFTR_mskp, 5, 58, "Button 3", NULL, NULL,
    GRAYED, PA_ALGN_LEFT, PA_BT_LEFT | PA_BT_RIGHT | PA_BT_MIDDLE,
    3L, 3L, 0L, 0L, NULL, NULL);
```

Dieser Code erzeugt drei Buttons in der Maske AUFTR_mskp mit den Beschriftungen "Button 1", "Button 2" und "Button 3". Der erste ist nur mit der linken Maustaste anklickbar, der zweite mit der linken und rechten Maustaste und der dritte mit allen Maustasten. Die Handles der Buttons werden in dem Array hButtons abgelegt.

12 Varbuttons

Reiter für Varianten dienen zum Umschalten auf eine beliebige Variante durch einen Mausklick. Die Reiter werden durch Paintareas des neuen Typs PA_VARBUTTON realisiert. Als Bezeichnung für diese Paintareas wird der Begriff *Varbutton* in der Dokumentation verwendet. *FIX* bietet ein API an, das die Definition einer Zeile mit mehreren Paintareas vom Typ PA_VARBUTTON erlaubt. *FIX/Win* enthält Zeichenfunktionen, die Varbuttons entsprechend darstellen, so dass die von anderen Programmen bekannte Optik entsteht. *FIX* übernimmt dabei folgende Aufgaben, ohne dass eine explizite Programmierung notwendig ist:

- Darstellung der Zeile mit Varbuttons in der aktuell sichtbaren Variante.
- Umschaltung der Variante, wenn eine der Paintareas angeklickt wurde.
- Automatische Darstellung von Buttons zum Verschieben der Paintareas, wenn die Gesamtbreite größer wird, als der verfügbare Platz.
- Verschieben der Paintareas beim Anklicken der Buttons zum Verschieben.
- Automatische Steuerung des aktiven Zustands der Buttons zum Verschieben: Wenn kein weiteres Verschieben möglich ist, wird der entsprechende Button grau dargestellt und ist damit nicht anklickbar.

Voraussetzung dafür ist allerdings, dass die betroffene Variante zu diesem Zeitpunkt das aktive Objekt ist. Ist das nicht der Fall, dann besteht die Möglichkeit, durch Aufruf von set_activeobj_list(), das aktuelle Objekt zusätzlich als aktiv zu markieren. Allerdings muss dann die Behandlung der Events beim Anklicken der Reiterleiste (im Wesentlichen BT_LEFT_PA) selbst in der Anwendungslogik der aktiven Maske programmiert werden. Um das oben beschriebene Verhalten zu erreichen können dazu die Funktionen des APIs für Varbuttons aufgerufen werden. Einer der folgenden Abschnitte zeigt dies an einem Beispiel.

Die Makros und Strukturen, die für Varbuttons benötigt werden, werden in der Datei `fix/fixwin.h` definiert, die über `fix/fix.h` eingebunden wird. Die Funktionsprototypen werden in `proto/fixwin_pto.h` definiert. Diese Datei wird über `fix/libfix.h` eingebunden.

12.1 Definition einer Zeile mit Varbuttons

Die Definition der Varbuttons für eine Maske muss vor dem Darstellen und dem Abarbeiten der Maske stattfinden. Ein guter Zeitpunkt ist unmittelbar nach dem Laden der Maske. Bei der Definition werden die Varbuttons noch nicht direkt gezeichnet. Die Daten der Varbuttons werden in einer maskeninternen Struktur gehalten. Erst nach dem Anzeigen der Maske oder beim Umschalten der Variante werden anhand der Daten Varbuttons als Paintareas im Layout der Maske erzeugt.

Zur Definition der Varbuttons einer Maske kann zu Beginn die Funktion

```
BOOLEAN m_init_varbuttons(mask *mskp, int cnt, long placement, int z, int s_from, int s_to)
```

aufgerufen werden. Sie definiert durch Angabe des Wertes cnt die maximal mögliche Anzahl von Varbuttons und den Bereich in dem die Varbuttons für die Maske mskp und deren Varianten dargestellt werden sollen. Der Wert in z bestimmt die Zeile, der Wert in s_from bestimmt die erste Spalte und der Wert in s_to bestimmt die letzte Spalte. Alle Angaben sind relativ und beziehen sich auf die linke obere Ecke der Maske. In placement ist die Ausrichtung der Varbuttons anzugeben. Je nach Wert werden die Varbuttons anders von *FIX/Win* gezeichnet. Folgende Werte sind möglich:

- PA_VBTN_PLMT_TOP - Ausrichtung am oberen Rand.

- PA_VBTN_PLMT_BOTTOM - Ausrichtung am unteren Rand.
- PA_VBTN_PLMT_LEFT - Ausrichtung am linken Rand.
- PA_VBTN_PLMT_RIGHT - Ausrichtung am rechten Rand.

Die beiden letzten Werte werden von *FIX* und *FIX/Win* zur Zeit nicht unterstützt und sind für zukünftige Erweiterungen vorgesehen.

Als Ergebnis liefert die Funktion TRUE bei Erfolg oder FALSE im Fehlerfall. Ein Fehler kann eine der folgenden Ursachen haben:

- Die Maske mskp besitzt keine Varianten.
- Für Maske mskp wurde die Funktion m_init_varbuttons() bereits aufgerufen.
- Der durch s_from und s_to definierte Bereich ist breiter als die Maske.
- Der durch s_from und s_to definierte Bereich ist zu klein und kann nicht die Buttons zum Scrollen aufnehmen (< 6 Zeichen).
- Die durch z definierte Zeile liegt außerhalb der Maske.
- Die Spalte s_from oder die Spalte s_to liegt außerhalb der Maske.

Wenn die Anwendung die Funktion m_init_varbuttons() nicht vor der Definition des ersten Varbuttons aufruft oder wenn die Funktion fehlschlägt, dann ruft *FIX* die Funktion auf und definiert als maximale Anzahl von Varbuttons die Anzahl Varianten einer Maske. Als Bereich für die Varbuttons wird die erste Zeile der Maske verwendet. Die erste Position liegt bei 1 und die letzte liegt vor der letzten Position der Maske, so dass vorne und hinten noch ein Zeichen Platz zur Darstellung einer Ecke übrigbleibt. Für die Ausrichtung wird der Wert PA_VBTN_PLMT_TOP verwendet.

Zur Definition eines einzelnen Varbuttons ist die Funktion

```
BOOLEAN m_put_varbutton(mask *mskp, int posnr, int variant, char* token,
    char* text, char* tooltiptext, unsigned char attr, unsigned char align, unsigned char bt_mask)
```

aufzurufen. Sie überträgt die Daten für einen Varbutton in die maskeninterne Struktur. Die Reihenfolge der Varbuttons wird über eine Positionsnummer bestimmt, die in posnr zu übergeben ist. Hierfür ist ein Wert zwischen 0 und der maximal möglichen Anzahl (cnt) zu verwenden (0 <= posnr < cnt). Der Wert in variant definiert die Nummer der Variante, auf die umgeschaltet werden soll. Die Werte in token, text, attr, align und bt_mask werden in die interne Struktur eingetragen und später zur Erzeugung des Varbuttons verwendet. In token ist der Kurztext für die Paintarea anzugeben. Der darzustellende Text kann in text angegeben werden. Wird hier ein NULL-Zeiger übergeben, dann wird der Text durch die Übersetzungsfunktion bestimmt. Um dem Varbutton einen Tooltip zuzuordnen, ist in dem Parameter tooltiptext ein Text anzugeben. Soll kein Tooltip angezeigt werden, muss für diesen Parameter der Wert NULL oder eine leere Zeichenkette angegeben werden. Der Parameter attr enthält das Bildschirmattribut. Der Wert in align bestimmt die Ausrichtung des Textes innerhalb des Varbuttons. In bt_mask sind die Maustasten zu kodieren, die zum Anklicken verwendet werden können. Die Konstanten, die für align und bt_mask verwendet werden können, sowie eine genaue Beschreibung der übrigen Parameter ist dem Kapitel über Paintareas zu entnehmen. Als Besonderheit der Varbuttons kommt hinzu, dass in bt_mask der Wert für die linke Maustaste enthalten sein muss (PA_BT_LEFT). Ist das nicht der Fall, dann wird dieser Wert automatisch von *FIX* hinzugefügt. Varbuttons sind also immer mit der linken Maustaste anklickbar. Dies ist zugleich die einzige Maustaste, die von *FIX* behandelt wird. Werden zusätzliche Maustasten definiert, dann sind diese in der eigenen Anwendungslogik zu behandeln. Auf die angeklickte Paintarea kann dazu in der gewohnten Art und Weise (pa_get_clicked(), pa_get(), ...) zugegriffen werden.

Als Ergebnis liefert die Funktion TRUE bei Erfolg oder FALSE im Fehlerfall. Ein Fehler kann eine der folgenden Ursachen haben:

- Die Maske mskp besitzt keine Varianten.
- Die als posnr übergebene Position liegt außerhalb des Bereichs.
- Für die Position wurde bereits ein Varbutton definiert.
- Die Maske besitzt keine Variante mit der als variant übergebenen Variantennummer.
- Der Varbutton ist größer als der zur Verfügung stehende Platz.
- Die Anwendung hat die Funktion m_init_varbuttons() nicht aufgerufen und der Aufruf von m_init_varbuttons() durch *FIX* schlug fehl.

Beispiel:

```
m_init_varbuttons(POS_mskp, 3, PA_VBTN_PLMT_TOP, 0, 1, 60);
m_put_varbutton(POS_mskp, 0, 0, "Positionen", NULL, NULL,
    NORMAL, PA_ALGN_CENTER, PA_BT_LEFT);
m_put_varbutton(POS_mskp, 1, 1, "Positionen Einzel", NULL, NULL,
    NORMAL, PA_ALGN_CENTER, PA_BT_NONE);
m_put_varbutton(POS_mskp, 2, 2, "Positionen Tabelle 2", NULL, NULL,
    NORMAL, PA_ALGN_CENTER, PA_BT_LEFT|PA_BT_RIGHT);
```

Der Code definiert eine Leiste mit drei Varbuttons, die am oberen Rand ausgerichtet werden. Die Leiste wird in Zeile 0 positioniert und geht von Spalte 1 bis Spalte 60.

12.2 Umschalten von Varianten

Wenn *FIX* das Anklicken eines Varbuttons mit der linken Maustaste erkennt, wird geprüft, ob dieser Varbutton im aktuellen Objekt liegt und ob für dieses Objekt eine Leiste mit Varbuttons definiert wurde. Wenn ja, dann wird die Nummer der zugeordneten Variante ermittelt und der Varbutton wird als aktiv gekennzeichnet. Danach wird das Event L_MODE an die Anwendung gesendet. Die Anwendung hat nun die Möglichkeit, den Wechsel der Variante zu unterbinden, indem sie als Ergebnis L_STAY liefert. Um dies entscheiden zu können, kann sie die Nummer der Variante, auf die geschaltet werden soll, durch Aufruf von

```
int fx_get_target_variant()
```

ermitteln.

Beim Drücken der Taste L_MODE, also beim Umschalten der Variante über die Tastatur, wird von *FIX* die nächste Variante als Zielvariante ermittelt. Danach wird das Event an die Anwendungslogik weitergegeben. Diese erhält beim Aufruf von fx_get_target_variant() die nächste Variante und kann genauso darauf reagieren, wie beim Anklicken eines Buttons. Als aktiven Varbutton der Leiste bestimmt *FIX* im Falle der Taste L_MODE den ersten Varbutton, der auf die Zielvariante schalten würde. Wenn es pro Variante nur einen Varbutton in der Leiste gibt, dann ist das immer der richtige Varbutton, der nach dem Variantenwechsel als aktiv dargestellt werden soll. Es kann jedoch auch vorkommen, dass einer Variante zwei Varbuttons zugeordnet wurden. Im Falle des Anklickens kann *FIX* den aktiven Varbutton bestimmen. Bei einem Tastenevent ist das jedoch nicht möglich. Deshalb hat die Anwendung die Möglichkeit mit einer der Funktionen

```
BOOLEAN m_set_active_varbutton(mask *mskp, HPAINTAREA h_pa)
BOOLEAN m_set_active_varbutton_nr(mask *mskp, int posnr)
```

einen Varbutton der Maske als aktiven Varbutton zu bestimmen.

12.3 Aktualisieren und Löschen von Varbutton

Zum Aktualisieren der Eigenschaften eines Varbuttons ist die Funktion

```
BOOLEAN m_update_varbutton(mask *mskp, int posnr, paintarea* pa, unsigned long validmask)
```

aufzurufen. Sie bekommt als Parameter einen Zeiger auf die Maske (mskp), die Positionsnummer des Varbuttons (posnr), Paintarea-Daten (pa) und einen long-Wert, der beschreibt, welche der Paintarea-Daten verändert werden sollen (validmask). Damit ist es möglich, nur bestimmte Daten des Varbuttons zu aktualisieren. Eine besondere Bedeutung kommt den Werten in pa->pa_props.longval1 und pa->pa_props.longval2 zu. Wenn der Wert in longval1 gesetzt wird, dann wird er durch Oder-Verknüpfung bei dem aktiven Varbutton um PA_VBTN_ACTIVE und bei allen Varbuttons um das der Maske zugeordnete Placement (PA_VBTN_PLMT_..) ergänzt. In longval2 ist die Variante des Varbuttons abzulegen. Falls der Varbutton zum Zeitpunkt des Aufrufs sichtbar ist, wird die Funktion pa_update() mit den Werten in pa und validmask aufgerufen. Der Aufbau des Wertes pa und die möglichen Werte für validmask sind der Beschreibung der Funktion pa_update() zu entnehmen. Wenn der Aufruf von pa_update() keinen Fehler meldet, oder wenn der Varbutton zum Zeitpunkt des Aufrufs von m_update_varbutton() nicht sichtbar ist, werden die Paintarea-Daten in der maskeninternen Datenstruktur aktualisiert und stehen somit für das Erzeugen des Varbuttons in anderen Varianten zur Verfügung. Auch hier wird der Wert in validmask ausgewertet, um nur bestimmte Daten zu berücksichtigen.

Als Ergebnis liefert die Funktion TRUE bei Erfolg oder FALSE im Fehlerfall. Ein Fehler kann eine der folgenden Ursachen haben:

- Für die Maske mskp wurde keine Leiste mit Varbuttons definiert.
- Die als posnr übergebene Position liegt außerhalb des Bereichs.
- Für die Position wurde kein Varbutton definiert oder der definierte Varbutton wurde gelöscht.
- Die als pa->pa_props.longval2 übergebene Variantennummer liegt außerhalb des Bereichs. Der Wert wird nur geprüft, wenn er auch verwendet wird, also wenn in validmask der Wert PA_LONGVAL2 enthalten ist.
- Der Aufruf von pa_update() durch *FIX* schlug fehl.

Zum Löschen eines Varbuttons ist die Funktion

```
BOOLEAN m_delete_varbutton(mask *mskp, int posnr)
```

aufzurufen. Sie entfernt den Varbutton aus der internen Datenstruktur. Wenn der Varbutton zum Zeitpunkt des Aufrufs sichtbar ist, dann wird versucht alle anderen Varbuttons neu darzustellen. Dabei kann es dazu kommen, dass Varbuttons, die vorher außerhalb der Leiste lagen, jetzt sichtbar sind.

Als Ergebnis liefert die Funktion TRUE bei Erfolg oder FALSE im Fehlerfall. Ein Fehler kann eine der folgenden Ursachen haben:

- Für die Maske mskp wurde keine Leiste mit Varbuttons definiert.
- Die als posnr übergebene Position liegt außerhalb des Bereichs.
- Für die Position wurde kein Varbutton definiert oder der definierte Varbutton wurde gelöscht.

12.4 Weitere nützliche Funktionen

Mit Hilfe der Funktion

```
int m_get_varbutton_cnt(mask *mskp;
```

kann die maximal mögliche Anzahl Varbuttons für eine Maske ermittelt werden. Als Wert wird der bei m_init_varbuttons() als cnt definierte Wert zurückgegeben. Wurde m_init_varbuttons() nicht aufgerufen, dann wird 0 zurückgegeben.

Die Positionsnummern des ersten und des letzten sichtbaren Varbuttons können mit den Funktionen

```
int m_get_first_varbutton_nr(mask *mskp)
int m_get_last_varbutton_nr(mask *mskp)
```

ermittelt werden. Dabei ist zu beachten, dass zwischen diesen beiden Positionen durchaus Positionen liegen können, für die kein Varbutton definiert wurde.

Die Funktion

```
int m_get_active_varbutton_nr(mask *mskp)
```

liefert die Positionsnummer des aktiven Varbuttons der Maske. Wenn für die Maske keine Leiste mit Varbuttons definiert wurde, liefert die Funktion -1 zurück.

Mit

```
BOOLEAN m_set_active_varbutton_nr(mask *mskp, int posnr)
```

kann die Positionsnummer des aktiven Varbuttons festgelegt werden. Die Funktion liefert FALSE zurück, wenn für die Maske keine Leiste mit Varbuttons definiert wurde, der Wert für posnr außerhalb des Bereichs liegt oder für diese Position kein Varbutton definiert wurde. Ansonsten legt die Funktion den Varbutton an dieser Position als aktiven Varbutton fest und kehrt mit TRUE zurück. Die dem Varbutton zugeordnete Variante sollte die Variante sein, auf die als nächstes umgeschaltet wird. Eine Prüfung der dem Varbutton zugeordneten Variante kann innerhalb von m_set_active_varbutton_nr() nicht erfolgen, da die nächste Variante auch von der Anwendung bestimmt werden kann. Statt dessen prüft *FIX* bei der Darstellung der Leiste (innerhalb von m_show_varbuttons(); diese Funktion wird auch bei einem Wechsel der Variante zur Darstellung der Varbuttons in der neuen Variante aufgerufen), ob der mit m_set_active_varbutton_nr() definierte Varbutton auf die aktuelle Variante zeigt. Ist das nicht der Fall, dann bestimmt *FIX* den aktuellen Varbutton neu, indem der erste Varbutton gesucht wird, der auf die aktuelle Variante zeigt.

Als Alternative zu m_set_active_varbutton_nr() kann die Funktion

```
BOOLEAN m_set_active_varbutton(mask *mskp, HPAINTAREA h_pa)
```

verwendet werden. Sie bekommt statt einer Positionsnummer das Handle des Varbuttons. Deshalb kann sie nur für dargestellte Varbuttons aufgerufen werden. m_set_active_varbutton_nr() kann hingegen auch für Varbuttons aufgerufen werden, die außerhalb des sichtbaren Bereichs liegen und erst durch ein Scrollen der Leiste sichtbar werden. Beiden Funktionen ist gemeinsam, dass sie die Leiste und damit den aktiven Varbutton nicht neu darstellen. Dazu ist die Variante zu wechseln oder die Funktion

```
BOOLEAN m_show_varbuttons(mask *mskp)
```

aufzurufen. Sie entfernt alle Varbuttons, restauriert den alten Hintergrund an der Stelle, an der die Leiste mit den Varbuttons zum liegen kommt und stellt alle Varbuttons der Leiste neu (mit den aktuellen Eigenschaften) dar.

Um das Handle eines dargestellten Varbuttons zu ermitteln, kann die Funktion

```
HPAINTAREA m_get_varbutton(mask *mskp, int posnr)
```

aufgerufen werden. Sie liefert das Handle des Varbuttons der Maske mskp an der Position posnr. Wenn für die Maske keine Leiste mit Varbuttons definiert wurde oder wenn die Positionsnummer außerhalb des Bereichs 0 - cnt liegt, dann liefert die Funktion HPAINTAREA_INVALID zurück. Wenn für die Position kein Varbutton definiert wurde, oder wenn der Varbutton zur Zeit nicht sichtbar ist, dann liefert die Funktion HPAINTAREA_EMPTY zurück.

12.5 Spezielle Situationen in der Anwendung

Die folgenden Abschnitte zeigen spezielle Situationen in Anwendungen und deren Behandlung im Zusammenhang mit Varbuttons.

12.5.1 Umschalten von Varianten in fremden Masken

Wenn *FIX* das Anklicken eines Varbuttons erkennt, der nicht zur aktuellen Maske gehört, dann muss das Umschalten der Variante von der Anwendung übernommen werden. Dazu sendet *FIX*, wie beim Anklicken aller Typen von Paintareas, das Event BT_LEFT_PA an die Anwendungslogik der aktuellen Maske. Voraussetzung dafür ist, dass die Maske, die den Varbutton enthält in der Liste der aktiven Masken steht, die über set_activeobj_list() für das aktuelle Objekt zu definieren ist.

Die Anwendungslogik hat die Möglichkeit mit Hilfe der Funktionen pa_get_clicked() und pa_get() Informationen über die angeklickte Paintarea zu ermitteln. Durch Vergleich des Typs (pa_type == PA_VARBUTTON) und des Objektzeigers (pa_objp == <Objekt mit Varianten>) kann sie feststellen, ob ein Varbutton in einer bestimmten Maske angeklickt wurde. Wenn ja, dann kann die Umschaltung der Variante durch die *FIX*-Funktion m_setvariant() vorgenommen werden. Die Nummer der Zielvariante ist dabei aus dem Wert pa_props.longval2 des angeklickten Varbuttons zu entnehmen. Da der Variantenwechsel die alte Variante vom Bildschirm entfernt und die neue darstellt, wird auch die Leiste mit Varbuttons neu dargestellt. Als aktiver Varbutton wird dabei der erste Varbutton bestimmt, der die aktuelle Variante als Ziel definiert. Wenn es für eine Variante mehr als einen Varbutton gibt, der diese Variante als Ziel hat, dann kann der von *FIX* bestimmte aktive Varbutton falsch sein. Deshalb sollte die Anwendung vor der Umschaltung der Variante den aktiven Varbutton über eine der Funktionen

```
BOOLEAN m_set_active_varbutton(mask *mskp, HPAINTAREA h_pa)
BOOLEAN m_set_active_varbutton_nr(mask *mskp, int posnr)
```

selbst bestimmen. Das Ermitteln des aktiven Varbuttons durch *FIX* unterbleibt in diesem Fall, es sei denn, der durch m_set_active_varbutton()/m_set_active_varbutton_nr() festgelegte Varbutton ist nicht der aktuellen Variante zugeordnet. Wenn es für jede Variante nur einen Varbutton gibt, dann ist der Aufruf von m_set_active_varbutton()/ m_set_active_varbutton_nr() nicht notwendig.

Neben den Varbuttons der Leiste kann es sich bei dem angeklickten Varbutton auch um einen Varbutton zum Scrollen der Leiste handeln. Diese Varbuttons sind daran zu erkennen, dass in den Bits 8-9 des longval1 einer der Werte

- PA_VBTN_SCRL_FWRD - für Vorwärts scrollen
- PA_VBTN_SCRL_BWRD - für Rückwärts scrollen

steht. Das Scrollen der Leiste ist durch Aufruf der Funktion

```
BOOLEAN m_scroll_varbuttons(mask *mskp, BOOLEAN forward)
```

auszulösen. mskp enthält den Zeiger auf die Maske, deren Leiste verschoben werden soll. forward bestimmt die Richtung, in der die Varbuttons verschoben werden. TRUE bedeutet, dass die Varbuttons nach vorne (links bei horizontaler Anordnung) verschoben werden. FALSE bedeutet, dass die Varbuttons nach hinten (rechts bei horizontaler Anordnung) verschoben werden. Wenn die Maske mskp keine Leiste besitzt, liefert die Funktion FALSE zurück. Ansonsten wird der erste sichtbare Varbutton neu bestimmt und die Leiste mit Varbuttons wird neu dargestellt.

Beispiel

Dieses Beispiel zeigt wie in der Anwendungslogik der Kopfmaske die Umschaltung der Varianten der Positionsmaske (POS_mskp) vorgenommen werden.

```
case BT_LEFT_PA:
    /* Paintarea wurde mit linker Maustaste angeklickt */
    hClickedButton = pa_get_clicked();
    pa_get(hClickedButton, &pa_click1);
    if (pa_click1.pa_objp == (obj *)POS_mskp) {
        /* Paintarea befindet sich in POS_mskp */
        if ( pa_click1.pa_type == PA_VARBUTTON ) {
            /* Paintarea ist ein Varbutton in POS_mskp */
            if ((pa_click1.pa_props.longval1 & ~PA_VBTN_PLMT_BITS) ==
                PA_VBTN_SCRL_FWRD) {
                /* Paintarea ist der Button zum vorwaerts scrollen */
                m_scroll_varbuttons(POS_mskp, TRUE);
            } else if ((pa_click1.pa_props.longval1 & ~PA_VBTN_PLMT_BITS) ==
                PA_VBTN_SCRL_BWRD) {
                /* Paintarea ist der Button zum rueckwaerts scrollen */
                m_scroll_varbuttons(POS_mskp, FALSE);
            } else {
                /* Paintarea ist ein Button, Umschalten der Variante */
                m_set_active_varbutton(POS_mskp, hClickedButton);
                m_setvariant(POS_mskp, pa_click1.pa_props.longval2);
            }
        }
    }
    rc = L_STAY;
    break;
```

12.5.2 Darstellung des aktiven Varbuttons

Um den aktiven Varbutton stärker hervorgehoben darzustellen, kann in der Benutzerbibliothek eine eigene Routine zur Darstellung des Varbuttons erstellt werden. Eine Alternative besteht darin, für den aktiven Varbutton ein anderes Attribut zu vergeben und in der Farbzuordnungstabelle eine entsprechende Farbe für dieses Attribut zu definieren. Bei jedem Variantenwechsel müssen die Attribute der Varbuttons neu definiert werden. Dazu kann folgende Funktion implementiert werden:

```
void update_varbuttons(mask* mskp)
{
    int i, cnt, act;
    paintarea pa;
    cnt = m_get_varbutton_cnt(mskp);
    act = m_get_active_varbutton_nr(mskp);
    for (i = 0; i <= cnt; i++) {
        if (i == act) {
            pa.pa_attr = NORMAL;
        } else {
```

```
            pa.pa_attr = LOW;
        }
        m_update_varbutton(mskp, i, &pa, PA_ATTR);
    }
}
```

Sie ermittelt im ersten Schritt die Anzahl der Buttons und die Positionsnummer des aktiven Varbuttons. Dann läuft sie in einer Schleife über alle Varbuttons und bestimmt das Attribut jedes Varbuttons neu. Wenn es sich um den aktiven Varbutton handelt, dann verwendet sie NORMAL und sonst LOW.

Die Funktion ist immer dann aufzurufen, wenn sich der aktive Varbutton ändert. Zusätzlich ist nach der Definition aller Varbuttons ein aktiver Varbutton zu definieren. Wird die Umschaltung in fremden Masken implementiert, dann ergeben sich drei Stellen für den Aufruf der Funktion. Der folgende Beispielcode zeigt die Stellen aus der *FIX*-Beispielanwendung (Windemo):

Nach der Definition aller Varbuttons:

```
m_init_varbuttons(POS_mskp, 3,
    PA_VBTN_PLMT_TOP, 0, 1, 60);
m_put_varbutton(POS_mskp, 0, 0, "Positionen", NULL, NULL,
    NORMAL, PA_ALGN_CENTER, PA_BT_LEFT);
m_put_varbutton(POS_mskp, 1, 1, "Positionen Einzel", NULL, NULL,
    NORMAL, PA_ALGN_CENTER, PA_BT_NONE);
m_put_varbutton(POS_mskp, 2, 2, "Positionen Tabelle 2", NULL, NULL,
    NORMAL, PA_ALGN_CENTER, PA_BT_LEFT|PA_BT_RIGHT);
m_set_active_varbutton_nr(POS_mskp, 0);
update_varbuttons(POS_mskp);
```

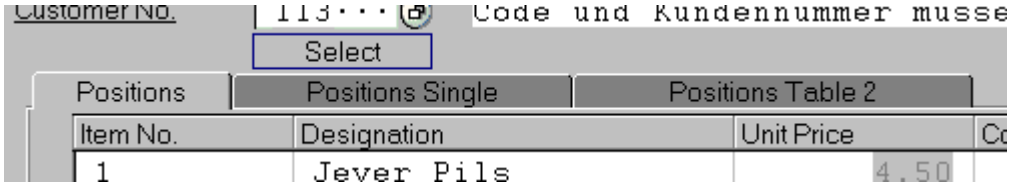
Vor dem Umschalten der Variante in der Kopfmaske:

```
if ( paTemp1.pa_type == PA_VARBUTTON ) {
    if ((paTemp1.pa_props.longval1 & ~PA_VBTN_PLMT_BITS) ==
        PA_VBTN_SCRL_FWRD) {
        m_scroll_varbuttons(POS_mskp, TRUE);
    } else if ((paTemp1.pa_props.longval1 & ~PA_VBTN_PLMT_BITS) ==
        PA_VBTN_SCRL_BWRD) {
        m_scroll_varbuttons(POS_mskp, FALSE);
    } else {
        m_set_active_varbutton(POS_mskp, hClickedButton);
        update_varbuttons(POS_mskp);
        m_setvariant(POS_mskp, paTemp1.pa_props.longval2);
    }
}
```

Vor dem Umschalten in der Positionsmaske. Hierzu ist das Event L_MODE abzufangen:

```
Event POS_event_control(obj *objp, Event event)
{
    switch(event) {
    ...
    case L_MODE:
        update_varbuttons(POS_mskp);
        break;
    ...
    }
}
```

Die folgende Abbildung zeigt das Ergebnis, wobei für das Attribut LOW die Hintergrundfarbe dunkelgrau (GRAY2) eingestellt wurde:


Customer No.
113···
Code und Kundennummer musse
Select
Positions
Positions Single
Positions Table 2
Item No.
Designation
Unit Price
1
Jever Pils
4.50

35 Nutzung von Kontextmenüs

1 Anzeige von Kontextmenüs

Durch den Aufruf der Funktion

```
BOOLEAN show_context_menu(context_menu* ctx_men);
```

kann ein Kontextmenü angezeigt werden. Der Prototyp der Funktion wird in `proto/fixwin_pto` definiert, die über `fix/libfix.h` eingebunden wird.

Das Argument vom Typ context_menu* bestimmt den Aufbau des Kontextmenüs. Der Typ wird in der Datei `fix/fixwin.h` definiert, die über die Datei `fix/fix.h` eingebunden wird. Die Struktur context_menu hat folgenden Aufbau:

```
typedef struct {
    long position_meth;
    long x;
    long y;
    unsigned long flags;
    long cnt;
    context_menu_entry* entry;
} context_menu;
```

Die Komponente position_meth definiert die Methode, mit der das Menü positioniert wird. Die Werte in x und y enthalten eine Position dazu.

Die Komponente flags enthält Angaben, über die das Verhalten des Menüs gesteuert werden kann.

Der Wert cnt bestimmt die Anzahl der Menüpunkte. Die einzelnen Menüpunkte werden in einem Array definiert, auf das die Komponente entry zeigt.

Tritt bei der Ausführung ein Fehler auf, dann liefert die Funktion FALSE zurück. Dies kann z.B. dann passieren, wenn die Komponente entry auf NULL zeigt, oder NULL als Argument übergeben wurde.

Wenn das Menü mit allen Untermenüs korrekt an *FIX/Win* übermittelt wurde, wird TRUE zurückgegeben.

1.1 Positionierungsmethoden

Zur Positionierung des Menüs bietet *FIX* verschiedene Methoden an. Zur Verwendung einer Methode ist eine der CTXMEN_POSMETH_.. Konstanten in die Komponente position_meth zu schreiben. Durch die Positionierungsmethode wird eine Position bestimmt, die als Bezugspunkt für das Menü dient. Zu diesem Punkt wird die Position in den Komponenten x und y addiert, um die Position zu erhalten, an der das Menü aufgeblendet wird. Je nach Methode werden die beiden Werte als Pixel oder Zeichenzellen interpretiert. Wird keine Positionierungsmethode angegeben (position_meth enthält einen nicht definierten Wert), dann definieren x und y die Position des Menüs in Pixel relativ zur linken oberen Ecke des *FIX/Win*-Fenster.

Da es sich bei dem Menü um ein eigenständiges Fenster handelt, kann es auch außerhalb des *FIX/Win*-Fensters positioniert werden.

CTXMEN_POSMETH_MOUSE

Als Bezugspunkt wird die aktuelle Position des Mauszeigers verwendet. Die Werte in x und y werden als Pixel interpretiert.

CTXMEN_POSMETH_CURSOR

Als Bezugspunkt wird die aktuelle Position des Cursors (Carets) verwendet. Die Werte in x und y werden als Pixel interpretiert. Wenn das Menü oberhalb des Bezugspunktes aufgeblendet wird (also wenn CTXMEN_TPM_BOTTOMALIGN gesetzt wird), dann wird die Position um die Größe einer Zelle nach oben verschoben. Ansonsten wird sie um die Größe einer Zelle nach unten verschoben. Diese Anpassung bewirkt, dass das Menü bei der Angabe von 0,0 für x,y genau über oder unter dem Cursor aufgeht und das aktuelle Feld nicht verdeckt.

CTXMEN_POSMETH_CELL

Die Verwendung dieser Methode bewirkt, dass die Werte in x und y als Zellenangaben interpretiert werden. Anhand der aktuellen Zellengröße wird von *FIX/Win* eine Pixelposition errechnet, die relativ zur linken oberen Ecke des *FIX/Win*-Fensters interpretiert wird.

1.2 Flags

Der Wert in der Komponente flags ist aus den CTXMEN_TPM_... Konstanten zu bilden. Wenn mehrere der Konstanten angegeben werden sollen, dann sind sie über eine oder-Verknüpfung zu verknüpfen. Es ist jedoch nicht möglich, alle Werte beliebig miteinander zu kombinieren. Die Werte werden deshalb in Gruppen zusammengefasst und es kann immer nur ein Element einer Gruppe verwendet werden. Die Werte und die Namen der Konstanten sind von Windows-Konstanten abgeleitet, die den gleichen Namen - jedoch ohne den Prefix CTXMEN - besitzen. Letztendlich wird der Wert von flags an die Windows-Funktion TrackPopupMenu() weitergegeben. Die Beschreibung der Konstanten kann deshalb auch in der Windows-Dokumentation zu dieser Funktion nachgelesen werden.

CTXMEN_TPM_LEFTBUTTON, CTXMEN_TPM_RIGHTBUTTON

Wenn CTXMEN_TPM_LEFTBUTTON oder keines der beiden Flags gesetzt wird, dann kann ein Menüpunkt nur mit der linken Maustaste ausgewählt werden. Wird CTXMEN_TPM_RIGHTBUTTON gesetzt, dann können beide Maustasten verwendet werden.

CTXMEN_TPM_LEFTALIGN, CTXMEN_TPM_CENTERALIGN, CTXMEN_TPM_RIGHTALIGN

Wenn CTXMEN_TPM_LEFTALIGN oder keines der drei Flags gesetzt wird, dann bezieht sich die Positionsangabe auf die linke Seite des Menüs. Wird CTXMEN_TPM_RIGHTALIGN angegeben, dann bezieht sie sich auf die rechte Seite. Die Angabe von CTXMEN_TPM_CENTERALIGN bewirkt, dass sich die Positionsangabe auf die (horizontale) Mitte des Menüs bezieht.

1.3 Menüpunkte

Die Anzahl der Menüpunkte ist in der Komponente cnt zu hinterlegen. Die Komponente entry muss auf ein entsprechend großes Array des Typs context_menu_entry zeigen. Jedes Element des Arrays definiert einen Menüpunkt. Die Struktur hat folgenden Aufbau:

```
typedef struct {
    char* text;
    char* bmp_fname;
    unsigned long flags;
    long fct;
    long param1;
```

```
    long param2;
} context_menu_entry;
```

1.4 Text des Menüpunktes

Die Komponente text definiert den Text des Menüpunktes. Um einen Auswahlbuchstaben zu definieren, ist dem entsprechenden Buchstaben ein & voranzustellen. Wenn eine Tastenbeschriftung angezeigt werden soll, dann ist diese hinter einem Zeichen am Ende des Textes anzugeben. Optional besteht die Möglichkeit, die Tastenbeschriftung automatisch anhand der Tastaturtabelle zu ermitteln (siehe CTXMEN_LOOKUPKEYNAME).

Die Länge eines Textes darf maximal 256 Zeichen inclusive Null-Byte betragen. Längere Texte werden abgeschnitten.

1.5 Bitmap des Menüpunktes

In der Komponente bmp_fname kann der Pfad zu einer Datei angegeben werden, die auf *FIX/Win*-Seite im BMP-Format vorliegen muss. Die Bitmap in dieser Datei wird neben dem Menüpunkt dargestellt. Wenn notwendig, wird dazu die Größe der Bitmap angepasst. Wenn ein relativer Pfad angegeben wird, dann bezieht er sich auf das Installationsverzeichnis von *FIX/Win*.

Die Bitmap wird nicht dargestellt, wenn das Flag CTXMEN_MF_CHECKED angegeben wird. Wenn eine Bitmap angegeben ist und dieses Flag angegeben wird, dann wird weder eine Bitmap noch eine Markierung angezeigt.

Wenn keine Bitmap angezeigt werden soll, dann ist in der Komponente bmp_fname der Wert NULL oder eine leere Zeichenkette zu hinterlegen.

Die Länge der Pfadangabe darf maximal 256 Zeichen inclusive Null-Byte betragen. Längere Pfadangaben werden abgeschnitten.

1.6 Flags des Menüpunktes

Der Wert für die Komponente flags entsteht durch Kombination der CTXMEN_MF_... Konstanten (Verknüpfung mit oder). Die Namen und Werte der Konstanten stammen von Windows, wobei die Namen dort nicht den Präfix CTXMEN besitzen. Der Wert von flags wird von *FIX/Win* der Windows-Funktion AppendMenu() übergeben. Die Beschreibung der Konstanten kann deshalb auch in der Windows-Dokumentation zu dieser Funktion nachgelesen werden. Da nicht alle Werte miteinander kombiniert werden können, werden sie in Gruppen beschrieben. Es kann immer nur eine Konstante einer Gruppe verwendet werden.

CTXMEN_MF_STRING, CTXMEN_MF_SEPARATOR

Wenn CTXMEN_MF_STRING oder keines der beiden Flags gesetzt wird, dann besteht der Menüpunkt aus dem Text der in der Komponenten text steht. Wird CTXMEN_MF_SEPARATOR angegeben, dann besteht der Menüpunkt aus einem Trennstrich und der Text sowie auch fct und param1, param2 werden ignoriert.

CTXMEN_MF_ENABLED, CTXMEN_MF_GRAYED, CTXMEN_MF_DISABLED

Wenn CTXMEN_MF_ENABLED oder keines der drei Flags gesetzt wird, dann kann der Menüpunkt ausgewählt werden. Wird CTXMEN_MF_DISABLED angegeben, dann kann der Menüpunkt nicht ausgewählt werden. Bei der Angabe von CTXMEN_MF_GRAYED kann der Menüpunkt ebenfalls nicht ausgewählt werden. Dieser Zustand wird zusätzlich visualisiert.

CTXMEN_MF_UNCHECKED, CTXMEN_MF_CHECKED

Wenn CTXMEN_MF_UNCHECKED oder keines der beiden Flags gesetzt ist, dann besitzt der Menüpunkt keine Markierung. Wird CTXMEN_MF_CHECKED angegeben, dann wird neben dem Menüpunkt eine Markierung angezeigt. Die Markierung wird nicht angezeigt, wenn für den Menüpunkt eine Bitmap definiert wurde.

CTXMEN_MF_MENUBARBREAK, CTXMEN_MF_MENUBREAK

Wenn das Flag CTXMEN_MF_MENUBARBREAK angegeben wird, dann wird ab diesen Menüpunkt eine neue Spalte begonnen. Die Spalte wird durch einen vertikalen Strich abgetrennt. Die Angabe von CTXMEN_MF_MENUBREAK hat mit Ausnahme des vertikalen Strichs die gleichen Auswirkungen.

CTXMEN_LOOKUPKEYNAME

Diese Konstante wird nur von *FIX/Win* unterstützt und ist kein Windows-Standard. Sie kann nur angegeben werden, wenn als fct der Wert CTXMEN_FCT_SENDEVENT angegeben wird. In diesem Fall definiert der Wert in param1 ein *FIX*-Event. *FIX/Win* sucht die Windows-Taste, die diesem Event in der Tastaturtabelle zugeordnet ist und erzeugt eine Tastenbeschreibung daraus. Die Tastenbeschreibung wird, durch Tab getrennt, als zusätzlicher Text des Menüpunktes verwendet.

1.7 Funktion des Menüpunktes

Der Wert in der Komponenten fct definiert die Funktion, die beim Auslösen des Menüpunktes ausgeführt wird. Eine der Konstanten mit den Namen CTXMEN_FCT_... kann als Wert für diese Komponente verwendet werden. Jede Funktion kann optionale Parameter enthalten, die in den Komponenten param1 und param2 abzulegen sind.

CTXMEN_FCT_NONE

Dieser Wert definiert die leere Funktion. Beim Auslösen des Menüpunktes wird keine Funktion aufgerufen. Der Wert kann beispielsweise zusammen mit CTXMEN_MF_SEPARATOR angegeben werden.

Die Werte param1 und param2 haben keine Bedeutung.

CTXMEN_FCT_SENDEVENT

Die Angabe dieser Funktion bewirkt das Senden eines Events bei der Auswahl des Menüpunktes. Der Wert param1 muss den Code des Events enthalten, der gesendet werden soll.

Der Wert param2 hat keine Bedeutung.

CTXMEN_FCT_COPY

Wenn ein Menüpunkt mit dieser Funktion ausgelöst wird, dann wird der (sichtbare) Inhalt des aktuellen Feldes in die Zwischenablage kopiert.

Die Werte param1 und param2 haben keine Bedeutung.

CTXMEN_FCT_PASTE

Wenn ein Menüpunkt mit dieser Funktion ausgelöst wird, dann wird der Text in der Zwischenablage als Eventfolge an die Anwendung gesendet.

Die Werte param1 und param2 haben keine Bedeutung.

CTXMEN_FCT_SUBMENU

Hierbei handelt es sich um eine Pseudofunktion. Bei der Auswahl des Menüpunktes wird keine Funktion ausgeführt. Statt dessen wird ein Untermenü aufgeblendet. Das Untermenü ist in der gleichen Weise wie das Hauptmenü in einer Struktur vom Typ context_menu zu definieren. In der Komponente param1 des Menüpunktes, der das Untermenü öffnen soll, ist ein Zeiger auf die Struktur des Untermenüs zu hinterlegen. Da die Definition eines Kontextmenüs zusammen mit allen Menüpunkten und Untermenüs auf einmal zu *FIX/Win* übertragen wird, muss die Definition des/der Untermenüs und ihrer Menüpunkte vor dem Aufruf von show_context_menu() erfolgen. Es besteht keine Möglichkeit, die Untermenüpunkte erst beim Auslösen des entsprechenden Menüpunktes zu definieren oder zu manipulieren, da in diesem Fall kein Event von Windows an *FIX/Win* und somit auch kein Event an die *FIX*-Anwendung gesendet wird.

Die Position des Untermenüs wird durch den zugeordneten Menüpunkt bestimmt. Die angegebene Positionierungsmethode ist ohne Bedeutung.

Der Wert param2 hat bei der Angabe des Menüpunktes, der das Untermenü öffnet, keine Bedeutung.

Beispiel:

```
context_menu ctx_men;
 context_menu_entry ctx_men_entry[4];
 context_menu ctx_men_sub;
 context_menu_entry ctx_men_sub_entry[2];

 ctx_men.position_meth = CTXMEN_POSMETH_MOUSE;
 ctx_men.flags = CTXMEN_TPM_RIGHTBUTTON;
 ctx_men.x = 0;
 ctx_men.y = 0;
 ctx_men.cnt = 4;
 ctx_men.entry = ctx_men_entry;

 ctx_men_entry[0].text = "Zeitstempel";
 ctx_men_entry[0].bmp_fname = "nsigrp/ctxbmp/TIME.bmp";
 ctx_men_entry[0].flags = CTXMEN_LOOKUPKEYNAME;
 ctx_men_entry[0].fct = CTXMEN_FCT_SENDEVENT;
 ctx_men_entry[0].param1 = (long)K_h1;

 ctx_men_entry[1].text = "Telefonliste";
 ctx_men_entry[1].bmp_fname = "nsigrp/ctxbmp/TELEFON.bmp";
 ctx_men_entry[1].flags = CTXMEN_LOOKUPKEYNAME;
 ctx_men_entry[1].fct = CTXMEN_FCT_SENDEVENT;
 ctx_men_entry[1].param1 = (long)K_h2;

 ctx_men_entry[2].text = "";
 ctx_men_entry[2].bmp_fname = "";
 ctx_men_entry[2].flags = CTXMEN_MF_SEPARATOR;
 ctx_men_entry[2].fct = CTXMEN_FCT_NONE;

 ctx_men_entry[3].text = "Weitere";
 ctx_men_entry[3].bmp_fname = "";
 ctx_men_entry[3].flags = 0L;
 ctx_men_entry[3].fct = CTXMEN_FCT_SUBMENU;
 ctx_men_entry[3].param1 = (long)&ctx_men_sub;

 ctx_men_sub.position_meth = CTXMEN_POSMETH_MOUSE;
 ctx_men_sub.flags = CTXMEN_TPM_RIGHTBUTTON;
```

```
ctx_men_sub.x = 0;
ctx_men_sub.y = 0;
ctx_men_sub.cnt = 2;
ctx_men_sub.entry = ctx_men_sub_entry;

ctx_men_sub_entry[0].text = "Kundenstamm";
ctx_men_sub_entry[0].bmp_fname = "nsigrp/ctxbmp/KUNDE.bmp";
ctx_men_sub_entry[0].flags = CTXMEN_LOOKUPKEYNAME;
ctx_men_sub_entry[0].fct = CTXMEN_FCT_SENDEVENT;
ctx_men_sub_entry[0].param1 = (long)K_h3;

ctx_men_sub_entry[1].text = "Artikelstamm";
ctx_men_sub_entry[1].bmp_fname = "nsigrp/ctxbmp/ARTIKEL.bmp";
ctx_men_sub_entry[1].flags = CTXMEN_LOOKUPKEYNAME;
ctx_men_sub_entry[1].fct = CTXMEN_FCT_SENDEVENT;
ctx_men_sub_entry[1].param1 = (long)K_h4;
show_context_menu(&ctx_men);
```

Das Beispiel definiert ein Menü mit den zwei Menüpunkten Zeitstempel und Telefonliste und einem Menüpunkt Weitere, der ein Untermenü öffnet. Das Untermenü enthält die Punkte Kundenstamm und Artikelstamm. Alle Menüpunkte senden beim Auslösen ein *FIX*-Event (Events K_h1 - K_h4) an die Anwendung. Die Angabe von CTXMEN_-LOOKUPKEYNAME bewirkt das Anzeigen der dem Event zugeordneten Taste im Text des Menüpunktes, falls für das Event eine Taste in der Tastaturtabelle von *FIX/Win* definiert wurde. Vor dem Menüpunkt, der das Untermenü öffnet befindet sich ein Menüpunkt der einen Separator darstellt.

36 Der Zugriff auf Meldungstexte

Wie erwähnt erlaubt *FIX* die Ablage im Programm verwendeter Texte in separaten Dateien. Für die *anwendungsspezifischen* Texte ist dies die Datei `$FXHOME/messages`, die mit Hilfe des Programms **msgprep** aus einer Textdatei erzeugt wird (vgl. “Die Erstellung von Meldungstexten” auf Seite 125).

Auf einen in `messages` gespeicherten Text kann im Programm mittels der Nummer zugegriffen werden, die dem Text in der Datei zugeordnet wurde. Da **msgprep** die Datei `messages` mit einem “Inhaltsverzeichnis” versieht, erfolgt dieser Zugriff sehr effizient. Auch wird ein Text, auf den einmal zugegriffen wurde, im Speicher gehalten, und erneute Zugriffe auf den gleichen Text liefern nur noch den entsprechenden Zeiger.

Ein erläuternder Text zu Objekten und ihren Elementen kann einfach durch Zuordnung der entsprechenden Nummer spezifiziert werden (Prompt), wobei *FIX* das Einlesen und Anzeigen des Textes übernimmt.

Mittels der Funktion

```
char  *fxlapptxt(long n)
```

kann der Entwickler im Programm aber auch explizit auf einen bestimmten Text zugreifen. Kann `messages` nicht gelesen werden oder existiert zu der angegebenen Nummer kein Text, liefert die Funktion einen Zeiger auf einen konstanten String der Länge 0.

Hinweis:

In `fx_texte` vorkommende Ersatzdarstellungen und Umschaltcodes wie ^A bzw. \t werden bereits von **msgprep** in die entsprechenden Kontrollzeichen (siehe hierzu Seite 248) umgesetzt. Tastenbezeichnungen wie #ku hingegen werden, da terminalabhängig, erst beim Einlesen aus `messages` durch die in der Tastenbeschreibung angegebenen Tastenbeschriftungen ersetzt.

Ein Überschreiben des von fxlapptxt() zurückgegebenen Strings ist nicht zulässig.

37 Meldungen

1 Allgemeines

Zur Ausgabe von Meldungen stellt *FIX* dem Entwickler den gleichen Mechanismus zur Verfügung, der intern verwendet wird. Es handelt sich dabei um einen Satz von Funktionen, die alle, eventuell von einem Warnton begleitet, ein dynamisch dimensioniertes Window bei der vorletzten Bildschirmzeile aufblenden und darin einen zuvor aufbereiteten Text ausgeben.

void	fflash(int type, char *fmt, ...)
void	oflash(obj *objp, int type, char *fmt, ...)
void	l_msg(int type, char *fmt, ...)
Event	msg(int type, char *fmt, ...)
Event	rt_msg(int type, char *fmt, ...)
BOOLEAN	jn_msg(int type, char *fmt, ...)

oflash und fflash() geben eine Meldung aus, die vom Anwender nicht quittiert werden muss. Die Meldung verschwindet, sobald *FIX* die nächste Eingabe gelesen hat.

l_msg() verfährt wie fflash(). Die Meldung verschwindet jedoch erst beim nächsten Aufruf der Funktion clrmsgln() oder bei einer neuen Meldung.

msg() wartet im Gegensatz zu fflash() auf eine Taste, bringt anschließend die Meldung zum Verschwinden und gibt das empfangene Tastenereignis zurück. Die Tasten sh und g1 werden allerdings lokal behandelt, d.h. hier wird nach der entsprechenden Aktion wieder in den Wartezustand zurückgekehrt.

rt_msg() verfährt wie msg(), wartet aber, bis die Taste RT gedrückt wird.

jn_msg() verfährt wie msg(), wartet aber, bis eines der als Darstellung von Ja und Nein definierten Zeichen eingegeben wird. Für Ja gibt die Funktion TRUE, für Nein FALSE zurück.

Alle Funktionen platzieren die Schreibmarke wieder dort, wo sie sich beim Aufruf der Funktion befand. Meldungen, die über die Bildschirmbreite hinausgehen, werden (ohne Rücksicht auf Wortgrenzen) gefaltet.

Der Parameter *type* steuert Vortext und Warnton und muss eines der folgenden Makros sein:

NOBEEP	kein Warnton, kein Vortext	
OHNE	Warnton, kein Vortext	
ACHTUNG	Warnton, Vortext “ACHTUNG: ”	
FEHLER	Warnton, Vortext “FEHLER: ”	
WARNUNG	Warnton, Vortext “WARNUNG: ”	
PROTO	kein Warnton, Vortext “PROTO: ”	(für Testzwecke !)
AUSWAHL	kein Warnton, Vortext “AUSWAHL: ”	

Die Vortexte sind der Datei `$FXDIR/runtime/messages` entnommen.

Der zweite Parameter, *fmt*, ist ein Formatstring im Sinne der C-Funktion printf(). Die übrigen Parameter sind Argumente für die im Formatstring enthaltenen Formatelemente (%s, %d, %f, %c usw.).

Der formatierte Text darf einige Sonderzeichen enthalten, die von *FIX* bei der Ausgabe interpretiert werden:

'\001'	Videoattribut “normal” setzen
'\002'	Videoattribut “invers” setzen
'\003'	Videoattribut “unterstrichen” setzen
'\004'	Videoattribut “halbhell” setzen
'\005'	Videoattribut “blinkend” setzen
'\016'	in Semigrafik-Zeichensatz wechseln
'\017'	in Standard-Zeichensatz wechseln
'\020'	Videoattribut und Zeichensatz speichern
'\021'	gespeichertes Videoattribut und Zeichensatz restaurieren
'\b'	Schreibmarke eine Stelle nach links bewegen
'\r'	Schreibmarke zum Anfang der Zeile bewegen
'\t'	Schreibmarke zur nächsten Tabulatorposition bewegen

Die nicht darstellbaren Zeichen ‘\024’ (DC4) und ‘\025’ (NAK) werden von *FIX* als interne Markierungen verwendet und dürfen daher im Meldungstext nicht vorkommen.

Durch Aufruf der Funktion

```
void    clrmsgln(void)
```

kann das Meldungsfenster zum Verschwinden gebracht werden.

2 Platzierung des Meldungsfensters

Die Platzierung des Meldungsfensters (Ausgabefenster für Funktionen wie l_msg(), fflash(), usw.) kann über die Ressourcen MsgWindowXPos und MsgWindowYPosOffset erfolgen. MsgWindowXPos muss zwischen 0 und der Bildschirmbreite liegen. Überstehende Meldungen werden abgeschnitten. MsgWindowYPosOffset wird auf den Standardwert addiert. Negative Werte verschieben die Position nach oben, positive nach unten. Wird das Meldungsfenster vollständig außerhalb des Bildschirms platziert, so wird eine Meldung nicht mehr angezeigt. Dieses Verhalten ist problematisch, vor allem bei Meldungen die eine Eingabe fordern (z.B. jn_msg()). Deshalb sollte dieser Zustand vermieden werden.

3 Darstellung von Meldungen

Durch Bereitstellung einer eigenen Funktion besteht die Möglichkeit, eine Meldung selbst auszugeben. Dieses Feature erlaubt es, Meldungen mittels fx_transfer_to_frontend() an das Frontend weiterzugeben oder Meldungen in eine Logdatei zu protokolieren. Die Adresse der eigenen Funktion ist in der globalen Variablen S_ShowMessage einzutragen. Sie muss folgenden Prototyp (aus `msg_pto.h`) entsprechen:

```
int showMessage (int typ, int art, int screenstate, char* msg, BOOLEAN* processed)
```

Der Type der Message wird in dem Parameter typ mitgegeben. Er definiert den Aufrufer der Funktion. Folgende Werte sind möglich:

Table 1: Zuordnung des Meldungstyps zum Funktionsnamen

Wert	0	1	2	3	4	5	6	7
Aufrufer	msg	rt_msg	jn_msg	err_msg	fflash	flash	o_flash	l_msg

Der Parameter art entspricht dem beim Aufruf der Messagefunktion mitgegebenen Wert - ist also eines der Macros NOBEEP, OHNE, ACHTUNG, ...

screenstate definiert den Zustand, in dem sich der Bildschirm befindet:

- 1 - keine *FIX*-Bildschirmverwaltung, keine Standardeingabe.
- 2 - keine *FIX*-Bildschirmverwaltung, Standardeingabe vorhanden.
- 3 - *FIX*-Bildschirmverwaltung initialisiert.

msg enthält den bereits formatierten Text der Meldung. Die der Messagefunktion mitgegebenen weiteren Parameter wurden dort schon an der entsprechenden Stelle eingesetzt.

Über den Parameter processed besteht die Möglichkeit den weiteren Ablauf zu bestimmen. Wird an dieser Adresse TRUE eingetragen, dann sieht *FIX* die Meldung als ausgegeben an und gibt den Rückgabewert von (*S_ShowMessage)() als Event an den Aufrufer zurück. Vorher erfolgt jedoch ein Prüfung des Events:

Beim Messagetype 1 oder 3 muss K_RT zurückgegeben werden.

Beim Messagetype 2 muss einer der in FXTRUTH definierten Werte zurückgegeben werden. Dabei sind Groß- und Kleinbuchstaben erlaubt.

Werden diese Forderungen nicht erfüllt, wird (*S_ShowMessage)() nochmals aufgerufen.

Wird an der Adresse processed kein anderer Wert oder FALSE hinterlegt, greift bei der Rückkehr der normale Mechanismus von *FIX* zur Ausgabe der Meldung in der Meldungszeile. Der Rückgabewert bleibt unberücksichtigt.

3.1 Verhinderung einer Rekursion

Beim Aufruf von (*S_ShowMessage)() kann es zu einer Endlosrekursion kommen, wenn Funktionen von *FIX* aufgerufen werden, die selber wieder eine Meldung ausgeben. Diese Rekursion wird von *FIX* verhindert. Das bedeutet, dass Meldungen immer über den normalen Mechanismus in der Meldungszeile ausgegeben werden, wenn *FIX* feststellt, dass keine Rückkehr aus (*S_ShowMessage)() erfolgt ist. (*S_ShowMessage)() wird also nicht nochmals aufgerufen.

3.2 Widersprüchliche Meldungen

Die Ausgabe von Meldungen in der Dialogbox von Windows ist nur für rt_msg und jn_msg sinnvoll, da nur diese Meldungen einen modalen Charakter haben, der dem der Dialogbox entspricht. Deshalb werden andere Meldungen (l_msg, fflash,..) in der Meldungszeile ausgegeben. Somit ist es möglich, dass der Benutzer zwei widersprüchliche Meldungen gleichzeitig sieht.

Beispiel

Der Benutzer gibt in einem Feld keinen Wert ein und erhält einen Hinweis von *FIX* in der Meldungszeile. Daraufhin gibt er einen Wert ein und bekommt eine Meldung in einer Dialogbox, dass der eingegebene Wert falsch sei. Diese Meldung überschreibt nicht die Meldungszeile und der Benutzer sieht zwei widersprüchliche Meldungen.

Dieses Beispiel tritt in der Praxis nicht auf, da die erste Meldung von *FIX* mit fflash() ausgegeben wird und durch Drücken der ersten Taste zur Eingabe des Wertes entfernt wird. Es sind jedoch ähnliche Situationen denkbar, auf die die Anwendung hin untersucht werden sollte, wo beispielsweise zwischen den Meldungen kein Tastendruck liegt oder l_msg zur Ausgabe der ersten Meldung verwendet wird.

4 Anzeige der Tastenhilfe

Mit Hilfe der neuen *FIX*-Funktion

```
void show_keyhelp(BOOLEAN on)
```

kann die Anzeige der Tastenhilfe angeschaltet (Wert für on = TRUE) oder ausgeschaltet (Wert für on = FALSE) werden.

Diese Funktion darf nicht bei Hintergrundprozessen benutzt werden, da sie den virtuellen Bildschirm modifiziert.

Der Text der Tastenhilfe kann mit Hilfe der Funktion

```
char* get_keyhelp_text()
```

ausgelesen werden. Sie liefert einen Zeiger auf den entsprechenden Meldungstext, der je nach Nummer des Prompttextes mit der Funktion fxlsystxt() oder fxlapptxt() ermittelt wurde. Enthält der Text Tastenlabels, dann haben diese folgendes Format:

```
\020\024(xx)\025\021
```

Die oktal angegebenen Zeichen sind Steuerzeichen für den Bildschirm. \020 speichert den aktuellen Zustand des Bildschirms. \021 stellt ihn wieder her. \024 markiert den Start des Tastenlabels. \025 markiert das Ende. Zwischen den Klammern steht die Eventbezeichnung des Tastenlabels (xx). Um den Eventcode zu ermitteln kann die Funktion

```
Event findkey(char* key)
```

verwendet werden. Sie bekommt die zweistelligen Bezeichnung der Taste (also ohne die Klammern) als Argument.

Die Prototypen der beiden neuen Funktionen werden in der Datei `proto/keyhelp_pto.h` definiert. Diese Header-Datei wird von der Datei `fix/libfix.h` eingebunden.

38 Spezielle Features

1 Vorrang-Dateiendungen

1.1 Konzept

Die Anwendung kann eine Menge von Definitionen der Form

$$ext{:}suf_1,\ldots,suf_n$$

vorgeben und dadurch *FIX* veranlassen, in gewissen Situationen beim Zugriff auf einen Pfad *path* mit der Endung *ext* zunächst sukzessive nach einer existierenden Datei $pathsuf_k$ zu suchen. Existiert diese, wird sie anstelle von *path* verwendet. Die angegebenen Suffixe haben dabei absteigende Präferenz.

Mehrere Definitionen sind durch ‘;’ zu trennen. Definitionen mit gleicher rechter Seite können zusammengefasst werden, etwa

```
.sel,.cho:.ora
```

Auch ein leeres Muster für die Endung ist möglich, wie bei

```
:-Inf
```

Bei überlappenden Mustern greift die Definition mit dem längeren Muster. Kein Muster darf mehr als einmal vorkommen.

Achtung:

Der Einsatz dieses Features verlangt vom Entwickler eine hohe Disziplin bei der Wahl von Dateinamen. Hilfreich sein kann der Mechanismus insbesondere für Anwendungen mit Selos, die nicht portable SQL-Anweisungen enthalten.

Von den *FIX*-Tools unterstützen z.Z. lediglich **mdemo** und **perfix** (nicht aber **perfix-p**, **led** und **rled**) dieses Feature.

Die V*orrang-Suffixe* werden von *FIX* berücksichtigt bei Pfadangaben zum

- Laden von Masken, Menüs, Choices, Selos
- Lesen von Daten für Choices aus Datei
- Lesen (Taste HP, → help()) und Schreiben (Taste K_g8 im Entwicklermodus) von Hilfetext-Dateien *der Anwendung*

darüberhinaus bei den Argumenten der Funktionen

- expandfile()
- expand_fopen()

jedoch nicht, wenn das “Suchumfeld” für *path* $FXDIR oder $FXRUNTIMEDIR ist.

Unberücksichtigt bleiben die Vorrang-Suffixe auch dann, wenn

- der angegebene Pfad existiert, aber keine reguläre Datei bezeichnet,

oder

- der Pfad oder eine seiner Komponenten unzulässig lang sind (Fehler ENAMETOOLONG)

oder

- eine führende Komponente des Pfades kein Verzeichnis ist (Fehler ENOTDIR) oder das Verzeichnis nicht durchsucht werden kann (Fehler EACCES)

sowie, wenn einer der beiden letztgenannten Fehler bei der Suche nach einer geeigneten Dateiendung auftritt.

Regelmäßig ignoriert werden die Vorrang-Suffixe beim Zugriff auf

- Layoutbeschreibungen und binäre Layouts
- Klartext- und kompilierte Meldungsdateien
- Anwender-Hilfetexte (Tasten g6, g8;→ chelp())

und

- die Protokolldatei (→ S_logfile)

Hinweis

Die Zeichen ‘;’, ‘:’ und ‘,’ können in Mustern oder Suffixen nicht benutzt werden, da sie stets als Separatorzeichen interpretiert werden.

1.2 Library

Die Definition von Vorrang-Dateiendungen erfolgt durch Aufruf der Funktion

```
int     fxSetPreferredFileSuffixes(char *defstring)
```

in der Anwendung. Die Funktion gibt 0 zurück, wenn *FIX* die Definitionsmenge akzeptiert.

Zum Löschen der augenblicklich gültigen Suffixe muss für *defstring* (char *)0 benutzt werden.

Bei einem unzulässigem Argument liefert die Funktion einen Wert kleiner 0 und die Suffixe zum Zeitpunkt des Aufrufs bleiben weiter gültig.

Sind Vorrang-Suffixe definiert, wird der Ablauf der Pfadinterpretation nach stderr protokolliert, sofern das Programm mit dem Schalter -test 32768 gestartet wird.

Wird eine Datei mit Suffix benutzt, so gilt:

- die Komponente ob_mfo_file der Objekt-Datenstruktur enthält den Namen inkl. des Suffixes
- Meldungen, die den Dateinamen beinhalten, geben diesen in der Regel mit Suffix aus

1.3 mdemo und perfix

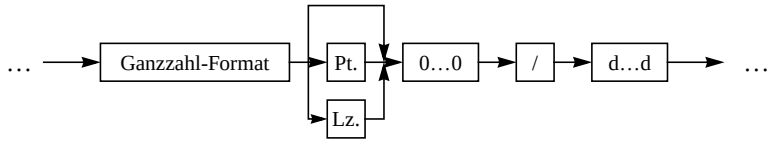
Um **mdemo** und **perfix** zu veranlassen, Vorrang-Dateiendungen zu berücksichtigen, müssen diese als Wert der Umgebungsvariable FXFILESUFFIXES hinterlegt werden.

Beispiel:

```
FXFILESUFFIXES=".sel:.ora.sel;,cho:.ora.cho" ; export FXFILESUFFIXES
```

2 Darstellung numerischer Werte als Brüche

FIX unterstützt in Masken und Selos Formate für numerische Felder bzw. Ergebnisspalten, die die Darstellung von Zahlen in Form von Vielfachen und Bruchteilen eines vorgegebenen Teilers erlauben.[1]



Hierbei muss gelten:

0…0	Folge von bis zu 9 Nullen, die als Platzhalter für den Zähler dienen
d…d	Folge von *gleichvielen* Ziffern, die den Teiler (größtmöglichen Nenner) n_{max} definiert; der Wert muss größer als 1 sein.
Ganzzahl-Format	numerisches Format aus den Formatzeichen '+' , '-' , ',' , '#' , '∗' , '&' mit 1 bis 32 Ziffernpositionen; das letzte Zeichen muss von ',' verschieden sein.
Pt.	Trennzeichen (→ ispunct()) mit Ausnahme der obigen Formatzeichen
Lz.	Leerzeichen

Im Formatstring kann obiges Muster von beliebigen[2] Zeichenfolgen umgeben sein, die jedoch selbst kein derartiges Muster enthalten dürfen. Sie werden einfach in die Ausgabe übernommen.

Um einen Wert im *Bruchteil-Format* darzustellen, bestimmt *FIX* zwei Zahlen g (dec_t mit Scale 0) und z (integer, $0 \le z < n_{max}$) so, dass $g + z / n_{max}$ dem Absolutbetrag des Wertes möglichst nahekommt, und eliminiert dann gemeinsame Teiler von z und n_{max}, d.h. "kürzt den Bruch". Das Vorzeichen des Wertes wird als Bestandteil des ganzzahligen Anteils geführt:

-1.0625 = $-(1 + {}^{2}/_{32})$ → "-1 1/16"

Ist der Zähler 0, werden Zähler und Nenner nicht dargestellt:

1 = $1 + {}^{0}/_{32}$ → "-1 / "

Immer, wenn *FIX* auf ein numerisches Format trifft, wird zunächst geprüft, ob es sich um ein Format obiger Form oder eines konventioneller Art handelt.

Bei der Eingabe in ein derartiges Feld werden ganzzahliger Anteil, Zähler und Nenner ähnlich den Komponenten von FXINVTYPE-Feldern komponentenweise bearbeitet. Durch Eingabe von '.' kann die nachfolgende (bzw. wieder die erste) Komponente - *Subfeld* - angesprungen werden. Wird unmittelbar nach dem Betreten eines Subfeldes eine Ziffer eingegeben, ersetzt sie den gesamten Wert des Subfeldes; anderenfalls wird sie hinter der Position der Schreibmarke eingefügt. Die Eingabe von '-' vertauscht wie üblich das Vorzeichen.

Das Feld kann - sofern nicht leer - nur verlassen werden, wenn alle drei Subfelder plausible Werte haben (Ausnahme: ist der Zähler 0 oder leer, darf auch der Nenner leer sein). Es muss dann gelten:

- $1 < n \le n_{max}$, n Teiler von n_{max}
- $0 \le z < n$
- g entsprechend dem Ganzzahl-Format darstellbar.

Zur Umwandlung von Zahlen in eine den Bruch darstellende Zeichenkette bietet *FIX* die Funktion

```
int     fxfracfmtdec(dec_t *np, char *format, char *outbuf)
```

an.

---

1. Grund hierfür ist die Forderung, Maßangaben außerhalb des internationalen Einheitensystems, beispielsweise ${}^{1}/_{32}$", darstellen zu können.

2. Aus technischen Gründen darf eine Formatangabe in einer Maskenbeschreibung nicht mit einer Ziffer beginnen.

3 Ausblenden von Selo-Windows

Aufgrund seiner Dimension kann ein Selo manchmal Teile der darunter liegenden Maske überdecken, die der Benutzer gerne noch einmal sehen möchte. Daher bietet *FIX* dem Entwickler die Möglichkeit, ein Tasten-Event vorzugeben, mit dem das Selo kurzzeitig unsichtbar gemacht werden kann. In der Auswahlphase der Selobearbeitung führt das Erkennen des in

```
Event       S_HideSeloKey          /* Vorbelegung: NOTHING */
```

hinterlegten (Tasten-)Events dazu, dass *FIX* das Selo bis zum Erkennen eines beliebigen weiteren Events nicht am Bildschirm darstellt. Bei Verwendung der Event-Kodierung eines darstellbaren Zeichens als Wert von S_HideSeloKey greift dieser Mechanismus nicht.

4 Code-gesteuerte Benutzung von Selos mittels fxse-API

Das fxse-API erlaubt die Nutzung der mit einem Selo verbundenen Information zur Datenermittlung und -übernahme durch Programmcode. Das Selo nimmt hierbei zwar intern dieselben Zustände wie bei einer Bearbeitung an, wird aber nicht am Bildschirm sichtbar.

Zur Nutzung des fxse-API ist die Header-Datei `fxse.h` einzuschließen, die die notwendigen Datenstrukturen und Macros definiert und Library-Funktionen deklariert. Dies sind:

- die Datenstrukturen struct fxse_cb und struct fxse_item
- eine Reihe von Fehlercode-Makros (FXSE_ILLEGAL_ARGUMENT, …)
- die Funktionen fxse_init() (Seite 346), fxse_start_search() (Seite 347), fxse_supply() (Seite 347), fxse_choose() (Seite 348) und fxse_finalize() (Seite 348)

Benutzt wird das API etwa wie folgt:

1. Durch fxse_init() wird das fxse-API reserviert[1]. Hierbei wird Code ähnlich dem ausgeführt, der gewöhnlich zwischen dem Aufruf von perform() und der ersten Interaktion mit dem Benutzer abläuft. Der Aufrufer erhält Zugriff auf eine vom API bereitgestellte Einzelsatz-Maske zur Hinterlegung von Startwerten, eine Memory Relation zur Aufnahme der Treffersätze und eine Info-Struktur.
2. fxse_start_search() simuliert die Datensuche. Der ausgeführte Code ähnelt dem, der beim interaktiven Gebrauch nach dem Anstoßen der Suche mittels f8 oder RT bis zum Anzeigen eines ersten Treffersatzes abläuft.
3. Mittels fxse_supply() können nun weitere Treffer bis zu einer vorgegebenen Anzahl gelesen werden. Gelesene Treffersätze werden an die oben erwähnte Memory Relation angefügt. fxse_supply() darf auch mehrfach hintereinander aufgerufen werden.

   Der ausgeführte Code ist ähnlich dem, der beim Vorwärtsblättern durch die Treffermenge abläuft.
4. fxse_choose() übernimmt einen Satz der Treffermenge als Auswahl. Dieser wird in den Auswahlpuffer des Selos übertragen, so dass seine Werte für spätere Zugriffe mittels fx_selo_fld_adr() zur Verfügung stehen. Optional werden auch die zu Spalten hinterlegten Wertübernahme-Beziehungen berücksichtigt.

   Der ausgeführte Code ähnelt dem, der beim interaktiven Gebrauch beim Auswählen eines Treffersatzes durch RT oder Angabe seines Markierung abläuft.
5. Durch fxse_finalize() wird das fxse-API freigegeben. Hierbei wird Code ähnlich dem ausgeführt, der gewöhnlich zwischen der die Selobearbeitung beendenden Interaktion mit dem Benutzer und der Rückkehr aus dem Aufruf von perform() abläuft.

Hinweis:

Der Aufruf von fxse_start_search() und der letzte (lesende) Aufruf von fxse_supply() müssen innerhalb derselben Datenbanktransaktion erfolgen.

1. Wegen der Nutzung statischer Datenstrukturen ist das fxse-API nicht *reentrant*.

5 Listener an Memory Relations

An einer Memory Relation kann ein Zeiger auf eine Struktur registriert und deregistriert werden, die vorwiegend Funktionszeiger enthält. Solange dieser *Listener* hinterlegt ist, ruft *FIX* bei gewissen Operationen auf der Memory Relation Funktionen aus der Struktur auf.

Die in der Header-Datei `fix/mrlistener.h` definierte Struktur hat folgende Form:

```
typedef struct mrlistener_struct {
    void    (*onBind)(MemRelType mrp, obj *objp);
    void    (*onUnbind)(MemRelType mrp, obj *objp);
    void    (*onRowsInserted)(MemRelType mrp, int firstRowPos, int lastRowPos,
                              unsigned long rowid);
    void    (*onDeletingRows)(MemRelType mrp, int firstRowPos, int lastRowPos,
                              unsigned long rowid);
    void    (*onCellsUpdated)(MemRelType mrp, int firstRowPos, int lastRowPos,
                              unsigned long rowid, int colno);
    char    *reserved;          /* for FIX internal use only */
} mrlistener_t;
```

Erstes Argument bei Aufrufen der Funktionen ist jeweils die / eine der Memory Relation(s), an der ein solcher Listener horcht.

Die einzelnen Funktionszeiger im Listener dürfen NULL sein (z.B. wenn die Anwendung diese Art von Information nicht benötigt; interessant etwa für onBind() und onUnbind()).

5.1 Aufrufe

5.1.1 Herstellen einer Bindung der Memory Relation an ein *FIX*-Objekt

FIX ruft *nach* dem Einrichten der Bindung die zuständige Funktion im Listener mit folgenden Argumenten auf:

1. der Memory Relation, die gebunden wurde,
2. dem Objekt, zu dem eine Bindung hergestellt wurde.

Das zweite Argument dient dem Programmierkomfort; es könnte auch durch Aufruf von fx_mrgetobject() ermittelt werden.

5.1.2 Lösen der Bindung der Memory Relation an ein *FIX*-Objekt

FIX ruft *vor* dem Lösen der Bindung die zuständige Funktion im Listener mit folgenden Argumenten auf:

1. der Memory Relation, deren Bindung gelöst wird,
2. dem Objekt, zu dem bislang eine Bindung bestanden hat.

Das zweite Argument dient dem Programmierkomfort; es könnte auch durch Aufruf von fx_mrgetobject() ermittelt werden.

5.1.3 Ein- und Anfügen eines Tupels

FIX ruft *nach* der Operation die zuständige Funktion im Listener mit folgenden Argumenten auf:

1. der Memory Relation, der ein Satz(-bereich) hinzugefügt wurde,

2. der ersten neuen Satzposition (*firstRowPos*),
3. der letzten neuen Satzposition (*lastRowPos*),
4. der ROWID des eingefügten Tupels (sofern *firstRowPos* = *lastRowPos*).

Es gilt 1 ≤ *firstRowPos* ≤ *lastRowPos* ≤ Anzahl Sätze.[1] Das Argument *rowid* dient dem Programmierkomfort und ist nur signifikant, wenn *firstRowPos* = *lastRowPos*.

Hinweis:

Nach einer Nachricht dieser Art werden typischerweise Nachrichten nach Abschnitt 5.1.5 für die einzelnen Zellen des eingefügten Tupels erfolgen.

5.1.4 Löschen eines Tupels

FIX ruft *vor* der Operation die zuständige Funktion im Listener mit folgenden Argumenten auf:

1. der Memory Relation, aus der ein Satz(-bereich) zur Löschung ansteht,
2. der ersten zur Löschung anstehenden Satzposition (*firstRowPos*),
3. der letzten zur Löschung anstehenden Satzposition (*lastRowPos*),
4. der ROWID des zur Löschung anstehenden Tupels (sofern *firstRowPos* == *lastRowPos*).

Es gilt 1 ≤ *firstRowPos* ≤ *lastRowPos* ≤ Anzahl Sätze.[2] Das Argument *rowid* dient dem Programmierkomfort ist nur signifikant, wenn *firstRowPos* = *lastRowPos*.

5.1.5 Wertänderung von Tupelzellen

FIX ruft *nach* der Operation, sofern sich neuer und vorgefundener Wert unterscheiden, die zuständige Funktion im Listener mit folgenden Argumenten auf:

1. der Memory Relation, in der der Wert einer Zelle (oder eines Bereichs von Zellen) geändert wurde,
2. der ersten von der Wertänderung betroffenen Satzposition (*firstRowPos*),
3. der letzten von der Wertänderung betroffenen Satzposition (*lastRowPos*),
4. der ROWID des von der Wertänderung betroffenen Tupels (sofern *firstRowPos* = *lastRowPos*),
5. der von der Wertänderung betroffenen Spaltenposition (*colno*).

Es gilt 1 ≤ *firstRowPos* ≤ *lastRowPos* ≤ Anzahl Sätze[3] und 1 ≤ *colno* ≤ Anzahl Spalten. Das Argument *rowid* dient dem Programmierkomfort ist nur signifikant, wenn *firstRowPos* = *lastRowPos*.

Hinweis

mr_qsort() ordnet die Sätze einer Memory Relation um, ohne dass Funktionen des Listeners angestoßen werden. Die Restrukturierung wird also weder als “Löschen von Sätzen und Einfügen an anderer Stelle” noch als “Update” auf Grund der Vertauschung von Zelleninhalten gemeldet.

1. Für dieses Release gilt sogar *firstRowPos* = *lastRowPos*, da *FIX* keine Operation zum Einfügen mehrerer Sätze kennt.
2. Für dieses Release gilt sogar entweder *firstRowPos* = *lastRowPos* (ein Satz wird gelöscht) oder 0 < Anzahl Sätze && *firstRowPos* = 1 && *lastRowPos* = Anzahl Sätze (alle Sätze werden gelöscht).
3. Für dieses Release gilt sogar *firstRowPos* = *lastRowPos*.

5.2 Registrierung und Deregistrierung eines Listeners

Hierzu dienen die in der Header-Datei `fix/mrlistener.h` deklarierten Funktionen

```
int           mr_addListener(MemRelType mrp, mrlistener_t *mrlp)
int           mr_removeListener(MemRelType mrp, mrlistener_t *mrlp)
mrlistener_t  *mr_getListener(MemRelType mrp)
```

39 Features beim Einsatz von FIX/Win

Ein *FIX*-Programm kann über ein Frontend bedient werden. Derzeit ist *FIX/Win* das einzige Frontend, dass zum Einsatz kommen kann. Der folgende Text beschreibt die speziellen Features, die nur mit dem Einsatz von *FIX/Win* genutzt werden können.

1 Allgemeine Funktionsweise

Bei der Bedienung eines *FIX*-Programms über *FIX/Win* ist das *FIX*-Programm von *FIX/Win* aus zu starten. Dazu ist in der Programmtabelle von *FIX/Win* ein Eintrag anzulegen, der ein Startscript zum Start der Anwendung festlegt. Das Script hat die Aufgabe, die Umgebung für das *FIX*-Programm aufzubauen und dann die Binärdatei zu starten. Als Parameter muss die Binärdatei dazu den Schalter

```
-service
```

bekommen. Dieser Schalter wird zum einen von *FIX* selbst ausgewertet. Zum anderen besteht die Möglichkeit in der Anwendung die globale Variable

```
BOOLEAN B_service = FALSE;
```

auszuwerten, die den Wert TRUE besitzt, wenn der Schalter -service angegeben wurde.

Der Aufbau der Verbindung von *FIX/Win* zu einem *FIX*-Programm geht über den Service rexec unter UNIX und über den Service fxsrv unter Windows. Diese Services arbeiten mit den Standardkanälen von Programmen (stdin, stdout, und stderr). Wenn das *FIX*-Programm gestartet wird, wird über stdout und stdin ein spezielles Protokoll gefahren. Deshalb dürfen diese Kanäle nicht vom Programm anderweitig genutzt werden. Ausgaben auf stdout sind deshalb strikt verboten. Ausgaben auf stderr werden von *FIX* an *FIX/Win* geleitet und im Meldungsfenster ausgegeben.

2 Icons

Über die bei Terminals verfügbare Funktionalität hinaus bietet *FIX/Win* die Möglichkeit, Events statt durch Tastendruck durch das Anklicken von Icons auszulösen. Dabei wird die zu einem bestimmten Zeitpunkt bereitgestellte *Iconlist* - darunter ist eine auf Frontend-Seite hinterlegte, geordnete Folge von Elementen zu verstehen, die jeweils u.a. aus einem Bezeichner, einer Bitmap und einer Event-Sequenz bestehen - durch eine numerische *Iconlist-ID* charakterisiert (Default ID ist 1). Um die Iconlist, z.B. bei einem Wechsel in ein anderes Modul der Anwendung, auszutauschen, stellt *FIX* die Funktion

```
int fx_iconlist(unsigned int *new_iclist_p, unsigned int *old_iclist_p)
```

bereit. Ein Aufruf von fx_iconlist() in einer Terminal-gestützten Anwendung gibt zwar einen Fehlercode zurück, ist aber ansonsten wirkungslos.

Abhängig vom Programmzustand ist meist nur ein Teil der in der Iconlist enthaltenen Elemente relevant insofern, dass das Anklicken zu diesem Zeitpunkt sinnvoll ist. Daher ermöglicht es *FIX*, einem Feld oder Menüpunkt analog zum Prompttext ein *Iconset* - charakterisiert durch eine numerische *Iconset-ID* - zuzuordnen. Ein Iconset ist eine auf Frontend-Seite hinterlegte, vordefinierte Menge von Element-Bezeichnern. Wenn *FIX* auf Eingabe in ein Feld wartet, wird

die mit diesem Feld assoziierte Iconset-ID an das Frontend weitergegeben; entsprechend wird mit der mit einem Menüpunkt assoziierten Iconset-ID verfahren, wenn das jeweilige Menü aktives Objekt ist und dieser Menüpunkt angewählt ist. Das Frontend benutzt die übermittelte Iconset-ID, um zu bestimmen, welche Elemente der Iconlist aktiv oder passiv sein sollen.

Zur Spezifikation der Iconset-ID enthält **led** ein Feld mit der Beschriftung “Iconset”; in der Beschreibungsdatei wird die Iconset-ID durch … **iconset** <unsigned> … angegeben.

Bei einer Terminal-gestützten Anwendung wird die in der Beschreibungsdatei mit einem Feld oder Menüpunkt assoziierte Iconset-ID nicht in die Datenstruktur übernommen; der Wert der entsprechenden Komponente ist undefiniert.

Automatische Umschaltung der Iconlist

Besitzt die globale Variable

```
BOOLEAN     S_RestoreDefaultIconlist;     /* Voreinstellung: FALSE */
```

einen Wert ungleich FALSE, so merkt sich *FIX*, wenn eine Choice, ein Selo oder ein Hilfetext zum aktuellen Objekt wird, die zu diesem Zeitpunkt gültige Iconlist und stellt die Default-Iconlist (ID 1) ein. Beim Verlassen des Objekts wird wieder die ursprüngliche Iconlist eingestellt.

3 An- und Auswahl mittels Mausklick

Eine von *FIX/Win* gesteuerte Anwendung muss darauf vorbereitet sein, die BT_... Events zu berücksichtigen, die ihr - analog zu einem Tastendruck - mitteilen, dass eine Maustaste betätigt wurde (bei Benutzung eines Terminals werden die BT_...-Events von *FIX* niemals generiert). Beim gegenwärtigen Stand der Implementierung fehlen einer Anwendungslogik - außer bei Masken - allerdings die Voraussetzungen, diese Events selbst zu behandeln, wie z.B. die Position des Mauszeigers abzufragen. Daher sollte in diesen Fällen eine Anwendungslogik die BT_...-Events lediglich durchreichen und *FIX* die Reaktion überlassen.

Die *FIX*-Library stellt eine Funktion bereit, um abzufragen, in welcher Form ein Objekt die Maus unterstützt (→ o_GetClickSensitivity()).

3.1 Menü

Zur Anwahl von Menüpunkten durch Klicken mit der linken Maustaste vgl. Kapitel *29*.

3.2 Maske

Zur Anwahl von Feldern bzw. Sätzen durch Klicken mit der linken Maustaste vgl. Kapitel *31*. Die *FIX*-Library stellt Funktionen bereit, um bei der Behandlung von BT_...-Events in der Anwendungslogik zu ermitteln,

- an welcher Stelle des Bildschirms sich der Mauszeiger zum Zeitpunkt eines Klickens befand (→ o_RetrievePositionClickedOn()),
- über welchem Feld welchen Satzes sich der Mauszeiger befand (→ o_RetrieveItemClickedOn()).

3.3 Selo

Empfängt *FIX* das Event BT_LEFT, während ein Selo aktiv ist, so wird hierdurch der Satz angewählt, über dem sich der Mauszeiger befindet; existiert kein solcher Satz, erfolgt ein akustisches Signal.

3.4 Choice

Empfängt *FIX* das Event BT_LEFT, während eine Choice aktiv ist, so wird hierdurch der Satz (de)selektiert, über dem sich der Mauszeiger befindet; existiert kein solcher Satz, erfolgt ein akustisches Signal.

3.5 Hilfetext

Empfängt *FIX* das Event BT_LEFT, während ein Hilfetext aktiv ist, wird es ignoriert: eine akustische Warnung erfolgt anders als bei einer Taste ohne Bedeutung nicht.

3.6 Meldung

Empfängt *FIX* das Event BT_LEFT, während auf die Quittierung einer Meldung gewartet wird, wird es ignoriert: eine akustische Warnung erfolgt anders als bei einer Taste ohne Bedeutung nicht.

4 Validierung der Verbindung

Ein “Absturz” von *FIX/Win* bzw. WINDOWS oder das Ausschalten des PCs ohne vorheriges Verlassen der *FIX*-Anwendung kann dazu führen, dass der Anwendungsprozess auf dem UNIX-Host weiterläuft, ohne jemals wieder bedient zu werden. Damit ein solcher Prozess automatisch, d.h. ohne Eingriff eines Administrators, terminiert, fordert die *FIX*-Anwendung vom Frontend ein Lebenszeichen an, wenn über einen längeren Zeitraum Eingabe aussteht. Trifft dieses Lebenszeichen nicht innerhalb eines bestimmten Zeitraums ein, wird angenommen, dass das Frontend die Anwendung nicht mehr bedienen kann, und die Anwendung wird beendet.

Zur Einstellung des Zeitraums können zwei Umgebungsvariablen gesetzt werden:

Variable	Bedeutung
FXREADTIMEOUT0	Zeit in Sekunden, nach der vom Frontend ein Lebenszeichen angefordert wird, wenn keine Eingabe ansteht (Default: 3600)
FXREADTIMEOUT1	Zeit in Sekunden, innerhalb derer das Lebenszeichen des Frontend eintreffen muss (Default: 120)

Ohne Lebenszeichen wird die Anwendung durch fastexit(127, "…") abgebrochen.

5 Übertragen von Daten an FIX/Win

Zum Übertragen von Daten an *FIX/Win* kann die Funktion

```
int fx_exec_frontend(char *str)
```

verwendet werden. Sie übermittelt die Daten in str an *FIX/Win* und *FIX/Win* gibt diese Daten an die Funktion

```
FWOWN_API char* CALLBACK FwownExec(CallBackFkt, void* p_callID, char* )
```

der Benutzerbibliothek weiter. Diese Funktion ist vom Anwendungsentwickler so zu gestalten, dass sie die von der Anwendung gesendeten Daten auswerten kann. Eine ausführliche Beschreibung ist im Handbuch zu *FIX/Win* zu finden.

6 Übertragen von binären Daten

Die *FIX*-Funktion fx_exec_frontend() besitzt folgende Beschränkungen:

- Es können keine binäre Daten übertragen werden.
- Die Menge der Daten ist begrenzt. (interner Puffer von *FIX/Win*)
- Es können keine Daten vom Frontend zum Backend übertragen werden

Deshalb gibt es eine weitere *FIX*-Funktion, die binäre Daten beliebiger Länge senden kann:

```
int fx_transfer_to_frontend(int bytes, char* buff)
```

Sie bekommt als Parameter die Anzahl der Bytes, die zu senden sind und einen Zeiger auf eine Adresse von der diese Bytes gelesen werden sollen. Als Rückgabewerte liefert die Funktion 0 oder -1, wenn kein Frontend die Anwendung bedient.

Das Frontend legt die so empfangenen Daten in einem Puffer ab und wartet, bis *FIX* eine Taste anfordert. Erst jetzt ist es in der Lage eine Antwort auf die Anforderung zurückzusenden, die dann von *FIX* gelesen wird. Vorher wird jedoch der Puffer der Funktion

```
FwownProcessData(CallBackFkt CallBack, long* bytes, char** data)
```

zur Verarbeitung angeboten. Diese Funktion muss der Anwendungsprogrammierer in der Benutzerbibliothek von *FIX/Win* bereitstellen. Der Parameter bytes enthält einen Zeiger auf einen Longwert, der die Länge der Daten enthält. Die Adresse der Daten wird in dem Parameter data hinterlegt. Beide Werte können von der Methode geändert werden, so dass sich sowohl die Größe als auch der Inhalt des Datenpuffers ändern kann. Wenn *FIX* die Daten empfangen hat, dann wird das Event

```
K_SB
```

an die Logik der Anwendung gesendet. Die Anwendung kann daraufhin mit Hilfe der Funktion

```
int getFrontendData(char** buff)
```

auf den Puffer mit den gesendeten Daten zugreifen. Der Zeiger auf den Puffer wird in dem Speicherbereich abgelegt, der durch den ersten Parameter spezifiziert wird. Als Rückgabewert liefert die Methode die Größe des Puffers. Der Puffer sollte entweder direkt verarbeitet werden oder in einen anderen Speicherbereich kopiert werden, da er unmittelbar vor dem Einlesen des nächsten Events von *FIX* wieder freigegeben wird.

Dieses komplizierte Vorgehen ist darin begründet, dass *FIX/Win* und insbesondere die *FIX*-Applikation nur über einen Thread verfügen. So ist es nicht möglich, dass die Methode fx_transfer_to_frontend() die bearbeiteten Daten zurückliefert, da durch das Warten auf die Daten, der einzige Thread, der die Daten liest, blockiert wäre.

7 Anzeige der Schreibmarke

Ab *FIX/Win* 3.0.0 wird die Schreibmarke nur noch in Masken, Tabellen und Choices eingeblendet. In allen anderen Objekten wird sie ausgeblendet. Dazu wird unmittelbar vor der Abarbeitung eines Objektes (perform) der aktuelle Zustand gerettet und danach umgeschaltet. Nach der Abarbeitung wird der gerettete Zustand wieder hergestellt.

Während der Abarbeitung kann die Funktion

```
void frontend_show_caret(int on)
```

aufgerufen werden, um die Schreibmarke explizit an- (on==TRUE) oder auszuschalten (on==FALSE).

40 Zeichensatz

Entstehungsbedingt benutzen UNIX und die meisten darauf aufsetzenden Anwendungen den ASCII-Zeichensatz, dem jedoch die in europäischen Sprachen weitverbreiteten akzentuierten Zeichen fehlen. So entstanden zunächst länderspezifische Abarten, die darin enthaltene, aber selten benutzte Zeichen umnutzten, so beispielsweise die deutsche Variante des 7Bit-Zeichensatzes ISO 646:1983, die *FIX* bis Version 2.9.0 bei der Interpretation von Zeichen stillschweigend zugrundelegte. Darin werden die ASCII-Zeichen '@', '[', '\', ']', '{', '|', '}', '~' als '§', 'Ä', 'Ö', 'Ü', 'ä', 'ö', 'ü', 'ß' benutzt.

In den letzten Jahren unterstützen die verschiedenen UNIX-Derivate jedoch durchweg diverse 8Bit-Zeichensätze, für westeuropäische Sprachen beispielsweise den Zeichensatz ISO 8859:15.

Daher verfährt *FIX* wie folgt: Innerhalb gewisser Restriktionen kann in einer Definitionsdatei jeder Zeichencode 0 bis 255 benannt und gewissen Zeichenklassen zugeordnet werden. Diese Zeichenklassen entsprechen den unter UNIX üblichen:

L[OWER]	Kleinbuchstabe
U[PPER]	Großbuchstabe
D[IGIT]	Ziffer
X[DIGIT]	Hexadezimalziffer
P[UNCTATION]	Satzzeichen
B[LANK]	Leerzeichen
S[PACE]	(formatierender) Leerraum
C[ONTROL]	Kontroll-/Steuerzeichen

Folgende, dem ASCII-Zeichensatz entsprechende Zuordnung (*Basiszeichensatz*) ist vordefiniert:

Zeichen	Hex.-Code	Eigenschaften
NUL	00	C
SOH	01	C
STX	02	C
ETX	03	C
EOT	04	C
ENQ	05	C
ACK	06	C
BEL	07	C
BS	08	C
HT	09	C, S
NL	0A	C, S
VT	0B	C, S
NP	0C	C, S
CR	0D	C, S
SO	0E	C
SI	0F	C
DLE	10	C
DC1	11	C

DC2	12	C
DC3	13	C
DC4	14	C
NAK	15	C
SYN	16	C
ETB	17	C
CAN	18	C
EM	19	C
SUB	1A	C
ESC	1B	C
FS	1C	C
GS	1D	C
RS	1E	C
US	1F	C
SP	20	B
!	21	P
“	22	P
#	23	P
$	24	P
%	25	P
&	26	P
’	27	P
(	28	P
)	29	P
*	2A	P
+	2B	P
,	2C	P
-	2D	P
.	2E	P
/	2F	P
0	30	D, X
…		
9	39	D, X
:	3A	P
;	3B	P
<	3C	P
=	3D	P
>	3E	P
?	3F	P
@	40	P
A	41	U, X
…		
F	46	U, X
G	47	U
…		
Z	5A	U
[	5B	P
\	5C	P
]	5D	P
^	5E	P

_	5F	P
‘	60	P
a	61	L, X
…		
f	66	L, X
g	67	L
…		
z	7A	L
{	7B	P
|	7C	P
}	7D	P
~	7E	P
DEL	7F	C
	80	
…		
	FF	

Als Zeichenbenennung wird für die Zeichen *x* von ‘!’ bis ‘~’ “*@x*”, für das Leerzeichen ”SP” verwendet. Alle anderen Benennungen sind undefiniert.

Die Klassenzugehörigkeit der grau hinterlegt dargestellten Zeichen darf in der Definitionsdatei verändert werden, wobei allerdings nur die Eigenschaften L, U, P und C vergeben werden können.

Um zu testen, ob ein Zeichen einer bestimmten Klasse angehört, stellt die *FIX*-Library Funktionen bereit, die konzeptionell den Makros aus der Standard-Header-Datei `ctype.h` entsprechen:

int fxisupper(int code)	U
int fxislower(int code)	L
int fxisdigit(int code)	D
int fxisxdigit(int code)	X
int fxisalpha(int code)	U | L
int fxisalnum(int code)	U | L | D
int fxispunct(int code)	P
int fxisblank(int code)	B
int fxisspace(int code)	B | S
int fxiscntrl(int code)	C
int fxisgraph(int code)	U | L | D | P
int fxisprint(int code)	U | L | D | P | B

int fxtolower(int code)
int fxtoupper(int code)

Wird ein vom Basiszeichensatz abweichender Zeichensatz verwendet, dürfen anstelle der Makros aus `ctype.h` nur noch obige Funktionen verwendet werden.

Definition eines Zeichensatzes

FIX enthält im Verzeichnis `$FXDIR/etc` Unterverzeichnisse für drei gängige Zeichensätze:

- `ISO-15` (entspricht dem Zeichensatz ISO 8859:15)
- `ISO-2` (entspricht dem Zeichensatz ISO 8859:2)
- `IBM437` (entspricht dem IBM-Zeichensatz CP437)

- GERMAN7 (entspricht der deutschen Erweiterung von ISO 646:1983)

Jedes dieser Verzeichnisse enthält eine Zeichensatz-Definitionsdatei `charset`.[1,2]

Verwendet wird der Zeichensatz im Unterverzeichnis `$FXCHARSET` (diese Umgebungsvariable wird bei der Installation in `etc/stdprofile` eingetragen). Das Unterverzeichnis wird entlang der in der Umgebungsvariablen FXCHARSETPATH angegebenen Verzeichnis-Liste, sonst in `$FXDIR/etc` gesucht.

In der Datei `charset` können modifizierbaren Codes selbsterklärende *Zeichenbenennungen*, z.B. Copyright_Symbol, zugeordnet werden. Die Datei muss aus Zeilen folgender Form bestehen:

<Zeichenbenennung> = <Code> [, <Eigenschaft>]*

wobei

<Zeichenbenennung>	::=	Identifier (max. 20 Zeichen)
	|	@7Bit-Zeichen
<Code>	::=	Zahl zwischen 1 und 255
<Eigenschaft>	::=	U[PPER]
	|	L[OWER] <UpperCode>
	|	C[ONTROL]
	|	P[UNCT]
<UpperCode>	::=	Zahl zwischen 1 und 255

Zahlen können dezimal, hexadezimal (0xhh, 0Xhh) oder oktal (0ooo) angegeben werden.

Daran anschließen kann sich ein mit # eingeleiteter Kommentar, der sich dann bis zum Ende der Zeile erstreckt (auch reine Kommentarzeilen sind möglich).

Die Zeichenbenennungen müssen ebenso wie die Codes paarweise verschieden sein.[3] Nur Abweichungen vom Basiszeichensatz (vgl. Seite 265) müssen definiert werden. Beim Einlesen der Datei `charset` wird der Basiszeichensatz zugrunde gelegt.

Beispiele:

Der Umlaut ‘ä’ besitzt im IBM437-Zeichensatz den Code 132, der Umlaut ‘Ä’ den Code 142. Als Bezeichnung soll *ae* dienen. ‘ä’ fällt in die Klasse “Kleinbuchstabe” (LOWER) (und damit auch in die Klasse “alphanumerisches Zeichen”). Daraus ergibt sich

ae = 132, LOWER 142

In der deutschen Erweiterung von ISO 646:1983 werden für ‘ä’ und ‘Ä’ die Codes 0x7B bzw. 0x5B verwendet, die im Basiszeichensatz die Zeichen ‘[’ und ‘{’ bezeichnen. Durch die Zeile

ae = 0x7B, LOWER 0x5b

wird die ursprüngliche Klassenzuordnung für 0x7b überschrieben und *ae* als Bezeichnung eingeführt.

1. Der Name dieser Datei ist nicht veränderbar.
2. Achtung: die Zeichensatz-Definitionen ISO-15 und IBM437 definieren alle 8Bit-Zeichen mit Ausnahme der Codes der im Deutschen gebräuchlichen Umlaute als Kontrollzeichen.
3. Wird ein *FIX*-Programm mit dem Schalter -test 16384 gestartet, erfolgt eine Warnung, wenn ein Code, der bereits eine Zeichenbenennung besitzt, durch eine nachfolgende Definition umbenannt wird.

Hinweis:

`ISO-2/charset` definiert den Zeichensatz ISO 8859:2, der zusätzlich zu den ASCII-Zeichen vor allem in Osteuropa gebräuchliche Zeichen enthält. Wie `ISO-15/charset` weist allerdings auch `ISO-2/charset` 8Bit-Zeichen mit Ausnahme der Codes der im Deutschen gebräuchlichen Umlaute als Kontrollzeichen aus, so dass für die Unterstützung der polnischen oder tschechischen Sprache Anpassungen erforderlich sind. `ISO-2/charset.pl_pl` ist ein Muster hierfür.

41 Semigrafik

Bei der Darstellung von Feldwerten, der Layoutgestaltung, in Hilfetexten und in Meldungen unterstützt *FIX* neben dem Standard- einen geräteneutralen zweiten Zeichensatz, der vor allem für Grafikzeichen gedacht ist. *FIX* verwendet ihn vor allem für Ecken, Linien, Feldbegrenzer usw. Der vordefinierte Grafik-Zeichensatz kann zwar erweitert werden[1]; soll die Anwendung portabel sein, wird jedoch empfohlen, sich auf die Zeichen zu beschränken, die *FIX* zur eigenen Verwendung vordefiniert.

Die gerätespezifische Realisierung erfolgt gemäß der Bildschirmbeschreibung (vgl. Seite 511f): dort erwartet *FIX* zum einen die Steuersequenzen, mit denen zwischen Standard- und alternativem Zeichensatz des Terminals umgeschaltet wird, und zum anderen eine Abbildung von Zeichencodes auf Zeichen des Geräte-Zeichensatzes.

Folgende Codes verwendet *FIX* in einer feststehenden Bedeutung:

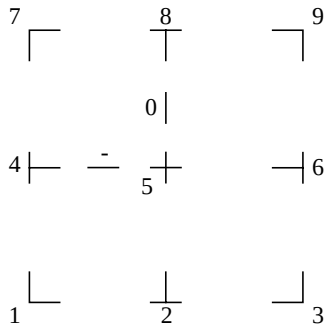
Code	Bedeutung	Default	von *FIX* benutzt
‘ ’	Leerraum	‘ ’	•
‘_’	Leerstelle im Feld	‘_’	•
‘^’	Füllstelle im Feld	‘^’	•
‘[’	linker Feldbegrenzer	‘[’	•
‘]’	rechter Feldbegrenzer	‘]’	•
‘(’	linker y/n-Feldbegrenzer	‘(’	•
‘)’	rechter y/n-Feldbegrenzer	‘)’	•
‘<’	linker Feldbegrenzer (scrolled)	wie Code ‘[’	•
‘>’	rechter Feldbegrenzer (scrolled)	wie Code ‘]’	•

Seit Version 2.9.2 differenziert *FIX* intern zwischen inneren (einfachen) und äußeren (“schattierten”) Linienelementen:

Einfache Linien

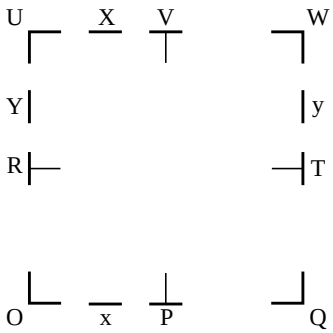
Code	Bedeutung	Default	von *FIX* benutzt
‘1’	untere linke Ecke	‘+’	•
‘2’	unteres T-Stück	‘+’	i. Allg. undefiniert
‘3’	untere rechte Ecke	‘+’	•
‘4’	linkes T-Stück	‘+’	i. Allg. undefiniert
‘5’	Kreuz	‘+’	i. Allg. undefiniert
‘6’	rechtes T-Stück	‘+’	i. Allg. undefiniert
‘7’	obere linke Ecke	‘+’	•
‘8’	oberes T-Stück	‘+’	i. Allg. undefiniert
‘9’	obere rechte Ecke	‘+’	•
‘-’	waagerechter Strich	‘-’	•
‘0’	senkrechter Strich	‘|’	•

1. Den Grafik-Zeichensatz für einzelne Anwendungen zu definieren, ist gegenwärtig nur durch separat abgelegte Bildschirm- und Tastenbeschreibungen möglich (siehe Seite 511 und Seite 513).



Schattierte Linien

Code	Bedeutung	Default	von *FIX* benutzt
‘O’	äußere untere linke Ecke	wie Code ‘1’	•
‘P’	äußeres unteres T-Stück	wie Code ‘2’	
‘Q’	äußere untere rechte Ecke	wie Code ‘3’	•
‘R’	äußeres linkes T-Stück	wie Code ‘4’	
‘S’	(z.Z. undefiniert)		
‘T’	äußeres rechtes T-Stück	wie Code ‘6’	
‘U’	äußere obere linke Ecke	wie Code ‘7’	•
‘V’	äußeres oberes T-Stück	wie Code ‘8’	
‘W’	äußere obere rechte Ecke	wie Code ‘9’	•
‘X’	äußerer oberer Strich	wie Code ‘-’	•
‘x’	äußerer unterer Strich	wie Code ‘-’	•
‘Y’	äußerer linker Strich	wie Code ‘0’	•
‘y’	äußerer rechter Strich	wie Code ‘0’	•
‘Z’	(z.Z. undefiniert)		



Die Bedeutung sonstiger, oben nicht aufgeführter Zeichencodes größer-gleich ‘ ’ ist *für FIX* unerheblich. Codes kleiner ‘ ’ unterstützt der zweite Zeichensatz nicht.

FIX/Win, FIX/Web

FIX/Win und *FIX/Web* realisieren den Grafik-Zeichensatz durch Bitmaps und erlauben so die Definition beliebiger Zeichen.

42 Bildschirmverwaltung

1 Bildschirmaufbau

Wesensmerkmal von *FIX*-Programmen ist die Interaktion zwischen Benutzer und Programm, die in einer ständigen Abfolge zweier Schritte besteht:

- Das Programm benötigt Eingabe; es präsentiert dem Benutzer einen Bildschirminhalt und wartet auf eine Reaktion.
- Der Benutzer betätigt eine oder mehrere Tasten; auf diese Eingabe hin setzt das Programm seine Verarbeitung fort.

Um den Wechsel zwischen zwei aufeinander folgenden Bildschirminhalten möglichst effizient zu gestalten, verwendet *FIX* eine hochoptimierte Bildschirmverwaltung, deren Kernstück ein Bildschirmpuffer oder *virtueller Bildschirm* ist. *FIX*-Funktionen, die Ausgabe produzieren, verwenden keine Standard-E/A, sondern manipulieren lediglich den Bildschirmpuffer. Sie haben keine unmittelbare Wirkung am Terminal, da der Inhalt des Bildschirmpuffers dem Benutzer erst angezeigt wird, wenn das Programm auf Eingabe wartet oder dem Benutzer etwas mitteilen will.

Dadurch, dass *FIX* sich den zuletzt angezeigten Bildschirminhalt merkt (bzw. sicherstellt, dass seine Vorstellung davon, was auf dem Bildschirm zu sehen ist, mit der Realität übereinstimmt), lassen sich Änderungen des Bildschirminhalts stark optimieren. Wechsel der Cursor-Position oder des Videoattributs, die in der Regel längere Steuersequenzen erfordern, können minimiert werden, da das *logische* Schreiben von Zeichen (z.B. “zeichne einen Rahmen”) nicht mit der tatsächlichen Ausgabereihenfolge übereinzustimmen braucht und ein Zeichen nicht ausgegeben werden muss, wenn an der betreffenden Stelle des Bildschirms bereits das richtige Zeichen (mit dem richtigen Videoattribut) steht.

Die *FIX*-eigene Bildschirmverwaltung wird durch fxinit() gestartet und bleibt bis zum Aufruf von fxquit() aktiv. Einige Funktionen wie execute_cmd(), editor(), shell() unterbrechen sie jedoch kurzzeitig.

Die Funktionen von *FIX* schreiben in einen zweidimensionalen Puffer von der Größe des Bildschirms, den *virtual screen*. In einem zweiten Puffer - dem *current screen* - hält *FIX* fest, welche Zeichen sich zurzeit auf dem Bildschirm befinden. Wann immer es notwendig erscheint, dass der Anwender Änderungen am Bildschirm gewahr wird, vergleicht *FIX* die beiden Puffer miteinander und gibt diejenigen Positionen neu aus, wo sich Zeichen oder Videoattribut geändert haben. Der current screen wird dabei entsprechend aktualisiert.

Ein solcher “Neuaufbau” des Bildschirm geschieht automatisch, wenn

- Eingabe erwartet (oder von *FIX* simuliert) wird,
- die Abarbeitung eines Objekts oder die Erfassung eines Feldes beendet wird,
- aus einem Menü heraus eine Aktion gestartet wird, die eine längere Anlaufzeit haben könnte,
- dem Benutzer etwas mitgeteilt wird (Ausgabe einer Meldung, Maskenzustand).

Für den Entwickler stehen zum Neuaufbau des Bildschirms die Funktionen

```
void    sync_refresh(char *dbgmsg, BOOLEAN force)
void    refresh(void)
```

bereit. sync_refresh() gleicht üblicherweise den Terminal-Bildschirm an den virtuellen Bildschirm an[1], d.h. macht seit dem vorausgegangenen Aufruf erfolgte Ausgabe sichtbar (vgl. aber “Konfigurierung” unten). Current screen und virtual screen stimmen anschließend überein, bis erneut Ausgabe erfolgt. Der Aufruf refresh() entspricht sync_refresh("refresh", FALSE).

Durch Benutzung der E/A-Funktionen der C-Library wird die optimierte Ausgabe unterlaufen, da mittels printf() oder ähnlichen Funktionen durchgeführte Ausgabe die im current screen festgehaltene Vorstellung vom Bildschirminhalt entwertet. So veränderte Bildschirmbereiche bringt *FIX* erst wieder in Ordnung, wenn sich virtual screen und current screen dort unterscheiden. Mit der Funktion

```
void    touchscreen(void)
```

kann *FIX* allerdings angewiesen werden, beim nächsten Neuaufbau den Bildschirm vollständig auszugeben. Die Funktion sollte benutzt werden, wenn der Bildschirm unter Umgehung der Bildschirmverwaltung verändert wurde.

Konfigurierung

FIX will stets dann, wenn im virtuellen Bildschirm eine Änderung vorgenommen wurde, die dem Benutzer direkt vermittelt werden soll, eine Aktualisierung des Bildschirms ausführen. Es gibt allerdings viele Situationen, in denen mehrere Bildschirmaktualisierungen schnell aufeinander folgen und der Benutzer den Zwischenschritt meist gar nicht wahrnimmt:

Beispiel:

Der Benutzer steht auf einem Menüpunkt, dessen Aktion im Aufblenden einer Maske besteht, die sich über das Menü legt. Standardmäßig geht FIX wie folgt vor:

1. Der Menüpunkt - aktuell als “betreten” (INVERS) dargestellt - wird als “ausgelöst” (UNDERLINE) in den virtuellen Bildschirm ausgegeben.
2. Damit der Benutzer sieht, dass der Menüpunkt ausgelöst wurde und die Ausführung der Aktion beginnt, wird der Bildschirm mittels eines sync_refresh()-Aufrufs ausgegeben.
3. Die Maske wird geladen und in den virtuellen Bildschirm ausgegeben.
4. Das Startfeld der Maske wird betreten und *FIX* wartet auf einen Tastendruck; diesem Warten geht stets ein Aufruf zur Bildschirmaktualisierung voran, damit der Benutzer erkennen kann, was er bedient.

Bei einer geringen Bandbreite oder einer langsamen Verbindung zwischen dem Ausgabegerät / grafischem Frontend und dem Rechner, auf dem die *FIX*-Anwendung läuft, kann es wünschenswert sein, die Zahl der Bildschirmaktualisierungen zu verringern.

Ab Version 3.1.0 teilt *FIX* daher die Situationen, in denen eine Bildschirmaktualisierung sinnvoll erscheint, in zwei Gruppen ein:

1. solche, in denen die Bildschirmaktualisierung unverzichtbar scheint,
2. solche, in denen die Bildschirmaktualisierung nicht unbedingt erforderlich ist (sie werden als SYNC-Refreshs bezeichnet).

Dabei stuft *FIX* nur die folgenden Situationen in die erste Gruppe ein:

- das Warten auf Eingabe
- die Änderung des Zustandstextes im rechten oberen Window-Rahmen eines Objekts mittels set_obj_state():

  Danach erfolgt vielfach ein Datenaustausch mit der Datenbank, der längere Zeit in Anspruch nehmen kann (vgl. Seite 471, Seite 472, Seite 473).
- die Deaktivierung der *FIX*-Bildschirmverwaltung
- die Rückkehr aus der Verzweigung in eine Shell (→ shell())
- die Rückkehr aus der Verzweigung in einen Editor (→ editor())

1. Bei Bedienung mittels *FIX/Win* oder *FIX/Web* geschieht dies durch Senden entsprechender Anweisungen an das Frontend.

sowie

- das Anzeigen der Lizenzinformation bei beschränkten Lizenzen.

Entsprechend verwendet *FIX* intern ausschließlich die Funktion

```
void sync_refresh(char *dbgmsg, BOOLEAN force)
```

deren zweiter Parameter die Einstufung des Aufrufs kennzeichnet (TRUE bedeutet “unverzichtbar”). Der erste Parameter *dbgmsg* dient für einen kurzen Text, der die Aufrufsituation benennt[1].

Bei Aufrufen von sync_refresh() mit Argument FALSE kann die Bildschirmaktualisierung unterdrückt werden. Hierzu muss (1.) dieses Feature global durch Setzen der Ressource SkipSyncRefresh auf den Wert TRUE aktiviert worden sein und (2.) zum Zeitpunkt des Aufrufs die globale Variable

```
int S_skip_sync_refresh
```

(mit Vorbelegung 0) einen Wert ungleich 0 besitzen. Dieses Verfahren ermöglicht es, auch gezielt nur für bestimmte Abschnitte der Anwendung, SYNC-Refreshs unterbinden oder erzwingen.

2 Umsetzung von Zeichencodes bei der Ausgabe

Der Code, den ein Zeichen innerhalb seines Zeichensatzes einnimmt, muss nicht zwangsläufig mit dem übereinstimmen, der die Wiedergabe des Zeichens am Ausgabegerät bewirkt. Unter Umständen erfordert die Wiedergabe eines Zeichens aus dem Zeichensatz sogar das Senden einer Codesequenz.

FIX erlaubt daher, zu den Zeichen des Standard-Zeichensatzes - getrennt nach Geräten - Codesequenzen anzugeben, die zur Ausgabe des Zeichens am Bildschirm verwendet werden sollen. Mehr hierzu auf Seite 514.

3 Videoattribute

FIX unterstützt *intern* acht Videoattribute:

- normal (NORMAL)
- invers (INVERS)
- unterstrichen (UNDERLINE)
- halbhell (LOW)
- blinkend (BLINK)
- fett (BOLD)
- unsichtbar (NOTVISIBLE)
- grau (GRAYED)

Kombinationen dieser Attribute sind nicht möglich, wobei die mit der Version 3.0.0 eingeführten Attribute BOLD, NOTVISIBLE und GRAYED durch die Kombination des Wertes ALTERNATE und einem der anderen Videoattribute konstruiert werden (siehe `include/video.h`).

Die Realisierung der Videoattribute am Bildschirm erfolgt gemäß der Einträge in der Bildschirmbeschreibungsdatei (ein internes Attribut wie INVERS könnte also auch auf eine Farbe abgebildet werden). Unter Umständen kann ein Terminal nicht alle Attribute unterstützen.

1. Der Text wird lediglich benutzt, wenn auf Grund der Angabe des Schalters -dev Bildschirmaktualisierungen protokolliert werden.

FIX/Win, FIX/Web

FIX/Win und *FIX/Web* realisieren die Videoattribute i.d.R. durch Farben.

4 Basis-E/A

Die *FIX*-Library enthält eine Reihe von Funktionen zur Manipulation des Bildschirminhalts auf niedriger Ebene. Ihre Verwendung wird nur in Ausnahmefällen notwendig sein und sollte vermieden werden.

```
BOOLEAN   getpos(int *yadr, int *xadr)
void      move(int y, int x)
void      clrtoeol(void)
void      clrline(void)
void      clear(void)
void      standend(void)
void      standout(void)
void      underln(void)
void      lowint(void)
void      blink(void)
void      gr_start(void)
void      gr_end(void)
int       save_video(void)
int       restore_video(void)
void      putstr(unsigned char *str)
void      beep(void)
```

Höhe (ohne Statuszeile) und Breite des Bildschirms stellt *FIX* in den globalen Variablen

```
int  S_lines;
int  S_cols;
```

bereit. Die linke obere Ecke des Bildschirms hat die Koordinaten (0, 0) (Zeile/Spalte). Eine *Schreibmarke* bestimmt eine aktuelle Position auf dem virtuellen Bildschirm; beim Anzeigen des Bildschirminhalts (→ sync_refresh()) steht an dieser Stelle der Cursor.

getpos() hinterlässt die Position der Schreibmarke in **yadr* (Zeile) und **xadr* (Spalte).

move() setzt die Schreibmarke auf Zeile *y*, Spalte *x*.

clrtoeol() löscht die Zeichen ab der Position der Schreibmarke bis zum Ende der Zeile (“löschen” bedeutet “auf ' ' setzen”); die Position der Schreibmarke bleibt unverändert.

clrline() löscht alle Zeichen der Zeile, in der sich die Schreibmarke befindet; die Schreibmarke wird nach Spalte 0 bewegt.

clear() löscht den virtuellen Bildschirm; die Schreibmarke wird in die linke obere Ecke bewegt.

standend() schaltet die Videoattribute “invers”, “unterstrichen”, “halbhell” und “blinkend” aus.

standout() schaltet das Videoattribut “invers” ein.

underln() schaltet das Videoattribut “unterstrichen” ein.

lowint() schaltet das Videoattribut “halbhell” ein.

blink() schaltet das Videoattribut “blinkend” ein.

gr_start() schaltet in den zweiten Zeichensatz um.

gr_end() schaltet in den Standard-Zeichensatz um.

save_video() speichert Videoattribut und Zeichensatz in einem statischen Bereich, der bei jedem Aufruf überschrieben wird; zurückgegeben wird eine interne Kodierung für die gespeicherten Werte.

restore_video() setzt Videoattribut und Zeichensatz, die durch den letzten Aufruf von save_video() gespeichert wurden; zurückgegeben wird eine interne Kodierung für die neu gesetzten Werte.

putstr() schreibt einen String an die Stelle, an der sich die Schreibmarke befindet. Die Schreibmarke steht anschließend hinter dem letzten ausgegebenen Zeichen. Um innerhalb des Strings Videoattribut und Zeichensatz wechseln zu können, interpretiert putstr() einige nicht darstellbare Zeichen; zusätzlich zu den auf Seite 248 angegebenen sind dies

'\f'	Schreibmarke in die obere linke Ecke des Bildschirms bewegen
'\n'	Schreibmarke an den Anfang der nächsten Zeile bewegen

putstr() ignoriert die nicht darstellbaren Zeichen '\024' (DC4) und '\025' (NAK).

beep() erzeugt ein akustisches Signal.

5 Ausgabe unter Umgehung der Bildschirmverwaltung

Gelegentlich kann es in Anwendungen zweckmäßig sein, kurzlebige Ausgabe an der Bildschirmverwaltung vorbei zu produzieren. Dies kann einerseits mit E/A-Funktionen aus der C-Library erfolgen, doch bietet auch *FIX* hierfür Funktionen an. Zu gegebener Zeit muss allerdings durch Aufruf von touchscreen() dafür gesorgt werden, dass das tatsächliche Aussehen des Bildschirms wieder mit der Vorstellung, die *FIX* davon hat, in Übereinstimmung gebracht wird.

```
void    tty_move(int y, int x)
void    tty_clrtoeol(void)
void    tty_clear(void)
void    writetty(int y, int x, char *str, int clflg)
```

tty_move() bewegt den Cursor in Spalte *y*, Zeile *x*.

tty_clrtoeol() löscht die Zeile hinter der Position des Cursors.

tty_clear() löscht den Bildschirm und bewegt den Cursor in die linke obere Ecke.

writetty() schreibt den String *str* in Spalte *y*, Zeile *x*. Ist *clflg* von 0 verschieden, wird zuvor die Zeile gelöscht. Anschließend wird der Cursor an die Position bewegt, an der *FIX* ihn zuvor vermutete, und touchscreen() aufgerufen.

Eine Zeichencode-Umsetzung (vgl. Seite 275) oder ein Videoattribut-Wechsel (vgl. Seite 248) erfolgen hier nicht.

43 Funktionen der *FIX*-Library

Dieses Kapitel führt alle dem Entwickler angebotenen Funktionen der *FIX*-Library auf, geordnet nach Sachgebieten:

- Ablaufumgebung
- Datenbank
- Events
- Objekt
- Window-Verwaltung
- Menü
- Feld
- Maske
- Einzelsatz-Maske
- Mehrsatz-Maske
- Tabellenmaske
- Selo
- Choice
- Memory Relation
- Hilfetext
- Layout
- Paintarea
- Meldungen
- Fehlerbehandlung
- E/A
- Frontend
- dec_t-Funktionen
- date_t-Funktionen [1]
- dtime_t- und intrvl_t-Funktionen
- Wahrheitswert-Funktionen
- Konversion
- Vermischtes
- Grundfunktionen

Mit † gekennzeichnete Funktionen existieren zur Wahrung der Kompatibilität mit früheren *FIX*-Versionen und sollten bei der Neuerstellung und Pflege von Anwendungen nicht mehr benutzt werden.

1. date_t ist ein Makro aus `fix/fixdate.h`, das der besseren Lesbarkeit wegen anstelle von long verwendet wird, wenn dieser C-Typ zur Speicherung eines Datums benutzt wird.

Hier nicht dokumentierte Funktionen der Library können bei einem der nächsten Versionswechsel ohne Ankündigung entfallen. Gleiches gilt für dokumentierte Routinen, deren Funktionalität im Zuge der Standardisierung von Routinen der C-Library abgedeckt wird.

1 Ablaufumgebung

addfct — C-Funktion bekannt machen

Definition:

BOOLEAN addfct(name, fct)
char *name; int (*fct)();

Beschreibung:

ordnet dem Namen *name* die Funktion *fct* zu, falls *name* noch keine Funktion bezeichnet, und nimmt *name* in die Liste der Funktionen auf, die z.B. in Menüs (call-Aktion) oder für die Feldprüfung (Prüfroutine) zur Verfügung stehen. Der Typ der Funktion *fct* ist beliebig.

Ergebnis:

TRUE - Erfolg
FALSE - Fehler (*name* schon verwendet, Speichermangel etc.)

Siehe auch:

getfct()

arginspect — Kommandozeile interpretieren

Definition:

void arginspect(argcp, argv)
int *argcp; char *argv[];

Beschreibung:

interpretiert die standardmäßig von *FIX* behandelten Programmschalter (zB. -test <n>, -DM usw.) und eliminiert sie aus *argv*[]; **argcp* wird entsprechend vermindert.

Siehe auch:

set_program_name()

fastexit — Programmabbruch erzwingen

Definition:

void fastexit(status, txt)
int status; char *txt;

Beschreibung:

beendet die *FIX*-eigene Bildschirmverwaltung, setzt - bei Terminalbetrieb - die Terminaleinstellung wieder in den Zustand, in der sie beim Start des *FIX*-Programms war, schreibt die Meldung *txt* (falls *txt* == (char *)0, einen Standardtext) nach stderr und beendet das Programm mit dem Ergebnis *status*.

Siehe auch:

fxquit()

Hinweis:

Bei Benutzung der Funktion sollte man die Auswirkungen auf die Datenbank in Betracht ziehen.

Wird in Source ausgeliefert (`$FXDIR/src`).

fxinit — *FIX*-Anwendungsprogramm initialisieren

Definition:

void fxinit(argc, argv)
int argc; char **argv;

Beschreibung:

liest Umgebungsvariablen und Ressourcen, bestimmt Meldungsdateien, initialisiert globale Variablen, liest - bei Terminalbetrieb - Bildschirm- und Tastaturbeschreibung, aktiviert die Bildschirmverwaltung und Signalbehandlung. Der Aufruf von fxinit() ist Voraussetzung für die Benutzung nahezu aller anderen Funktionen der *FIX*-Library.

Siehe auch:

fxquit(), arginspect()

Hinweis:

Die (zur Wahrung der Kompatibilität beibehaltenen) Parameter *argc* und *argv* werden nicht ausgewertet.

Bei einem schwerwiegenden Fehler erfolgt ein Programmabbruch durch fastexit().

fxquit — *FIX*-Anwendung beenden

Definition:

void fxquit()

Beschreibung:

beendet die *FIX*-eigene Bildschirmverwaltung und setzt - bei Terminalbetrieb - die Terminaleinstellung wieder in den Zustand, in der sie beim Start des *FIX*-Programms war, ohne jedoch das Programm zu beenden.

Siehe auch:

Makro fxexit(), fastexit()

Hinweis:

Diese Funktion sollte jedem Aufruf von exit() vorausgehen.

get_program_name — Programmname ermitteln

Definition:

char *get_program_name()

Beschreibung:

ermittelt den mittels set_program_name() gespeicherten Namen.

Ergebnis:

Zeiger auf statischen Bereich, der den Namen enthält, bei Erfolg
(char *)0, wenn kein Aufruf von set_program_name() erfolgt ist

Siehe auch:

set_program_name()

getfct — Zeiger auf bekannt gemachte C-Funktion ermitteln

Definition:

int (∗getfct(name))()
char ∗name;

Beschreibung:

liefert die dem Namen *name* mittels addfct() zugeordnete Funktion.

Ergebnis:

Zeiger auf eine C-Funktion
(int (∗)())0 bei Fehler

Siehe auch:

addfct()

Hinweis:

Der Typ der Funktion ist ohne Bedeutung, ggf. muss ein Cast erfolgen.

set_program_name — Programmname speichern

Definition:

BOOLEAN set_program_name(path)
char ∗path;

Beschreibung:

speichert basename(*path*), d.h. den letzten Bestandteil des Pfades *path*, in einem statischen Bereich.

Ergebnis:

TRUE - Erfolg
FALSE - Fehler (Überlauf)

Siehe auch:

get_program_name()

Hinweis:

Diese Funktion wird von arginspect() mit dem Argument argv[0] aufgerufen.

2 Datenbank

fx_begin_transaction — virtuelle Transaktion beginnen

Definition:

int fx_begin_transaction()

Beschreibung:

beginnt eine neue virtuelle Transaktion, indem die Transaktionsstufe um 1 erhöht wird.

Ergebnis:

1 - Stufe um 1 erhöht
0 - Transaktionssicherung nicht verfügbar
-1 - Fehler (keine aktuelle Transaktion)

Siehe auch:

fx_commit_transaction(), fx_rollback_transaction()

fx_commit_transaction — virtuelle Transaktion beenden

Definition:

int fx_commit_transaction()

Beschreibung:

schließt eine virtuelle Transaktion ab, indem die Transaktionsstufe um 1 vermindert wird (sofern nicht bereits 0).

Auf Transaktionsstufe 0 wird “commit work” bzw., erfolgte auf einer höheren Stufe ein Aufruf von fx_rollback_transaction(), stattdessen “rollback work” ausgeführt und anschließend eine neue Datenbanktransaktion begonnen.

Ergebnis:

1 - Stufe um 1 vermindert
0 - Datenbanktransaktion beendet und neue begonnen oder Transaktionssicherung nicht verfügbar
-1 - Fehler (siehe SQLCODE)

Siehe auch:

fx_begin_transaction(), fx_rollback_transaction()

fx_connect_cnt — Datenbanksitzungszähler abfragen

Definition:

long fx_connect_cnt()

Beschreibung:

bestimmt die Anzahl der bislang erfolgreichen Aufrufe von fx_database_connect().

Ergebnis:

Anzahl der erfolgreichen Aufrufe von fx_database_connect()

Siehe auch:

fx_database_connect()

fx_database_connect — Datenbankverbindung aufbauen

Definition:

long fx_database_connect(dbname)
char *dbname;

Beschreibung:

stellt eine Verbindung zur Datenbank *dbname* über ESQL/C und ggf. über C-ISAM her. Ist *dbname* == (char *)0, wird der Wert der Umgebungsvariablen DBNAME als Datenbankname genommen.

Bei einer Datenbank mit Transaktionssicherung wird anschließend eine Datenbanktransaktion begonnen, die Transaktionsstufe auf 0 gesetzt.

Ergebnis:

0 - Erfolg
1 - zum Zeitpunkt des Aufrufs besteht bereits eine Datenbankverbindung
2 - kein Datenbankname verfügbar
3 - Fehler beim Öffnen über C-ISAM
< 0 - ESQL/C-Fehler bei ‘database’-Anweisung

Siehe auch:

fx_database_disconnect(), fx_connect_cnt()

Hinweis:

Wird im Programm mit *FIX*-Funktionen auf die Datenbank zugegriffen, muss diese mit fx_database_connect() geöffnet werden.

Besteht bereits eine Datenbankverbindung (dbconnected == TRUE), so ist die Funktion wirkungslos, dh. die vorhandene Datenbankverbindung bleibt bestehen.

fx_database_disconnect — Datenbankverbindung abbauen

Definition:

long fx_database_disconnect()

Beschreibung:

beendet die mit fx_database_connect() hergestellte Datenbankverbindung.

Bei einer Datenbank mit Transaktionssicherung erfolgt zuvor ein Rollback der aktuellen Datenbanktransaktion, die Transaktionsstufe wird undefiniert (< 0).

Ergebnis:

0 - Erfolg
1 - zum Zeitpunkt des Aufrufs besteht keine Datenbankverbindung
3 - Fehler beim Schließen über C-ISAM
< 0 - ESQL/C-Fehler bei ‘close database’-Anweisung

Siehe auch:

fx_database_connect()

Hinweis:

Besteht keine Datenbankverbindung (dbconnected == FALSE), so ist die Funktion wirkungslos.

fx_io - universelle Funktion zur Datenmanipulation

Definition:

int fx_io(mode, method, objp, sql_fct, copy_fct, usr_data)
int mode; int method; obj *objp;
int (*sql_fct)(/* int */); void (*copy_fct)(/* int */); char *usr_data;

Beschreibung:

dient in *FIX*-Programmen zur Manipulation der Daten. Die Methode *method* und die Art des Arguments *objp* bestimmen, in welcher Weise die Funktion *sql_fct* zur Durchführung der Zugriffsart *mode* herangezogen wird. Die Parameter können folgende Werte haben:

mode - Art des Zugriffs:

FX_READ - Daten lesen
FX_INSERT - Daten einfügen

FX_UPDATE - Daten ändern
FX_DELETE - Daten löschen

method - Methode, nach der der Zugriff vorgenommen werden soll:

STANDARD - Standard-Verfahrensweise
METH_LOCKING_READ_CURSOR - lesen mit sperrendem Cursor (STANDARD)
METH_SINGLE_STATEMENT - Operation mit einzelnen ESQL/C-Statements ausführen
METH_UNIQUE_KEY - spezielle Logik im Update-Fall, die verhindert, dass Sätze doppelt auftreten
METH_DELETE - statt eines bzw. mehrerer Update werden alle Sätze gelöscht und dann neu eingefügt

objp - Zeiger auf eine Einzelsatz-Maske oder Mehrsatz-Maske.

sql_fct - Zeiger auf eine Funktion, die mittels ESQL/C die Datenbankoperationen ausführt. Die Funktion muss alle für die Zugriffsarten *mode* und Methode *method* benötigten Operationen (vgl. `fix/fx_io.h`) realisieren.

copy_fct - Zeiger auf die Funktion, die die Daten zwischen Datenbank- und Feld-Hostvariablen überträgt. Ist kein Transport notwendig, muss der Parameter mit dem Makro NO_COPY belegt werden.

usr_data - Zeiger auf Daten, die eine von STANDARD verschiedene Methode *method* zusätzlich benötigt (zurzeit verwendet nur METH_UNIQUE_KEY diesen Parameter: er enthält dann ein mit 0 abgeschlossenes integer-Array, das die Spaltennummern in der Memory Relation ($1 \le i \le$ #Spalten) zu den Feldern enthält, die den Schlüssel bilden). Ansonsten sollte *usr_data* mit dem Makro NO_USR_DATA belegt werden.

Ergebnis:

0 (SUCCESS) - Erfolg
!= 0 - Fehler (z.B. SQLNOTFOUND)

Siehe auch:

m_fnr_to_colno()

fx_rollback_requested — Transaktionszustand abfragen

Definition:

BOOLEAN fx_rollback_requested()

Beschreibung:

erlaubt es abzufragen, ob seit Beginn der letzten Datenbanktransaktion fx_rollback_transaction() erfolgreich aufgerufen wurde, die aktuelle Datenbanktransaktion also bei Erreichen der Transaktionsstufe 0 zurückgefahren wird.

Ergebnis:

TRUE - erfolgreicher Aufruf von fx_rollback_transaction() erfolgt
FALSE - kein (erfolgreicher) Aufruf von fx_rollback_transaction() erfolgt

Siehe auch:

fx_rollback_transaction()

fx_rollback_transaction — virtuelle Transaktion zurücksetzen

Definition:

int fx_rollback_transaction()

Beschreibung:

schließt eine virtuelle Transaktion ab, indem die Transaktionsstufe um 1 vermindert wird (sofern nicht bereits 0).

Auf Transaktionsstufe 0 wird “rollback work” ausgeführt und anschließend eine neue Datenbanktransaktion begonnen; anderenfalls merkt *FIX* sich, dass ein Rollback gefordert worden ist, und führt kein “commit work” durch, bis “rollback work” erfolgt ist.

Ergebnis:

1 - Stufe um 1 vermindert
0 - Datenbanktransaktion beendet und neue begonnen oder Transaktionssicherung nicht verfügbar
-1 - Fehler (siehe SQLCODE)

Siehe auch:

fx_begin_transaction(), fx_commit_transaction(), fx_rollback_requested()

fx_sql_test — SQL-Fehler melden und Benutzerreaktion erwarten

Definition:

```
void fx_sql_test(txt)
char *txt;
```

Beschreibung:

prüft den Wert von SQLCODE und meldet, falls SQLCODE < 0 ist, mittels sql_error() den Fehler. *txt* kann dazu benutzt werden, den Fehlerkontext (Source, Art der ESQL/C-Anweisung) zu beschreiben, darf aber auch (char *)0 sein.

Siehe auch:

sql_error()

need_files — Anzahl gleichzeitig offener Relationen beschränken

Definition:

```
void need_files(anz)
int anz;
```

Beschreibung:

setzt die Maximalanzahl der von der C-ISAM-Schnittstelle gleichzeitig offen zu haltenden Relationen auf (Maximalanzahl offener Dateien pro Prozess - *anz*) / 2. *anz* sollte eine Obergrenze für die gleichzeitig offenen sonstigen Dateien der Anwendung sein.

Hinweis:

Diese Funktion existiert nur bei *FIX*-Versionen mit C-ISAM-Unterstützung.

Für *anz* > (Maximalanzahl offener Dateien pro Prozess - 4) erfolgt ein Programmabbruch durch fastexit().

3 Events

dcgetch — Event erwarten

Definition:

```
Event dcgetch()
```

Beschreibung:

holt das nächste Event aus dem Input-Puffer, d.h. wartet ggf. auf die nächste Taste.

Ergebnis:

Event
NOTHING, wenn Eingabe nicht erkannt

Siehe auch:

undcgetch(), undcgetstr(), Variable S_ReadEvent

Hinweis:

Vor der Tastaturabfrage oder, bei Einsatz eines Frontends, einer entsprechenden Anforderung an das Frontend wird ein Puffer überprüft: solange dieser Events oder Zeichen enthält, die mittels undcgetch() oder undcgetstr() zurückgestellt wurden, werden zuerst sie zurückgegeben.

fxReadEvent — Event vom Eingabestrom lesen

Definition:

Event fxReadEvent()

Beschreibung:

liest die Codesequenz für ein Event vom Filedeskriptor 0.

Ergebnis:

Event
K_IG - Lesen unterbrochen (z.B. durch Eintreffen eines Signals)
NOTHING - Fehler beim Lesen oder kein Event im Eingabestrom erkannt

Siehe auch:

dcgetch(), Variable S_ReadEvent

Hinweis:

Die Art der Eventerkennung differiert stark je nachdem, ob die Eingabe per Tastatur oder von einem Frontend erfolgt.

Das Eintreffen des Signals SIGINT führt unter gewissen Voraussetzungen zu einer Rückkehr mit dem INTR zugeordneten Event.

K_IG wird von dcgetch() intern behandelt und nicht als Rückgabewert weitergereicht.

get_lastinput — zuletzt von dcgetch() geliefertes Event abfragen

Definition:

Event get_lastinput()

Beschreibung:

gibt das zuletzt von dcgetch() gelieferte Event zurück. Es gibt allerdings Events, die dcgetch() zwar liest und abliefert, aber nicht zum Auslesen durch diese Funktion hinterlegt.

Ergebnis:

Event

Siehe auch:

dcgetch(), undcgetch()

Hinweis:

Die Funktion kann nützlich sein, wenn bei der Eventbehandlung unterschieden werden soll, ob ein Event aus einer (tatsächlichen oder zurückgestellten) Eingabe resultiert oder von *FIX* generiert wurde.

undcgetch — Event in den Input-Puffer zurückstellen

Definition:

Event undcgetch(c)
Event c;

Beschreibung:

stellt das Event *c* in den Input-Puffer zurück (ähnlich der C-Funktion ungetc()). Das zuletzt zurückgestellte Event wird wieder als erstes von dcgetch() hervorgeholt.

Ergebnis:

c bei Erfolg
NOTHING bei Fehler

Siehe auch:

dcgetch(), undcgetstr()

Hinweis:

Die Zurückstellung geschieht in einem 80 Events fassenden Stack.

undcgetstr — Zeichenfolge in den Input-Puffer zurückstellen

Definition:

int undcgetstr(str)
char *str;

Beschreibung:

stellt die Zeichen der Zeichenfolge *str* in den Input-Puffer zurück. Das erste Zeichen von *str* wird wieder als erstes von dcgetch() hervorgeholt.

Ergebnis:

0 bei Erfolg
Anzahl der Zeichen, die *nicht* zurückgestellt werden konnten, bei Fehler

Siehe auch:

dcgetch(), undcgetch()

Hinweis:

Die Zurückstellung geschieht in einem 80 Event fassenden Stack.

4 Objekt

is_an_obj — Plausibilität eines Objektzeigers prüfen †

Definition:

int is_an_obj(objp)
obj *objp;

Beschreibung:

überprüft, ob *objp* auf ein Objekt zeigt.

Ergebnis:

objp->ob_class, wenn ja
0 bei Fehler

Hinweis:

Ältere *FIX*-Versionen verwendeten diese Funktion in generierten Sourcen.

global_event_control — globale Eventbehandlung

Definition:

Event global_event_control(objp, ev_ctrl, ev)
obj *objp; Event (*ev_ctrl)(/* obj *, Event */); Event ev;

Beschreibung:

erlaubt eine zentrale, d.h. für ein gesamtes Programm gültige Behandlung von Events. Diese Funktion, die in einem eigenen Modul der *FIX*-Library enthalten ist, wird u.a. immer dann aufgerufen, wenn laut Beschreibung im Handbuch ein Event der Anwendungslogik für Menüs und Masken angeboten wird. Die Funktion wird beim Aufruf mit dem Objekt, der für das Objekt gültigen Anwendungslogik und dem Event versorgt.

Ergebnis:

(**ev_ctrl*)(*objp*, *ev*), falls *ev_ctrl* != (Event (*)(/* obj *, Event */))0
ev sonst

Siehe auch:

perform()

Hinweis:

Da die Funktion ein eigenes Modul bildet, kann ein Anwendungsprogramm eine eigene Routine dieses Namens definieren. Diese muss, sofern sie kein spezielles Vorgehen wünscht, ein Ergebnis liefern, das dem obigen entspricht.

o_GetClickSensitivity — abfragen, in welcher Form ein Objekt Maustasten-Events unterstützt

Definition:

int o_GetClickSensitivity(objp)
obj *objp;

Beschreibung:

teilt mit, in welcher Form die Standardlogik von *FIX* Maustasten-Events für das Objekt *objp* behandeln würde.

Ergebnis:

CLICK_EQUIV_TO_KEY - Event wird wie eine Sondertaste behandelt
CLICK_INSENSITIVE - Event wird von *FIX* ignoriert (No-Op)
CLICK_SENSITIVE - Event dient zur Positionierung
-2 - Objekttyp nicht unterstützt

Hinweis:

Das Ergebnis der Funktion ist unabhängig davon, ob das Programm durch ein Frontend bedient wird oder nicht.

o_RetrieveButtonClickedOn — abfragen, auf welchen Feldbutton geklickt wurde

Definition:

int o_RetrieveButtonClickedOn(objp, buttonidx_adr)
obj *objp; int *buttonidx_adr;

Beschreibung:

erlaubt bei der Behandlung von Maustasten-Events in der Anwendungslogik eines Objekts, den Index des Feldbuttons abzufragen, über dem sich der Mauszeiger befand, als geklickt wurde.

Ergebnis:

0 - Erfolg
-1 - *objp* nicht das aktuelle Objekt
-2 - Objekttyp nicht unterstützt
-3 - Objekt nicht klick-sensitiv
-4 - sonstiger Fehler (z.B. kein Maustasten-Event vorausgegangen)

Hinweis:

Gegenwärtig wird diese Funktionalität nur bei Masken unterstützt.

o_RetrieveItemClickedOn — abfragen, worauf geklickt wurde

Definition:

int o_RetrieveItemClickedOn(objp, fnr_adr, rowoffset_adr)
obj *objp; int *fnr_adr; int *rowoffset_adr;

Beschreibung:

erlaubt bei der Behandlung von Maustasten-Events in der Anwendungslogik eines Objekts, das Objektelement abzufragen, über dem sich der Mauszeiger befand, als geklickt wurde. Bei Masken ist dies zwangsläufig ein zum Zeitpunkt des Klickens sichtbares Feld. Die entsprechende Element-Position wird in **fnr_adr* abgelegt. Handelt es sich bei *objp* um eine Tabellenmaske, wird **rowoffset_adr* der Satz-Abstand zwischen angeklickten und aktuellen Satz zugewiesen (z.B. -1 beim Klicken auf den vorausgehenden Satz).

Ergebnis:

0 - Erfolg
-1 - *objp* nicht das aktuelle Objekt
-2 - Objekttyp nicht unterstützt
-3 - Objekt nicht klick-sensitiv
-4 - sonstiger Fehler (z.B. kein Maustasten-Event vorausgegangen)

Hinweis:

Gegenwärtig wird diese Funktionalität nur bei Masken unterstützt.

o_RetrievePositionClickedOn — abfragen, wohin geklickt wurde

Definition:

int o_RetrievePositionClickedOn(objp, y_adr, x_adr)
obj *objp; int *y_adr; int *x_adr;

Beschreibung:

erlaubt bei der Behandlung von Maustasten-Events in der Anwendungslogik eines Objekts, die Bildschirmposition abzufragen, an der sich der Mauszeiger befand, als geklickt wurde. Die entsprechende Zeilen- und Spaltenwerte werden in **y_adr* und **x_adr* hinterlegt.

Ergebnis:

0 - Erfolg
-1 - *objp* nicht das aktuelle Objekt
-2 - Objekttyp nicht unterstützt
-3 - Objekt nicht klick-sensitiv
-4 - sonstiger Fehler (z.B. kein Maustasten-Event vorausgegangen)

Hinweis:

Gegenwärtig wird diese Funktionalität nur bei Masken unterstützt.

o_free — Objekt freigeben

Definition:

void o_free(objp)
obj *objp;

Beschreibung:

gibt das Objekt *objp* und die darin eingebetteten Objekte frei; dies darf nur geschehen, wenn keine Referenzen von außen auf die freizugebenden Objekte mehr existieren (z.B. Links auf Felder, Referenzen im Selo etc).

Hinweis:

Der von den freigegebenen Objekten benutzte Speicherplatz wird wieder verfügbar gemacht.

Bei Masken gibt o_free() auch die Memory Relations der Maske selbst und/oder eingebetteter Mehrsatz-Masken frei.

o_get_props — anwendungsspezifische Information zum Objekt abfragen

Definition:

PrivatePropertyType o_get_props(objp)
obj *objp;

Beschreibung:

liefert den initialen (aus der Beschreibungsdatei übernommenen) bzw. den mittels o_set_props() zum Objekt *objp* gespeicherten (anwendungsspezifischen) Wert.

Ergebnis:

augenblicklicher Wert

Siehe auch:

o_set_props()

Hinweis:

Die Struktur PrivatePropertyType ist in `fix/props.h` definiert.

o_install_ev_control — objekteigene Anwendungslogik installieren

Definition:

Event (*o_install_ev_control(objp, fct))(/* obj *, Event */)
obj *objp; Event (*fct)(/* obj *, Event */);

Beschreibung:

ersetzt die vorhandene objekteigene Anwendungslogik des Objekts *objp* durch die Funktion *fct*. Falls *fct* == (Event(*)(/* obj *, Event */))0, wird die vorhandene Anwendungslogik beseitigt.

Ergebnis:

Zeiger auf die zuvor als Anwendungslogik installierte Funktion

Siehe auch:

perform(), m_present()

Hinweis:

Während der Bearbeitung eines Objekts kann die Anwendungslogik zwar ausgetauscht werden, dies wirkt sich aber erst bei einer erneuten Bearbeitung des Objekts aus.

Zur Zeit wird eine objekteigene Anwendungslogik nur bei Menüs und Masken unterstützt.

Die objekteigene Anwendungslogik wird an die Bearbeitung eingebetteter Masken weitergegeben.

o_move — Objekt-Window verschieben

Definition:

void o_move(objp, nz, ns)
obj *objp; int nz; int ns;

Beschreibung:

verschiebt die linke obere Ecke des Objekts *objp* an die Position (*nz*, *ns*). Wenn das Objekt am Bildschirm dargestellt ist, wird das Window verschoben, ohne dass sich die Schichtung der Windows hierdurch ändert.

Siehe auch:

rebuild()

Hinweis:

Keine Plausibilitätsprüfungen bzgl. der Koordinaten.

o_set_props — anwendungsspezifische Information zum Objekt hinterlegen

Definition:

void o_set_props(objp, value)
obj *objp; PrivatePropertyType value;

Beschreibung:

speichert zum Objekt *objp* den (anwendungsspezifischen) Wert *value*.

Siehe auch:

o_get_props()

Hinweis:

Die Struktur PrivatePropertyType ist in `fix/props.h` definiert.

perform — Objekt abarbeiten

Definition:

Event perform(objp, usr_control)
obj *objp; Event (*usr_control)(/* obj *, Event */);

Beschreibung:

ist die zentrale Routine zur Bearbeitung geladener Objekte. Ein Anwendungsprogramm enthält typischerweise mindestens einen Aufruf dieser Funktion. perform() kann auf Menüs, Masken, Selos und Choices angewendet werden. Nach einer erfolgreichen Plausibilitätsprüfung erscheint das Objekt als oberstes Window auf dem Bildschirm (o_appear()) und in der Statuszeile wird der dem Objekt zugeordnete Prompttext angezeigt. Beim Abschluss von perform() verschwindet das Objekt vom Bildschirm (o_disappear()), sofern es nicht die Eigenschaft STEADY oder (STEADY |ABOVE) hat, und der beim Eintritt in die Routine gültige Prompttext wird restauriert. Die globale Variable c_objp zeigt auf das (bei rekursivem Aufruf, innerste) Objekt, für das perform() aufgerufen wurde.

Ergebnis:

das Event, aufgrund dessen das Objekt verlassen wurde
L_STAY bei Fehler

Siehe auch:

m_present(), m_mode()

Hinweis:

Die mitgegebene Funktion *usr_control* wird nur benutzt, wenn das Objekt *objp* keine eigene Logik besitzt, d.h. *objp*->ev_control == (Event (*)(/* obj *, Event */))0.

Dasselbe Objekt kann nicht rekursiv ausgeführt werden.

Während ein Objekt in Bearbeitung ist, ist in der Komponente ob_state das Bit ACTIVE gesetzt.

restart — Objekt in Grundzustand versetzen

Definition:

```
void restart(objp)
obj *objp;
```

Beschreibung:

versetzt das Objekt *objp* - bei Masken auch die eingebetteten Masken (bottom up) - in den Grundzustand.

Bei einer Maske werden die bisherigen Feldinhalte gerettet, so dass sie für die Taste f2 (L_CPYFIELD) zur Verfügung stehen, und alle Nicht-Feld-Links auf ihren Default-Wert gesetzt; die Feldeigenschaften werden restauriert, wie sie in der Beschreibungsdatei definiert sind. Der Maskeninhalt wird anschließend neu dargestellt.

Bei einer Choice werden die Markierungen der Tupel gelöscht und, sofern vorhanden, auf das erste Tupel positioniert.

Bei Menüs und Selos hat die Funktion keine Bedeutung.

Hinweis:

Bei Mehrsatz-Masken ist diese Funktion nur wirksam, wenn die zugehörige Memory Relation keine Sätze enthält.

5 Window-Verwaltung

appear — Objekt-Window öffnen und/oder in den Vordergrund bringen

Definition:

```
void appear(objp)
obj *objp;
```

Beschreibung:

lässt das Objekt *objp* am Bildschirm im Vordergrund, d.h. zuoberst, erscheinen. Sind in *objp* Objekte mit der Eigenschaft STEADY bzw. (STEADY|ABOVE) eingebettet, erscheinen auch diese am Bildschirm, wobei Erstere unter dem Objekt *objp*, Letztere über *objp* zu liegen kommen. Ist das Objekt *objp* bereits am Bildschirm dargestellt, wirkt appear() wie foreground().

Siehe auch:

o_appear()

Hinweis:

Statt appear() sollte o_appear() verwendet werden.

disappear —Objekt-Window schließen

Definition:

void disappear(objp, really)
obj *objp; int really;

Beschreibung:

bringt das Objekt *objp* und alle darin eingebetteten Objekte zum Verschwinden.

Siehe auch:

o_disappear()

Hinweis:

Statt disappear() sollte o_disappear() verwendet werden.

Der (zur Wahrung der Kompatibilität beibehaltene) Parameter *really* wird nicht ausgewertet.

fitwindow — Positionierung von Selo's und Hilfetexten

Definition:

void fitwindow(f, hp_box)
field *f; box_cb *hp_box;

Beschreibung:

Standard-Positionierungsfunktion: Die Standard-Positionierung versucht die Überdeckung des Feldes, an dem das Selo oder der Hilfetext registriert ist, zu vermeiden. Dabei wird ein Selo so weit wie möglich vom Feld weg positioniert.

Siehe auch:

fitwindow_new()

Hinweis:

Die Funktion ist standardmäßig an der globalen Variablen S_fitwindow hinterlegt.

fitwindow_new — Positionierung von Selo's und Hilfetexten

Definition:

void fitwindow_new(f, hp_box)
field *f; box_cb *hp_box;

Beschreibung:

Verbesserte Positionierungsfunktion: Diese Positionierung versucht, die Überdeckung des Feldes, an dem das Selo oder der Hilfetext registriert ist, zu vermeiden. Dabei wird versucht, das Selo möglichst dicht an das Feld, am besten unterhalb des Feldes am Anfang beginnend zu positionieren.

Die Verwendung dieser Funktion wird vor allem bei Bildschirmgrößen über 25x80 empfohlen, da hierbei das Verhalten der Standard-Funktion oft nicht das erwartete Ergebnis zeigt.

Siehe auch:

fitwindow()

Hinweis:

Die Funktion kann an der globalen Variablen S_fitwindow hinterlegt werden.

foreground — Objekt-Window in den Vordergrund bringen

Definition:

int foreground(objp)
obj *objp;

Beschreibung:

bringt das am Bildschirm dargestellte Objekt *objp* in den Vordergrund. Sind in *objp* Objekte mit der Eigenschaft (STEADY|ABOVE) eingebettet, kommen diese über dem Objekt zu liegen.

Ergebnis:

0 bei Erfolg
-1, wenn *objp* nicht am Bildschirm dargestellt ist

Siehe auch:

o_appear(), is_foreground()

is_foreground — Vorhandensein eines Objekt-Windows prüfen

Definition:

BOOLEAN is_foreground(objp)
obj *objp;

Beschreibung:

testet, ob sich das Objekt *objp* am Bildschirm im Vordergrund befindet.

Ergebnis:

TRUE, wenn *objp* oberstes Objekt am Bildschirm ist oder über ihm nur eingebettete Objekte mit der Eigenschaft (STEADY|ABOVE) liegen
FALSE sonst

Siehe auch:

o_appear(), o_disappear(), foreground(), is_present()

is_present — Lage eines Objekt-Windows prüfen

Definition:

BOOLEAN is_present(objp)
obj *objp;

Beschreibung:

testet, ob das Objekt *objp* am Bildschirm dargestellt ist.

Ergebnis:

TRUE, wenn *objp* dargestellt
FALSE sonst

Siehe auch:

o_appear(), o_disappear()

Hinweis:

is_present(), angewendet auf ein verdecktes Objekt, liefert TRUE.

new_background — Hintergrund-Window öffnen

Definition:

obj *new_background()

Beschreibung:

verdeckt die sichtbaren Objekte durch ein weißes Window, das den ganzen Bildschirm einnimmt (Hintergrund).

Ergebnis:

ein Identifikator (Zeiger auf Pseudo-Objekt), der benötigt wird, wenn der Hintergrund wieder zum Verschwinden gebracht werden soll

Siehe auch:

rem_background()

o_appear - Objekt-Window öffnen und/oder in den Vordergrund bringen

Definition:

void o_appear(objp)
obj *objp;

Beschreibung:

lässt das Objekt *objp* am Bildschirm im Vordergrund, d.h. zuoberst, erscheinen. Sind in *objp* Objekte mit der Eigenschaft STEADY bzw. (STEADY|ABOVE) eingebettet, erscheinen auch diese am Bildschirm, wobei Erstere unter dem Objekt *objp*, Letztere über *objp* zu liegen kommen. Ist das Objekt *objp* bereits am Bildschirm dargestellt, wirkt o_appear() wie foreground().

Siehe auch:

o_disappear(), is_present(), foreground(), is_foreground()

o_disappear — Objekt-Window schließen

Definition:

void o_disappear(objp)
obj *objp;

Beschreibung:

bringt das Objekt *objp* und alle darin eingebetteten Objekte zum Verschwinden.

Siehe auch:

o_appear(), is_present()

o_rebuild — alle sichtbaren Windows oberhalb eines Objekt-Windows neu ausgeben

Definition:

void o_rebuild(objp)
obj *objp;

Beschreibung:

baut, wenn das Objekt *objp* sichtbar (d.h. am Bildschirm dargestellt und über dem zuletzt eingefügten Hintergrund liegend) ist, den von ihm beanspruchten Bildschirmbereich neu auf.

Siehe auch:

rebuild(), new_background()

rebuild — alle sichtbaren Windows neu ausgeben

Definition:

```
void rebuild(do_clear)
BOOLEAN do_clear;
```

Beschreibung:

baut die Windows aller sichtbaren Objekte neu auf dem Bildschirm auf. Wenn *do_clear* != FALSE ist, wird zuvor der Bildschirm gelöscht und der aktuelle Prompttext neu ausgegeben.

Siehe auch:

o_rebuild()

rem_background — Hintergrund-Window schließen

Definition:

```
BOOLEAN rem_background(objp)
obj *objp;
```

Beschreibung:

bringt einen mittels new_background() eingefügten Hintergrund zum Verschwinden, so dass die darunterliegenden Windows wieder sichtbar werden. Argument muss der vom letzten new_background()-Aufruf zurückgegebene Identifikator sein.

Ergebnis:

TRUE bei Erfolg
FALSE sonst

Siehe auch:

new_background()

6 Menü

en_get_props — anwendungsspezifische Information zum Menüpunkt abfragen

Definition:

```
PrivatePropertyType en_get_props(f)
menue_entry *f;
```

Beschreibung:

liefert den initialen (aus der Beschreibungsdatei übernommenen) bzw. den mittels en_set_props() zum Menüpunkt *f* gespeicherten (anwendungsspezifischen) Wert.

Ergebnis:

augenblicklicher Wert

Siehe auch:

en_set_props()

Hinweis:

Die Struktur PrivatePropertyType ist in `fix/props.h` definiert.

en_set_props — anwendungsspezifische Information zum Menüpunkt hinterlegen

Definition:

void en_set_props(f, value)
menue_entry *f; PrivatePropertyType value;

Beschreibung:

speichert zum Menüpunkt *f* den (anwendungsspezifischen) Wert *value*.

Siehe auch:

en_get_props()

Hinweis:

Die Struktur PrivatePropertyType ist in `fix/props.h` definiert.

fx_men_jmp — Kontext zu einem Menü-Event ermitteln

Definition:

int fx_men_jmp(menp)
menue *menp;

Beschreibung:

dient dazu, in der Anwendungslogik zu einem Menü den Kontext zu bestimmen, in dem der Aufruf der Anwendungslogik erfolgt ist.

Ergebnis:

1 - der Aufruf erfolgt während eines Rücksprungs in ein übergeordnetes Menü
0 - der Aufruf erfolgt nicht während eines Rücksprungs
< 0 - das Objekt *menp* ist kein Menü oder ist nicht in Bearbeitung

loadmenue — Menü laden

Definition:

menue *loadmenue(mname)
char *mname;

Beschreibung:

lädt das in der Beschreibungsdatei *mname* definierte Menü.

Ergebnis:

Zeiger auf die Interndarstellung bei Erfolg
(menue *)0 bei Fehler

Siehe auch:

loadmask(), loadchoice(), loadselo()

men_entry — Adresse eines Menüelements aus Position ermitteln

Definition:

menue_entry *men_entry(menp, n)
menue *menp; int n;

Beschreibung:

ermittelt einen Zeiger auf das Element des Menüs *menp* an *n*-ter Position der Beschreibungsdatei.

Ergebnis:

Zeiger auf Menüelement bei Erfolg
(menue_entry *)0 bei Fehler

men_moveto — auf Menüpunkt positionieren

Definition:

BOOLEAN men_moveto(menp, n)
menue *menp; int n;

Beschreibung:

positioniert auf den *n*-ten Menüpunkt des Menüs *menp* ($0 \leq n < menp$->ms_anz_f).

Ergebnis:

TRUE bei Erfolg
FALSE bei Fehler (*menp* kein Menü, kein *n*-ter Menüpunkt oder Menüpunkt verborgen)

Hinweis:

Ist *menp* am Bildschirm sichtbar, aber nicht das aktive Objekt, muss *menp* neu dargestellt werden.

7 Feld

chlook — Feldwert in einer Choice nachschlagen

Definition:

int chlook(f, chp)
field *f; choice *chp;

Beschreibung:

prüft, ob der Wert des Feldes *f* in der Datenmenge der Choice *chp* vorkommt, d.h. ein Tupel in der Memory Relation existiert, dessen ausgezeichnete (Voreinstellung: erste) Spalte mit dem Feldwert übereinstimmt; Spaltentyp und Feldtyp müssen hierbei gleich sein.

Ergebnis:

0 - gefunden
6010 - nicht gefunden
255 - Choice nicht ausführbar (Ursache wird gemeldet)
!= 0 - Fehler

Siehe auch:

chmust(), ch_exchange(), fxvalcmp()

Hinweis:

Bei einer dynamischen Choice wird die Datenmenge neu bestimmt.

Eine Choice, die keine Datenmenge besitzt, wird wie eine mit leerer Datenmenge behandelt.

Bei einem Feld vom Typ FXTRUTHTYPE ist auch eine Spalte vom Typ FXCHARTYPE zulässig.

Bei den Typen FXDTIMETYPE und FXINVTYPE müssen die Qualifier von Feld und Spalte im Sinne von fxvalcmp() kompatibel sein.

chmust — Feldwert in zugehöriger Choice nachschlagen

Definition:

int chmust(chp)
choice *chp;

Beschreibung:

Die Choice *chp* muss mit einem Feld verbunden sein. chmust() prüft, ob der Wert des Feldes, mit dem *chp* verbunden ist, in der Datenmenge von *chp* vorkommt. Mittels dieser Funktion wird die Feldeigenschaft DBMUST ausgewertet.

Ergebnis:

0 - gefunden
6010 - nicht gefunden
255 - Choice nicht ausführbar (Ursache wird gemeldet)
!= 0 - Fehler

Siehe auch:

chlook(), dbmust()

Hinweis:

Zur Suche wird chlook() benutzt.

Die Wertübernahme-Funktion von *chp* wird nicht ausgewertet.

dbmust — Feldwert in der Datenbank nachschlagen

Definition:

int dbmust(sep)
selo *sep;

Beschreibung:

Das Selo *sep* muss mit einem Feld verbunden sein. dbmust() prüft, ob der Wert des Feldes, mit dem *sep* verbunden ist, in der durch das Selo definierten Ergebnismenge vorkommt. Mittels dieser Funktion wird die Feldeigenschaft DBMUST ausgewertet.

Bei Erfolg wird der gefundene Satz, der in der entsprechenden Spalte diesen Wert hat, für fputselo() bereitgestellt und versucht, die Werte der exportierten Spalten in die angegebenen Felder zu schreiben.

Ergebnis:

0 - gefunden
6010 - nicht gefunden
255 - Selo nicht ausführbar (Ursache wird gemeldet)
!= 0 - Fehler bei Suche

Siehe auch:

f_reset_selo_targets(), chmust(), fputselo()

f_AssignFormat — Feldformat ersetzen

Definition:

int f_AssignFormat(f, fmtstring, display)
field *f; char *fmtstring; BOOLEAN display;

Beschreibung:

tauscht bei Feldern mit Formatangabe das Format aus. Sowohl das Feld *f* als auch das zu setzende Format *fmtstring* müssen eine Reihe von Bedingungen erfüllen:

- Das Feld muss bereits ein Format besitzen.
- Das Feld muss einen numerischen Datentyp besitzen.
- Bei dem Feld darf es sich nicht um einen Feld-Link handeln.
- Bei dem Feld darf es sich nicht um ein Root-Feld handeln.
- Zum Zeitpunkt des Aufrufs darf kein Aufruf von fx_accept() für dieses Feld anhängig sein.
- Das neue Format darf nicht länger als das ursprünglich in der Beschreibungsdatei spezifizierte Format sein.
- In der Ressource SwapOptions muss das Bit 1 gesetzt sein oder die globale Variable allowAssignFormat muss den Wert 1 besitzen.

Ein erfolgreicher Aufruf verändert die Werte der Komponenten f->fmtstring, f->len und f->displen:

- f->fmtstring zeigt auf eine Kopie des als Argument übergebenen Formatstrings.
- f->len und f->displen enthalten jeweils die Länge des neuen Formatstrings.

Darüberhinaus wird der Inhalt von f->info auf die neue Länge verkürzt. Der Parameter display bestimmt durch seinen Wert TRUE, dass das Feld nach der Änderung des Formats neu dargestellt werden soll.

Achtung:

Eine Änderung des Formats kann u.U. die spätere Einrichtung eines Feld-Links unmöglich machen, da hierfür i.d.R. die Übereinstimmung der Formate erforderlich ist.

Ergebnis:

1 - Erfolg
-1400 - Fehler in Format *fmtstring*
-1 - Feld ungeeignet (besitzt kein Format, Feld-Link, Root-Feld)
-2 - Feld und Formatstring passen nicht zueinander

Siehe auch:

f_RestoreFormat(), f_RemoveFormat()

f_color — Feldattribute abfragen oder ändern

Definition:

color f_color(f, mode, estate, color)
field *f; int mode; int estate; color color;

Beschreibung:

dient zur Behandlung der Videoattribute des Feldes *f*. Das Verhalten der Funktion wird bestimmt vom Parameter *mode*. *mode* == P_ADD verleiht dem Feld *f* das Videoattribut *color* im Zustand *estate*. Mit *mode* == P_GET kann abgefragt werden, welches Videoattribut *f* im Zustand *estate* besitzt.

estate muss einer der Zustände F_ACTIVE, F_INACTIVE, F_NOENT oder F_DISPLACED sein, *color* eines der Videoattribute aus `video.h`.

Ergebnis:

das neu eingestellte Videoattribut (*mode* == P_ADD)
color bzw. 0, je nachdem ob *color* zurzeit eingestellt ist oder nicht (*mode* == P_GET)

Hinweis:

Videoattribut-Kombinationen werden nicht unterstützt.

Während der Erfassung der Parameter für die Selo-Suche können, wenn dafür Felder von Tabellenmasken benutzt werden, Probleme beim Bildschirmaufbau auftreten.

f_format — Feldformat ermitteln

Definition:

```
char *f_format(f)
field *f;
```

Beschreibung:

bestimmt zu einem Feld *f* dessen Formatstring und hinterlegt ihn '\0' terminiert in einem statischen Bereich, der bei jedem Aufruf überschrieben wird.

Formatstrings werden bei FXDTIMETYPE- und FXINVTYPE-Feldern sowie - als Option - bei allen Arten numerischer Felder unterstützt.

Ergebnis:

Zeiger auf einen statischen Bereich, der den Formatstring enthält,
(char *)0, wenn das Feld keinen Formatstring besitzt

Siehe auch:

f_qual()

Hinweis:

Die Länge eines FXDTIMETYPE-Formatstrings beträgt höchstens 19 Zeichen (Teilstring von "yyyy-mm-dd/hh:nn:ss").

f_get_props — anwendungsspezifische Information zum Feld abfragen

Definition:

```
PrivatePropertyType f_get_props(f)
field *f;
```

Beschreibung:

liefert den initialen (aus der Beschreibungsdatei übernommenen) bzw. den mittels f_set_props() zum Feld *f* gespeicherten (anwendungsspezifischen) Wert.

Ergebnis:

augenblicklicher Wert

Siehe auch:

f_set_props()

Hinweis:

Die Struktur PrivatePropertyType ist in `fix/props.h` definiert.

f_isnull — Feld auf NULL prüfen

Definition:

BOOLEAN f_isnull(f)
field *f;

Beschreibung:

testet, ob der Wert des Feldes *f* NULL ist.

Ergebnis:

TRUE, wenn *f* den Wert NULL hat
FALSE, wenn der Wert von *f* ungleich NULL ist

Siehe auch:

fxisnull()

f_qual — Feld-Qualifier ermitteln

Definition:

int f_qual(f)
field *f;

Beschreibung:

bestimmt zu einem Feld *f* vom Typ FXDTIMETYPE oder FXINVTYPE dessen Qualifier.

Ergebnis:

> 0 - Qualifier gemäß `fix/fixdatetime.h`
0 - Fehler

Siehe auch:

f_format()

f_RemoveFormat - Feldformat entfernen

Definition:

int f_RemoveFormat(f, newlen, newfmt, newdisplen, display)
field *f; int newlen; int newfmt; int newdisplen; BOOLEAN display;

Beschreibung:

Die Funktion entfernt das Format des Feldes f. Danach werden die als Parameter übergebenen Längen zur Formatierung des Feldwertes verwendet.

Der Wert newlen darf nicht größer sein als die beim Laden des Feldes durch das Format definierte Länge. Ansonsten würde der Puffer zur Felderfassung (f->info) überschrieben. newdisplen muss größer oder gleich newlen sein. Der Parameter display bestimmt, ob ein Feld nach der Formatänderung neu dargestellt werden soll. Dieser Parameter ist beim Aufruf aus der Funktion an S_SetFormatHook unbedingt mit FALSE zu besetzen.

Zur Zeit wird nur das Entfernen von Formaten in numerischen Feldern unterstützt.

Ein einmal entferntes Format kann wieder restauriert werden.

Ergebnis:

1 - Das Format wurde entfernt.

0 - Das Feld besitzt kein Format oder das Format wurde bereits vorher entfernt.

-1 - Ein Entfernen des Formates ist nicht erlaubt, weil:

- Bit 1 der Ressource SwapOptions nicht gesetzt ist (siehe unten).
- vom Benutzer gerade ein Feldwert eingegeben wird (*FIX* befindet sich in der Felderfassungsroutine faccept()).
- newdisplen < newlen ist.
- das Feld ein Link-Feld ist.
- das Feld ein Root-Feld ist.
- der Wert newlen zu groß ist.
- das Feld kein numerisches Format besitzt.

-3 - Fehler beim Anfordern von Speicherplatz.

Siehe auch:

f_RestoreFormat(), f_AssignFormat()

f_reset_selo_targets — Zielspalten des Selos reinitialisieren

Definition:

```
int f_reset_selo_targets(f)
field *f;
```

Beschreibung:

Das Feld *f* muss mit einem Selo verknüpft sein. f_reset_selo_targets() weist den Feldern, in die das Selo Werte exportiert, den Defaultwert zu, sofern es sich bei dem Feld nicht um *f* selbst handelt.

Ergebnis:

≥ 0 - Erfolg (Anzahl der exportierenden Spalten)
< 0 - Fehler

Siehe auch:

dbmust(), fputval()

Hinweis:

Felder ohne Defaultwert oder mit dynamischem Defaultwert werden gelöscht.

f_RestoreFormat - Feldformat restaurieren

Definition:

```
int f_RestoreFormat(f, display)
field *f; BOOLEAN display;
```

Beschreibung:

Restauriert das Format eines Feldes. Das Format muss vorher mit f_RemoveFormat() entfernt worden sein. Display, mit dem Wert TRUE bestimmt, dass das Feld nach der Änderung des Formats neu dargestellt werden soll.

Ergebnis:

1 - Das Format wurde restauriert.

0 - Das Feld besitzt bereits ein Format.

-1 - Ein Restaurieren des Formates ist nicht erlaubt, weil:

- Bit 1 der Ressource SwapOptions nicht gesetzt ist (siehe unten).
- Vom Benutzer gerade ein Feldwert eingegeben wird (*FIX* befindet sich in der Felderfassungsroutine faccept()).

- Das Feld ein Link-Feld ist.
- Das Feld ein Root-Feld ist.
- Das Feld kein numerisches Feld ist.
- Zu dem Feld kein mit f_removeFormat() entferntes Format vorhanden ist.

Siehe auch:

f_RemoveFormat(), f_AssignFormat()

f_set_props — anwendungsspezifische Information zum Feld hinterlegen

Definition:

void f_set_props(f, value)
field *f; PrivatePropertyType value;

Beschreibung:

speichert zum Feld *f* den (anwendungsspezifischen) Wert *value*.

Siehe auch:

f_get_props()

Hinweis:

Die Struktur PrivatePropertyType ist in `fix/props.h` definiert.

f_state — Feldeigenschaften abfragen oder ändern

Definition:

unsigned long f_state(f, mode, state)
field *f; int mode; unsigned long state;

Beschreibung:

dient zur Behandlung der Eigenschaften des Feldes *f*. Das Verhalten der Funktion wird bestimmt vom Parameter *mode*. *state* muss eine sinnvolle Kombination von Eigenschaften wie NOENT, NOMOD etc. sein (eine Plausibilitätsprüfung erfolgt nicht). *mode* == P_SET gibt, *mode* == P_ADD verleiht zusätzlich, *mode* == P_DEL entzieht dem Feld *f* die Eigenschaften *state*. Mit *mode* == P_GET kann abgefragt werden, welche der Eigenschaften in *state f* besitzt.

Ergebnis:

die gesetzten Eigenschaften (*mode* == P_SET)
die zusätzlich gesetzten Eigenschaften (*mode* == P_ADD)
die nicht länger gesetzten Eigenschaften (*mode* == P_DEL)
die *f* eigenen Eigenschaften aus *state* (*mode* == P_GET)

Siehe auch:

f_type(), Makros STYP, SETSTYP, DELSTYP

Hinweis:

Mittels P_GET mit *state* == (unsigned long)~0 können alle Eigenschaften von *f* abgefragt werden.

Bei Feld-Links wird die Eigenschaft TOUCHED vom Root-Feld hergeleitet.

Das Setzen/Löschen der Eigenschaft ZERO_VALUED wird nicht unterstützt.

f_type — Feldtyp ermitteln

Definition:

int f_type(f)
field *f;

Beschreibung:

bestimmt zu einem Feld *f* dessen Typ.

Ergebnis:

> 0 - Typ gemäß `fix/fxtypes.h`

Siehe auch:

f_state()

faccept — Feldwert erfassen

Definition:

Event faccept(f)
field *f;

Beschreibung:

realisiert die Feldbearbeitung und übernimmt die Benutzereingabe in die korrespondierende Feld-Hostvariable.

Ergebnis:

Event, das von faccept() nicht behandelt werden konnte

Siehe auch:

fx_accept()

Hinweis:

faccept() unterstützt keine Felder mit obligatorischer oder bevorzugter Choice (**use**, **offer**).

Der Gebrauch von faccept() wird nicht empfohlen, sinnvoller ist fx_accept(), falls überhaupt das Bedürfnis besteht, an perform() vorbei einen Feldinhalt zu erfassen.

fdisp — Feld darstellen

Definition:

int fdisp(f)
field *f;

Beschreibung:

gibt das Feld *f* am Bildschirm aus, falls es nicht die Eigenschaft NODISPL hat.

Eine mit dem Feld verbundene, am Bildschirm dargestellte Choice oder Feld-Links auf das Feld werden ebenfalls neu ausgegeben, auch wenn das Feld selbst die Eigenschaft NODISPL besitzt.

Ergebnis:

die Länge der Felddarstellung

Siehe auch:

wrktoinfo()

Hinweis:

Diese Funktion wird in Anwendungen selten direkt, sondern meist über wrktoinfo() aufgerufen.

ferase — Feld löschen

Definition:

void ferase(f, display)
field *f; int display;

Beschreibung:

löscht die Feld-Hostvariable des Feldes *f*.

Hinweis:

In der Regel erhält das Feld den Wert NULL bzw., unterstützt das Feld NULL-Werte nicht, so werden numerische Felder auf 0, CHARACTER- und Wahrheitswert-Felder auf " " gesetzt.

Die Funktion hat folgende Nebeneffekte:
Gehört *f* zu einer Mehrsatz-Maske, so wird der Feldwert in den aktuellen Satz zu übertragen versucht.
Ist *f* das Entscheidungsfeld einer Tabellenmaske mit Zeilentypen, so wird der Zeilentyp neu gesetzt.
Die Felddarstellung wird neu aufgebaut und das Feld ausgegeben.

Der (zur Wahrung der Kompatibilität beibehaltene) Parameter *display* wird nicht ausgewertet.

fgetstring — Standard-Notation für Feldwert ermitteln

Definition:

char *fgetstring(f)
field *f;

Beschreibung:

liefert die Standard-Notation für den Wert des Feldes *f*. Bei Zahlen wird als Dezimalpunkt '.' verwendet, Datum-Werte werden gemäß dem Standard-Datumsformat, Zeitpunkt- und Zeitspannen-Werte normkonform und Wahrheitswerte durch die in S_no und S_yes hinterlegten Zeichen und NULL als Leerstring wiedergegeben.

Bei FXGRAPHICSTYPE-Werte enthält das Ergebnis den Wert, gefolgt von '\0'.

Ergebnis:

Zeiger auf statischen Bereich von MAXFIELDLEN Bytes, der bei jedem Aufruf überschrieben wird.

Siehe auch:

fputstring()

Hinweis:

Das Standard-Datumsformat ist "dd.mm.yyyy", sofern die Umgebungsvariable FXDATE nichts anderes spezifiziert.

Die Funktion wird auch von *FIX* intern benutzt, z.B. von stdimport() oder exec_ms_command().

fgetval — Wert aus Feld-Hostvariable an Adresse kopieren

Definition:

void fgetval(radr, f)
char *radr; field *f;

Beschreibung:

kopiert den Wert der Feld-Hostvariable des Feldes *f* an die Adresse *radr*. Bei FXCHARTYPE-Feldern wird der kopierte Wert mit '\0' terminiert.

Siehe auch:

fputval()

Hinweis:

In der Feld-Hostvariable vorgefundene unzulässige Werte bleiben erhalten.

radr muss auf eine zur Aufnahme geeignete Speicherfläche zeigen.

fis_empty — Feldinhalt prüfen

Definition:

```
BOOLEAN fis_empty(f)
field *f;
```

Beschreibung:

testet, ob die Darstellung des Feldes *f* signifikante Zeichen enthält.

Ergebnis:

TRUE, wenn die Darstellung des Wertes von *f* keine Zeichen außer Leerzeichen enthält
FALSE sonst

Siehe auch:

wrk_empty()

Hinweis:

Die Anwendung dieser Funktion auf Felder mit Formatstring ist problematisch, da das Ergebnis dann nicht nur vom Wert, sondern auch vom Formatstring abhängt.

fis_empty() erlaubt es i.d.R. nicht, zwischen NULL und 0 bzw. “ ” zu unterscheiden.

fputselo — Feldwert aus Selo-Spalte übernehmen

Definition:

```
BOOLEAN fputselo(target, f, colname)
field *target; field *f; char *colname;
```

Beschreibung:

übernimmt den Wert der Datenbankspalte *colname*, die zur Ergebnismenge des Selos zum Feld *f* gehören muss, als Wert des Feldes *target*. Voraussetzung ist, dass eine vorangegangene Datenbanksuche (über Auswahl im Selo oder dbmust()) erfolgreich war.

Beim Argument *colname* wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Ergebnis:

TRUE, wenn Wert übernommen wurde
FALSE sonst (Wert von *target* wurde nicht verändert)

Siehe auch:

Makro FPUTSELO

Hinweis:

Stimmen Feld- und Spaltentyp nicht überein, versucht *FIX*, den Wert zu konvertieren, sofern beide numerisch sind.

Bei FXDTIMETYPE und FXINVTYPE muss der Qualifier der Feld-Hostvariable von *target* gleich 0 sein oder mit dem Qualifier der Spalte übereinstimmen.

Enthält die Spalte den Wert NULL, erhält das Feld den Wert NULL bzw., unterstützt das Feld NULL-Werte nicht, so werden numerische Felder auf 0, CHARACTER- und Wahrheitswert-Felder auf “ ” gesetzt.

Das TOUCHED-Attribut des Feldes *target* wird nicht verändert.

Die Funktion hat bei Erfolg der Zuweisung folgende Nebeneffekte:
Gehört *f* zu einer Mehrsatz-Maske, so wird der Feldwert in den aktuellen Satz zu übertragen versucht.
Ist *f* das Entscheidungsfeld einer Tabellenmaske mit Zeilentypen, so wird der Zeilentyp neu gesetzt.
Die Felddarstellung wird neu aufgebaut und das Feld ausgegeben.

fputstring — Feldwert entsprechend Standard-Notation setzen

Definition:

BOOLEAN fputstring(f, str)
field *f; char *str;

Beschreibung:

setzt den Wert des Feldes *f* entsprechend der Standard-Notation *str*. Bei Zahlen muss als Dezimalpunkt ‘.’ verwendet werden, Datum-Werte werden gemäß dem Format aus FXDATE (oder “dd.mm.yyyy” wenn nicht gesetzt), Zeitpunkt- und Zeitspannen-Werte normkonform und Wahrheitswerte in Form der in S_no und S_yes hinterlegten Zeichen erwartet.

Ergebnis:

TRUE bei Erfolg
FALSE sonst

Siehe auch:

fgetstring()

Hinweis:

Im Fehlerfall erfolgt keine Zuweisung; bei CHARACTER-Feldern werden höchstens *f*->len Zeichen übernommen.

Außer bei CHARACTER- und FXGRAPHICSTYPE-Feldern wird eine Folge von Leerzeichen wie ein Leerstring behandelt.

Zeigt *str* auf einen Leerstring, erhält das Feld den Wert NULL bzw., unterstützt das Feld NULL-Werte nicht, so werden numerische Felder auf 0, CHARACTER- und Wahrheitswert-Felder auf “ ” gesetzt.

Bei FXGRAPHICSTYPE-Feldern muss *str* auf einen Leerstring oder auf *f*->len Zeichen mit Codes größer-gleich 32 zeigen.

Die Funktion hat unabhängig vom Erfolg der Zuweisung folgende Nebeneffekte:

Gehört *f* zu einer Mehrsatz-Maske, so wird der Feldwert in den aktuellen Satz zu übertragen versucht.

Ist *f* das Entscheidungsfeld einer Tabellenmaske mit Zeilentypen, so wird der Zeilentyp neu gesetzt.
Die Felddarstellung wird neu aufgebaut und das Feld ausgegeben.

fputval — Wert von Adresse in Feld-Hostvariable kopieren

Definition:

void fputval(f, vadr, display)
field *f; char *vadr; int display;

Beschreibung:

erwartet an der Adresse *vadr* einen Wert vom Datentyp des Feldes *f*; dieser wird als Feldwert in die zum Feld *f* korrespondierende Feld-Hostvariable übernommen.

Siehe auch:

fgetval(), wrktoinfo()

Hinweis:

Bei CHARACTER-Feldern werden höchstens *f*->len Zeichen übertragen.

Bei FXDTIMETYPE und FXINVTYPE muss der Qualifier der Feld-Hostvariable von *target* gleich 0 sein oder mit dem Qualifier des Wertes an der Adresse *vadr* übereinstimmen.

Wird an der Adresse *vadr* die Kodierung des Wertes NULL vorgefunden, erhält das Feld den Wert NULL bzw., unterstützt das Feld NULL-Werte nicht, so werden numerische Felder auf 0, CHARACTER- und Wahrheitswert-Felder auf " " gesetzt.

Bei FXGRAPHICSTYPE-Feldern muss *vadr* auf die Kodierung von NULL oder auf *f*->len Zeichen mit Codes größergleich 32 zeigen.

Die Funktion hat folgende Nebeneffekte:
Gehört *f* zu einer Mehrsatz-Maske, so wird der Feldwert in den aktuellen Satz zu übertragen versucht.
Ist *f* das Entscheidungsfeld einer Tabellenmaske mit Zeilentypen, so wird der Zeilentyp neu gesetzt.
Die Felddarstellung wird neu aufgebaut und das Feld, falls *display* != 0, ausgegeben. *display* sollte gewöhnlich 1 sein.

fset_tooltiptext — Text für den Feld-Tooltip setzen

Definition:

void fset_tooltiptext(f, text)
field *f; char *text;

Beschreibung:

Die Funktion setzt die Komponente *tooltiptext* der Feldstruktur. Dieser Text wird als Tooltiptext an *FIX/Win* übertragen, wenn *FIX/Win* als Frontend genutzt wird und der Benutzer den Mauszeiger eine gewisse Zeit über einem Feld stehen läßt. *f* ist der Zeiger auf die Feldstruktur, an der der Tooltiptext gesetzt werden soll, und *text* ist der zu setzende Text.

Hinweis:

Tooltips dürfen Newlines enthalten. Dadurch wird das Tooltip-Fenster mehrzeilig dargestellt. Die Breite richtet sich nach dem längsten Teilstück des Textes. Weitere Hinweise finden sich in dem Abschnitt "Tooltips auf Feldern" auf Seite 184.

fx_bindfield — Auswahl-Objekt an Feld hängen

Definition:

BOOLEAN fx_bindfield(f, objp)
field *f; obj *objp;

Beschreibung:

bindet das Objekt *objp*, bei dem es sich um ein Selo oder eine Choice handeln muss, an das Feld *f*. Weder darf *objp* bereits mit einem Feld verknüpft noch ein Selo oder eine Choice bereits mit *f* verknüpft sein. *f* und *objp* müssen zueinander kompatibel sein.

Ergebnis:

TRUE bei Erfolg
FALSE sonst

Siehe auch:

fx_unbindfield(), loadselo()

Hinweis:

Handelt es sich bei *objp* um eine Choice und ist *f*->f_slctmode == ch_none (Anschlussmodus undefiniert), wird *f*->f_slctmode ch_demand zugewiesen (konventionelle Choice), sonst bleibt der Wert unverändert.

fx_unbindfield — Auswahl-Objekt von Feld abhängen

Definition:

BOOLEAN fx_unbindfield(f, objp)
field *f; obj *objp;

Beschreibung:

löst die Bindung des Objekts *objp*, bei dem es sich um das/die an das Feld *f* gebundene Selo/Choice handeln muss, mit *f*.

Ergebnis:

TRUE bei Erfolg
FALSE sonst

Siehe auch:

fx_bindfield()

Hinweis:

Handelt es sich bei *objp* um eine Choice, wird *f*->f_slctmode ch_none zugewiesen.

fx_selo_fld_adr — Zeiger auf Spalte eines mittels Selo gefundenen Satzes ermitteln

Definition:

char *fx_selo_fld_adr(f, colname)
field *f; char *colname;

Beschreibung:

liefert die Adresse des Wertes der Datenbankspalte *colname*, die zur Ergebnismenge des Selos des Feldes *f* gehören muss, nachdem eine Datenbanksuche über Selo-Auswahl oder dbmust() stattgefunden hat und erfolgreich war. Der Wert kann von hier typgerecht kopiert werden.

Beim Argument *colname* wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Ergebnis:

Zeiger auf Speicherfläche, die Spaltenwert enthält
(char *)0, falls *f* kein Selo zugeordnet ist oder *colname* nicht in der Ergebnismenge vorkommt

Siehe auch:

dbmust(), fputselo(), Makro F_PUT_SELO()

Hinweis:

Bei SQL-Selos bleiben im Gegensatz zu C-ISAM-Selos alle NULL-Werte erhalten.

Bei Spalten mit Typ DECIMAL wird die Adresse einer dec_t-Variablen zurückgegeben, bei solchen mit Typ MONEY die Adresse einer double-Variablen.

fxdtypsize — Anzahl Bytes für Feld-Hostvariable ermitteln

Definition:

int fxdtypsize(f)
field *f;

Beschreibung:

liefert die Größe der Speicherfläche, die für eine typgerechte Speicherung von Werten des Feldes *f* erforderlich ist.

Ergebnis:

Speicherplatzbedarf in Bytes

wrk_empty — Feld-Hostvariable prüfen

Definition:

BOOLEAN wrk_empty(f)
field *f;

Beschreibung:

testet, ob das Feld *f* leer ist. Unterstützt das Feld *f* NULL-Werte, so gilt nur NULL als leer; anderenfalls gilt bei numerischen Feldern 0 bzw. bei CHARACTER- und Wahrheitswert-Feldern " " als leer.

Ergebnis:

TRUE, wenn das Feld leer ist
FALSE sonst

Siehe auch:

fis_empty()

wrktoinfo — Feldinhalt aus Feld-Hostvariable übernehmen

Definition:

BOOLEAN wrktoinfo(f, display)
field *f; int display;

Beschreibung:

baut die textuelle Darstellung des Wertes des Feldes *f* auf und überträgt, sofern *f* zu einer Mehrsatz-Maske gehört, den Feldwert auch in den aktuellen Satz. Falls *display* != 0, wird das Feld anschließend ausgegeben.

Ergebnis:

FALSE, wenn Wert nicht darstellbar
TRUE sonst

Siehe auch:

m_wrktoinfo()

Hinweis:

Wird als Wert NULL vorgefunden, obwohl das Feld *f* NULL nicht unterstützt, so wird der Wert bei numerischen Feldern durch 0, bei CHARACTER- und Wahrheitswert-Feldern durch " " ersetzt.

Ist der Wert nicht den Feldeigenschaften entsprechend formatierbar, wird eine Fehlerdarstellung erzeugt.

Für *display* sollte 1 angegeben werden.

8 Maske

buftovar — Satz aus Backup-Puffer restaurieren

Definition:

BOOLEAN buftovar(mskp, buf, buflen)
mask ∗mskp; char ∗buf; int buflen;

Beschreibung:

liest die Werte der Felder der Maske *mskp* von der Adresse *buf*, wobei *buf* ein vom Entwickler bereitgestellter Speicherbereich von *buflen* Bytes ist, der mittels vartobuf() gerettete Werte enthält.

Ergebnis:

TRUE bei Erfolg
FALSE, wenn Speicherbereich zu klein

Siehe auch:

vartobuf()

Hinweis:

berücksichtigt auch Feld-Links.

exec_ms_command — mit Objekt assoziiertes Kommando ausführen

Definition:

int exec_ms_command(mskp)
mask ∗mskp;

Beschreibung:

führt die mit der Maske *mskp* assoziierte Aktion aus (sofern vorhanden).

Handelt es sich um ein sh-Kommando, wird $i ersetzt durch den Inhalt des i-ten Feldes ($0 \leq i <$ Anzahl Maskenelemente, *mskp*->f[i].ob_class == FIELD), $∗ durch die Inhalte aller Felder, jeweils eingeschlossen in ‘"’. Die Darstellung von Werten entspricht der von fgetstring().

Ergebnis:

Endestatus

Siehe auch:

execute_cmd(), fgetstring()

findVariant —Varianten suchen

Definition:

mask ∗findVariant(mask ∗mskp, int n, BOOLEAN tablemode)
mask ∗mskp; int n; BOOLEAN tablemode;

Beschreibung:

Die Funktion sucht eine Variante zu einer Maske und liefert einen Zeiger auf die Variante zurück. Ist tablemode == TRUE, so wird nach einer Zeilentyp-Variante der Tabellenmaske gesucht.

Beim Ergebnis handelt es sich (außer bei Parameter *n*==0) nicht um die Hauptmaske, sondern um die in den Speicher als Variante geladene Maskenbeschreibung. Dies ist keine vollständig belegte Maskenstruktur. Beispielsweise besitzen

die Felder keine info-Komponente und der Objekttyp der Maske enthält zusätzliche Flags. Die so ermittelte Maske sollte aus diesem Grund zur Zeit nur zum Austausch und Entfernen von Feldformaten verwendet werden.

Wird eine vollwertige Maskenstruktur benötigt, dann ist mit m_setvariant() die Variante in die aktuelle Maskenstruktur zu kopieren.

Liefert die Funktion NULL zurück, dann bedeutet dies entweder, dass es die Variante nicht gibt (Wert für Parameter n zu groß) oder dass die Variante zum Zeitpunkt des Aufrufs noch nicht geladen wurde. Die Funktion lädt selbst keine Masken als Varianten. Zeilenvarianten werden von *FIX* unmittelbar nach dem Laden einer Maske geladen. Maskenvarianten werden erst beim ersten Aktivieren der Variante geladen und sind somit erst dann verfügbar.

Ergebnis:

Zeiger auf die Maskenvariante/Zeilentyp-Variante oder NULL.

fx_accept — Maskenfeld besuchen

Definition:

Event fx_accept(mskp, mfo_fnr, prv_fnr)
mask *mskp; int mfo_fnr; int prv_fnr;

Beschreibung:

realisiert den Besuch des Feldes mit Element-Position *mfo_fnr* der Maske *mskp*. Der Parameter *prv_fnr* steuert das Verhalten:
prv_fnr < *mfo_fnr* - Feld wird vorwärts betreten
prv_fnr == *mfo_fnr* - wieder gleiches Feld
prv_fnr > *mfo_fnr* - Feld wird rückwärts betreten

Ergebnis:

Event, durch das der Feldbesuch beendet wurde

Siehe auch:

faccept()

Hinweis:

Im Gegensatz zu faccept() unterstützt diese Funktion auch Felder mit obligatorischer oder bevorzugter Choice (**use**, **offer**).

fx_mfo_f — Zeiger auf ein Maskenelement aus Element-Bezeichner ermitteln

Definition:

field *fx_mfo_f(mskp, prg_nr)
mask *mskp; int prg_nr;

Beschreibung:

ermittelt einen Zeiger auf das Element mit Element-Bezeichner *prg_nr*.

Ergebnis:

Zeiger auf Maskenelement (nicht notwendigerweise Feld) bei Erfolg
(field *)0 bei Fehler

Siehe auch:

fx_mfo_nr(), m_f()

fx_mfo_nr — Position eines Maskenelements aus Element-Bezeichner ermitteln

Definition:

int fx_mfo_nr(mskp, prg_nr)
mask *mskp; int prg_nr;

Beschreibung:

ermittelt die Element-Position des Elements mit Element-Bezeichner *prg_nr*.

Ergebnis:

≥ 0 - Element-Position
-1 - Fehler

Siehe auch:

fx_prg_nr()

fx_prg_nr — Element-Bezeichner eines Maskenelements aus seiner Position ermitteln

Definition:

int fx_prg_nr(mskp, mfo_nr)
mask *mskp; int mfo_nr;

Beschreibung:

ermittelt den Element-Bezeichner des Elements mit Element-Position *mfo_nr*.

Ergebnis:

≥ 0 - Element-Bezeichner
-1 - Fehler

Siehe auch:

fx_mfo_nr()

get_triggering_event — auslösendes Event abfragen

Definition:

Event get_triggering_event()

Beschreibung:

Der Aufruf dieser Funktion macht nur Sinn innerhalb der Behandlung der Events L_LEAVE_RECORD und L_LEAVE_OBJECT. Sie liefert das Event, dessen Behandlung im Begriff ist, ein Verlassen des Satzes bzw. Objekts auszulösen.

Ergebnis:

Event, auf das reagiert wird

Siehe auch:

perform(), m_present()

Hinweis:

In einem anderen als dem obigen Kontext ist das Ergebnis undefiniert.

layout_putstr — Objekt-Layout im Programm verändern

Definition:

BOOLEAN layout_putstr(mskp, y, x, s, display)
mask *mskp; int y; int x; char *s; BOOLEAN display;

Beschreibung:

dient dazu, das Layout einer geladenen Maske zu modifizieren. In die *y*-te Zeile der Maske *mskp* wird, beginnend in Spalte *x*, der String *s* geschrieben. *s* darf die gleichen Zeichen enthalten wie ein Argument für putstr() mit Ausnahme von '\005', da das Videoattribut BLINK im Layout nicht unterstützt wird. Das zuletzt benutzte Videoattribut wird beim nächsten Aufruf restauriert. Ist *display* gleich TRUE, wird die Änderung sofort sichtbar, wenn die Maske am Bildschirm dargestellt ist; sonst erst, wenn dieser Bereich das nächste Mal neu aufgebaut wird.

Ergebnis:

TRUE bei Erfolg
FALSE bei Fehler

Siehe auch:

putstr(), layout_setchar(), layout_getchar(), layout_getcharset(), layout_getvideo()

Hinweis:

Am rechten Maskenrand erfolgt ein “wraparound”.

display sollte gewöhnlich TRUE sein.

Wenn mit layout_putstr() ein Text in ein Layout geschrieben wird und sich unter einer der Zellen, die von diesem Text belegt werden, eine Paintarea befindet, dann wird die komplette Paintarea vorher entfernt. Dazu werden alle Zellen, die zur Paintarea gehören mit Leerzeichen im Attribut NORMAL beschrieben.

loadmask — Maske laden

Definition:

mask *loadmask(mname)
char *mname;

Beschreibung:

lädt die in der Beschreibungsdatei *mname* definierte Maske, alle darin eingebetteten Masken sowie die Feldern zugeordneten Selos und Choices.

Bei Tabellenmasken mit Zeilentypen werden auch die Zeilentypen geladen.

Ergebnis:

Zeiger auf die Interndarstellung

Siehe auch:

loadmenue(), loadchoice()

Hinweis:

Im Fehlerfall erfolgt ein Programmabbruch durch fastexit().

Varianten werden erst beim erstmaligen Versuch, sie zu benutzen, geladen.

mGetRootMask — Wurzel des Objektbaums ermitteln, zu dem eine Maske gehört

Definition:

mask *mGetRootMask(mskp)
mask *mskp;

Beschreibung:

ermittelt die Maske an der Wurzel des Objektbaums, zu dem die Maske *mskp* gehört.

Ergebnis:

Zeiger auf Maske bei Erfolg
(mask *)0 bei Fehler

mIsDescendantOf — prüfen, ob eine Maske zu einem Objektbaum gehört

Definition:

BOOLEAN mIsDescendantOf(mskp, objp)
mask *mskp; obj *objp;

Beschreibung:

prüft, ob die Maske *mskp* zu dem Objektbaum gehört, dessen Wurzel das Objekt *objp* bildet.

Ergebnis:

TRUE, wenn *mskp* zu dem Objektbaum mit Wurzel *objp* gehört
FALSE sonst

m_colno_to_fnr — zu einer Spalte gehörendes Maskenelement ermitteln

Definition:

int m_colno_to_fnr(mskp, colno)
mask *mskp; int colno;

Beschreibung:

liefert die Element-Position des Elements der (Einzel- oder Mehrsatz-)Maske *mskp*, zu dem die *colno*-te Spalte (1 ≤ *colno* ≤ Anzahl Spalten) der Memory Relation gehört.

Ergebnis:

≥ 0 - Element-Position
-2 - unzulässiges Argument
-1 - kein Maskenelement gefunden

Siehe auch:

m_fnr_to_colno()

m_countvariant — Anzahl der Varianten ermitteln

Definition:

int m_countvariant(mskp)
mask *mskp;

Beschreibung:

zählt die Varianten der Maske *mskp* (Hauptmaske eingeschlossen). Das Ergebnis 1 bedeutet “keine echten Varianten vorhanden”.

Bei Tabellenmasken mit Zeilentypen bezieht sich das Ergebnis auf den aktuellen Zeilentyp.

Ergebnis:

Anzahl der Varianten

m_delete_varbutton - Varbuttons löschen

Definition:

BOOLEAN m_delete_varbutton(mskp, posnr)
mask *mskp, int posnr;

Beschreibung:

Die Funktion löscht einen Varbutton. Sie entfernt den Varbutton aus der internen Datenstruktur. Wenn der Varbutton zum Zeitpunkt des Aufrufs sichtbar ist, dann wird versucht, alle anderen Varbuttons neu darzustellen. Dabei kann es dazu kommen, dass Varbuttons, die vorher außerhalb derLeiste lagen, jetzt sichtbar sind.

Ergebnis:

Als Ergebnis liefert die Funktion TRUE bei Erfolg oder FALSE im Fehlerfall. Ein Fehler kann eine der folgenden Ursachen haben:

- Für die Maske *mskp* wurde keine Leiste mit Varbuttons definiert.
- Die als *posnr* übergebene Position liegt außerhalb des Bereichs.
- Für die Position wurde kein Varbutton definiert oder der definierte Varbutton wurde gelöscht.

m_display_data — Maskeninhalt darstellen

Definition:

void m_display_data(mskp)
mask *mskp;

Beschreibung:

stellt den Inhalt der Maske *mskp* neu dar, sofern die Maske am Bildschirm dargestellt ist.

Siehe auch:

o_rebuild()

m_f — Adresse eines Maskenelements aus Element-Position ermitteln

Definition:

field *m_f(mskp, mfo_nr)
mask *mskp; int mfo_nr;

Beschreibung:

ermittelt einen Zeiger auf das Element der Maske *mskp* an Element-Position *mfo_nr.*

Ergebnis:

Zeiger auf Maskenelement (nicht notwendigerweise Feld) bei Erfolg
(field *)0 bei Fehler

Siehe auch:

fx_mfo_f()

m_fnr_to_colno — zu einem Maskenelement gehörende Spalte ermitteln

Definition:

int m_fnr_to_colno(mskp, fnr)
mask *mskp; int fnr;

Beschreibung:

liefert die Nummer der Spalte in der Memory Relation der (Einzel- oder Mehrsatz-)Maske *mskp*, die zu dem Maskenelement an Element-Position *fnr* (0 ≤ *fnr* < Anzahl Maskenelemente) gehört.

Ergebnis:

> 0 - Nummer der Spalte
-2 - unzulässiges Argument
-1 - keine entsprechende Spalte vorhanden

Siehe auch:

m_colno_to_fnr()

m_get_active_varbutton_nr - Nummer des aktuellen Buttons ermitteln

Definition:

int m_get_active_varbutton_nr(mskp)
mask *mskp;

Beschreibung:

Die Funktion liefert die Positionsnummer des aktiven Varbuttons der Maske. Wenn für die Maske keine Leiste mit Varbuttons definiert wurde, liefert die Funktion -1 zurück.

Ergebnis:

> 0 - Positionsnummer des aktiven Varbuttons
-1 - Fehler

m_get_active_varbutton - aktuellen Button ermitteln

Definition:

HPAINTAREA m_get_active_varbutton(mskp)
mask *mskp;

Beschreibung:

Die Funktion liefert die Paintarea des aktiven Varbuttons der Maske. Wenn für die Maske keine Leiste mit Varbuttons definiert wurde, liefert die Funktion HPAINTAREA_INVALID zurück.

Ergebnis:

Handle auf Paintarea des aktiven Buttons
HPAINTAREA_INVALID - bei Fehler

m_get_first_varbutton_nr - Nummer des ersten Buttons ermitteln

Definition:

int m_get_first_varbutton_nr(mskp)
mask *mskp;

Beschreibung:

Liefert die Nummer des ersten sichtbaren Varbuttons.

Ergebnis:

>= 0 - Nummer des ersten sichtbaren Varbuttons
-1 - Fehler

m_get_last_varbutton_nr - Nummer des letzten Buttons ermitteln

Definition:

```
int m_get_last_varbutton_nr(mskp)
mask *mskp;
```

Beschreibung:

Liefert die Nummer des letzten sichtbaren Varbuttons.

Ergebnis:

>= 0 - Nummer des letzten sichtbaren Varbuttons
-1 - Fehler

m_get_varbutton - Varbuttons ermitteln

Definition:

```
HPAINTAREA m_get_varbutton(mskp, posnr)
mask *mskp, int posnr;
```

Beschreibung:

Die Funktion liefert das Handle des Varbuttons der Maske mskp an der Position posnr. Wenn für die Maske keine Leiste mit Varbuttons definiert wurde oder wenn die Positionsnummer außerhalb des Bereichs 0 - cnt liegt, dann liefert die Funktion HPAINTAREA_INVALID zurück. Wenn für die Position kein Varbutton definiert wurde, oder wenn der Varbutton zur Zeit nicht sichtbar ist, dann liefert die Funktion HPAINTAREA_EMPTY zurück.

Ergebnis:

Handle des Varbuttons an Position posnr
HPAINTAREA_INVALID - Fehler
HPAINTAREA_EMPTY - Varbutton z.Zt. nicht sichtbar

m_get_varbutton_cnt - Anzahl Varbuttons ermitteln

Definition:

```
int m_get_varbutton_cnt(mskp)
mask *mskp;
```

Beschreibung:

Die Funktion liefert die maximal mögliche Anzahl Varbuttons für eine Maske. Als Wert wird der bei m_init_varbuttons() als cnt definierte Wert zurückgegeben. Wurde m_init_varbuttons() nicht aufgerufen, dann wird 0 zurückgegeben.

Ergebnis:

> 0 - Anzahl Varbuttons
0 - Objekt hat keine Varbuttons

m_getvariant — aktuelle Variante ermitteln

Definition:

int m_getvariant(mskp)
mask *mskp;

Beschreibung:

liefert, falls es sich bei *mskp* um eine Maske mit Varianten handelt, die Nummer der aktuell benutzten Variante entsprechend der Aufzählung in der Beschreibungsdatei der Hauptmaske.

Bei Tabellenmasken mit Zeilentypen bezieht sich das Ergebnis auf den aktuellen Zeilentyp.

Ergebnis:

≥ 0 - Nummer der aktuellen Variante (0 entspricht der Hauptmaske)
-1 - Fehler

Siehe auch:

m_countvariant(), m_setvariant()

m_init_varbuttons - Reiter für Varianten initialisieren

Definition:

BOOLEAN m_init_varbuttons(mskp, cnt, placement, z, s_from, s_to)
mask *mskp; int cnt; long placement; int z; int s_from; int s_to;

Beschreibung:

Die Funktion definiert durch Angabe des Wertes *cnt* die maximal mögliche Anzahl von Varbuttons und den Bereich, in dem die Varbuttons für die Maske *mskp* und deren Varianten dargestellt werden sollen. Der Wert in *z* bestimmt die Zeile, der Wert in *s_from* bestimmt die erste Spalte und der Wert in *s_to* bestimmt die letzte Spalte. Alle Angaben sind relativ und beziehen sich auf die linke obere Ecke der Maske. In *placement* ist die Ausrichtung der Varbuttons anzugeben. Je nach Wert werden die Varbuttons anders von *FIX/Win* gezeichnet. Folgende Werte sind möglich:

- PA_VBTN_PLMT_TOP - Ausrichtung am oberen Rand.
- PA_VBTN_PLMT_BOTTOM - Ausrichtung am unteren Rand.
- PA_VBTN_PLMT_LEFT - Ausrichtung am linken Rand.
- PA_VBTN_PLMT_RIGHT - Ausrichtung am rechten Rand.

Die beiden letzten Werte werden von *FIX* und *FIX/Win* zur Zeit nicht unterstützt und sind für zukünftige Erweiterungen vorgesehen.

Wenn die Anwendung die Funktion m_init_varbuttons() nicht vor der Definition des ersten Varbuttons aufruft oder wenn die Funktion fehlschlägt, ruft *FIX* die Funktion auf und definiert als maximale Anzahl von Varbuttons die Anzahl Varianten einer Maske. Als Bereich für die Varbuttons wird die erste Zeile der Maske verwendet. Die erste Position liegt bei 1 und die letzte liegt vor der letzten Position der Maske, so dass vorne und hinten noch ein Zeichen Platz zur Darstellung einer Ecke übrigbleibt. Für die Ausrichtung wird der Wert PA_VBTN_PLMT_TOP verwendet.

Ergebnis:

Als Ergebnis liefert die Funktion TRUE bei Erfolg oder FALSE im Fehlerfall. Ein Fehler kann eine der folgenden Ursachen haben:

- Die Maske *mskp* besitzt keine Varianten.
- Für Maske *mskp* wurde die Funktion m_init_varbuttons() bereits aufgerufen.
- Der durch *s_from* und *s_to* definierte Bereich ist breiter als die Maske.
- Der durch *s_from* und *s_to* definierte Bereich ist zu klein und kann nicht die Buttons zum Scrollen aufnehmen (< 6 Zeichen).

- Die durch *z* definierte Zeile liegt außerhalb der Maske.
- Die Spalte *s_from* oder die Spalte *s_to* liegt außerhalb der Maske.

m_mode — Bearbeitungsmodus ermitteln

Definition:

int m_mode(mskp)
mask *mskp;

Beschreibung:

liefert den Bearbeitungsmodus der Maske *mskp*.

Ergebnis:

M_NOT_ACTIVE - Maske nicht in Bearbeitung
M_PERFORM - Maske wird durch perform() bearbeitet
M_PRESENT - Maske wird durch m_present() bearbeitet (ANZEIGE-Modus)

Siehe auch:

m_present(), perform()

m_put_varbutton - Varbutton hinzufügen

Definition:

BOOLEAN m_put_varbutton(mskp, posnr, variant, token, text, tooltiptext, attr, align, bt_mask)
mask *mskp; int posnr; int variant; char *token; char *text; char *tooltiptext; unsigned char attr; unsigned char align; unsigned char bt_mask;

Beschreibung:

Die Funktion überträgt die Daten für einen Varbutton in die maskeninterne Struktur. Die Reihenfolge der Varbuttons wird über eine Positionsnummer bestimmt, die in *posnr* zu übergeben ist. Hierfür ist ein Wert zwischen 0 und der maximal möglichen Anzahl (cnt) zu verwenden (0 < *posnr* < cnt). Der Wert in *variant* definiert die Nummer der Variante, auf die umgeschaltet werden soll. Die Werte in *token*, *text*, *attr*, *align* und *bt_mask* werden in die interne Struktur eingetragen und später zur Erzeugung des Varbuttons verwendet. In *token* ist der Kurztext für die Paintarea anzugeben. Der dargestellte Text kann in *text* angegeben werden. Wird hier ein NULL-Zeiger übergeben, dann wird der Text durch die Übersetzungsfunktion bestimmt. Um dem Varbutton einen Tooltip zuzuordnen, ist für denParameter *tooltiptext* ein Text anzugeben. Soll kein Tooltip angezeigt werden, muss für diesen Parameter der Wert NULL oder eine leere Zeichenkette angegeben werden. Der Parameter *attr* bestimmt das Bildschirmattribut. Der Wert in *align* bestimmt die Ausrichtung des Textes innerhalb des Varbuttons. In *bt_mask* sind die Maustasten zu kodieren, die zum Anklicken verwendet werden können. Die Konstanten, die für *align* und *bt_mask* verwendet werden können, sowie eine genaue Beschreibung der übrigen Parameter ist dem Kapitel über Paintareas zu entnehmen. Als Besonderheit der Varbuttons kommt hinzu, dass in *bt_mask* der Wert für die linke Maustaste enthalten sein muss (PA_BT_LEFT). Ist das nicht der Fall, dann wird dieser Wert automatisch von *FIX* hinzugefügt. Varbuttons sind also immer mit der linken Maustaste anklickbar. Dies ist zugleich die einzige Maustaste, die von *FIX* behandelt wird. Werden zusätzliche Maustasten definiert, dann sind diese in der eigenen Anwendungslogik zu behandeln. Auf die angeklickte Paintarea kann dazu in der gewohnten Art und Weise (pa_get_clicked(), pa_get(), ...) zugegriffen werden.

Ergebnis:

Als Ergebnis liefert die Funktion TRUE bei Erfolg oder FALSE im Fehlerfall. Ein Fehler kann eine der folgenden Ursachen haben:

- Die Maske *mskp* besitzt keine Varianten.
- Die als *posnr* übergebene Position liegt außerhalb des Bereichs.
- Für die Position wurde bereits ein Varbutton definiert.
- Die Maske besitzt keine Variante mit der als *variant* übergebenen Variantennummer.

- Der Varbutton ist größer als der zur Verfügung stehende Platz.
- Die Anwendung hat die Funktion m_init_varbuttons() nicht aufgerufen und der Aufruf von m_init_varbuttons() durch *FIX* schlug fehl.

m_present — Masken-Anzeige-Modus

Definition:

Event m_present(mskp, usr_control)
mask *mskp; Event (*usr_control)(/* obj *, Event */);

Beschreibung:

arbeitet die Maske *mskp* im ANZEIGE-Modus ab. Die Sätze der Maske werden lediglich dargestellt, ihre Elemente können im Gegensatz zu perform() aber nicht betreten werden. Bei Tabellenmasken werden die Felder des aktuellen Satzes unterstrichen.

Ergebnis:

Event, durch das die Maskenbearbeitung beendet wurde (L_NXTFIELD oder L_END)

Siehe auch:

m_mode(), perform()

Hinweis:

Die mitgegebene Funktion *usr_control* wird nur benutzt, wenn die Maske *mskp* keine eigene Logik besitzt, d.h. *mskp*->ev_control == (Event (*)(/* obj *, Event */))0.

m_restore_properties — allen Feldern die ursprünglichen Eigenschaften wiedergegeben

Definition:

void m_restore_properties(mskp)
mask *mskp;

Beschreibung:

setzt die Eigenschaften der Felder der Maske *mskp* zurück wie in der Maskenbeschreibung definiert.

Bei Tabellenmasken mit Zeilentypen wird die Beschreibung des aktuellen Zeilentyps zugrunde gelegt.

Siehe auch:

m_untouch(), f_state()

Hinweis:

entzieht auch die Eigenschaft TOUCHED.

m_scroll_varbuttons - Varbuttons verschieben

Definition:

BOOLEAN m_scroll_varbuttons(mkp, forward)
mask *mskp; BOOLEAN forward;

Beschreibung:

Die Funktion verschiebt die Zeile mit Varbuttons. mskp enthält den Zeiger auf die Maske, deren Buttons verschoben werden soll. forward bestimmt die Richtung, in der die Varbuttons verschoben werden. TRUE bedeutet, dass die Varbuttons nach vorne (links bei horizontaler Anordnung) verschoben werden. FALSE bedeutet, dass die Varbuttons nach hinten (rechts bei horizontaler Anordnung) verschoben werden. Wenn die Maske mskp keine Leiste mit Buttons be-

sitzt, liefert die Funktion FALSE zurück. Ansonsten wird der erste sichtbare Varbutton neu bestimmt und die Leiste mit Varbuttons wird neu dargestellt.

m_show_varbuttons - Varbuttons anzeigen

Definition:

BOOLEAN m_show_varbuttons(mskp)
mask *mskp;

Beschreibung:

Die Funktion entfernt alle Varbuttons vom Bildschirm, restauriert den alten Hintergrund an der Stelle, an der die Leiste mit den Varbuttons zum liegen kommt und stellt alle Varbuttons der Leiste neu (mit den aktuellen Eigenschaften) dar.

Ergebnis:

TRUE bei Erfolg, sonst FALSE.

m_set_active_varbutton - aktuellen Button festlegen

Definition:

BOOLEAN m_set_active_varbutton(mskp, h_pa)
mask *mskp; HPAINTAREA h_pa;

Beschreibung:

Die Funktion dient dazu, den aktiven Button selbst zu bestimmen. Das Ermitteln des aktiven Varbuttons durch *FIX* unterbleibt in diesem Fall, es sei denn, der durch m_set_active_varbutton() festgelegte Varbutton ist nicht der aktuellen Variante zugeordnet. Wenn es für jede Variante nur einen Varbutton gibt, dann ist der Aufruf von m_set_active_varbutton() nicht notwendig.

Ergebnis:

TRUE bei Erfolg, sonst FALSE.

Siehe auch:

m_set_active_varbutton_nr()

m_set_active_varbutton_nr - Nummer des aktuellen Buttons festlegen

Definition:

BOOLEAN m_set_active_varbutton_nr(mskp, posnr)
mask *mskp; int posnr;

Beschreibung:

Die Funktionen dienen dazu, den aktiven Button selbst zu bestimmen. Das Ermitteln des aktiven Varbuttons durch *FIX* unterbleibt in diesem Fall, es sei denn, der durch m_set_active_varbutton_nr() festgelegte Varbutton ist nicht der aktuellen Variante zugeordnet. Wenn es für jede Variante nur einen Varbutton gibt, dann ist der Aufruf von m_set_active_varbutton_nr() nicht notwendig.

Ergebnis:

TRUE bei Erfolg, sonst FALSE.

Siehe auch:

m_set_active_varbutton_nr()

m_setvariant — Variante wechseln

Definition:

BOOLEAN m_setvariant(mskp, n)
mask *mskp; int n;

Beschreibung:

ersetzt die Maske, auf die *mskp* augenblicklich zeigt, durch die *n*-te in der Beschreibungsdatei ihrer Hauptmaske aufgeführte Variante. *n* == 0 wechselt zurück zur Hauptmaske.

Bei Tabellenmasken mit Zeilentypen wird die Beschreibung des aktuellen Zeilentyps zugrunde gelegt.

Ergebnis:

TRUE bei Erfolg
FALSE sonst

Siehe auch:

m_getvariant()

m_touched — feststellen, ob Maske Felder mit der Eigenschaft TOUCHED besitzt

Definition:

BOOLEAN m_touched(mskp)
mask *mskp;

Beschreibung:

ermittelt, ob eines der Felder der Maske *mskp* die Eigenschaft TOUCHED besitzt.

Ergebnis:

TRUE - mindestens ein Feld von *mskp* hat die Eigenschaft TOUCHED
FALSE - kein Feld von *mskp* besitzt die Eigenschaft TOUCHED

Siehe auch:

m_untouch(), f_state(), mr_rowstate()

Hinweis:

berücksichtigt auch Feld-Links.

m_untouch — allen Felder die Eigenschaft TOUCHED entziehen

Definition:

void m_untouch(mskp)
mask *mskp;

Beschreibung:

entzieht allen Feldern der Maske *mskp* die Eigenschaft TOUCHED.

Siehe auch:

m_restore_properties(), m_touched(), f_state()

Hinweis:

berücksichtigt auch Feld-Links.

m_update_varbutton - Varbuttons aktualisieren

Definition:

BOOLEAN m_update_varbutton(mskp, posnr, pa, validmask)
mask *mskp; int posnr; paintarea *pa; unsigned long validmask;

Beschreibung:

Die Funktion ist zum Aktualisieren der Eigenschaften eines Varbuttons aufzurufen. Sie bekommt als Parameter einen Zeiger auf die Maske (*mskp*), die Positionsnummer des Varbuttons (*posnr*), Paintarea-Daten (*pa*) und einen long-Wert, der beschreibt, welche der Paintarea-Daten verändert werden sollen (*validmask*). Damit ist es möglich, nur bestimmte Daten des Varbuttons zu aktualisieren. Eine besondere Bedeutung kommt den Werten in *pa*->pa_props.longval1 und *pa*->pa_props.longval2 zu. Wenn der Wert in longval1 gesetzt wird, dann wird er durch Oder-Verknüpfung bei dem aktiven Varbutton um PA_VBTN_ACTIVE und bei allen Varbuttons um das der Maske zugeordnete Placement (PA_VBTN_PLMT_...) ergänzt. In longval2 ist die Variante des Varbuttons abzulegen. Falls der Varbutton zum Zeitpunkt des Aufrufs sichtbar ist, wird die Funktion pa_update() mit den Werten in *pa* und *validmask* aufgerufen. Der Aufbau des Wertes *pa* und die möglichen Werte für *validmask* sind der Beschreibung der Funktion pa_update() zu entnehmen. Wenn der Aufruf von pa_update() keinen Fehler meldet, oder wenn der Varbutton zum Zeitpunkt des Aufrufs von m_update_varbutton() nicht sichtbar ist, werden die Paintarea-Daten in der maskeninternen Datenstruktur aktualisiert und stehen somit für das Erzeugen des Varbuttons in anderen Varianten zur Verfügung. Auch hier wird der Wert in *validmask* ausgewertet, um nur bestimmte Daten zu berücksichtigen.

Ergebnis:

Als Ergebnis liefert die Funktion TRUE bei Erfolg oder FALSE im Fehlerfall. Ein Fehler kann eine der folgenden Ursachen haben:

- Für die Maske *mskp* wurde keine Leiste mit Varbuttons definiert.
- Die als *posnr* übergebene Position liegt außerhalb des Bereichs.
- Für die Position wurde kein Varbutton definiert oder der definierte Varbutton wurde gelöscht.
- Die als *pa*->pa_props.longval2 übergebene Variantennummer liegt außerhalb des Bereichs. Der Wert wird nur geprüft, wenn er auch verwendet wird, also wenn in *validmask* der Wert PA_LONGVAL enthalten ist.
- Der Aufruf von pa_update() durch *FIX* schlug fehl.

m_wrktoinfo — Inhalte aller Felder aus Feld-Hostvariablen übernehmen

Definition:

void m_wrktoinfo(mskp, display)
mask *mskp; int display;

Beschreibung:

führt für alle Felder der Maske *mskp* wrktoinfo() aus.

Siehe auch:

wrktoinfo()

mf_copy — Feldwert aus Feld-Hostvariable eines anderen Feldes übernehmen

Definition:

void mf_copy(destmskp, destnr, srcmskp, srcnr, display)
mask *destmskp; int destnr; mask *srcmskp; int srcnr; int display;

Beschreibung:

überträgt den Wert des Feldes mit Element-Bezeichner *srcnr* der Maske *srcmskp* in das Feld mit Element-Bezeichner *destnr* der Maske *destmskp*.

Siehe auch:

fputval()

Hinweis:

entspricht fputval(fx_mfo_f(*destmskp*, *destnr*), fx_mfo_f(*srcmskp*, *srcnr*)->wrkp, *display*).

mf_erase — Feld löschen

Definition:

void mf_erase(mskp, prg_nr, display)
mask *mskp; int prg_nr; int display;

Beschreibung:

löscht das Feld mit Element-Bezeichner *prg_nr* der Maske *mskp*.

Siehe auch:

ferase()

Hinweis:

entspricht ferase(fx_mfo_f(*mskp*, *fnr*), *display*).

mfn_varbind — Feldern Feld-Hostvariablen zuordnen

Definition:

void mfn_varbind(bindptr, mskp)
mfn_bind *bindptr; mask *mskp;

Beschreibung:

bindet die Felder der Maske *mskp* über ihre Namen mit den in dem mfn_bind-Array *bindptr* aufgeführten Variablen, indem die wrkp-Komponente des Feldes auf die in *bindptr* bereitgestellte Variable umgelenkt wird. Der ursprüngliche Feldwert wird in die Variable übertragen.

Siehe auch:

fx_mfo_f(), fx_mfo_nr(), fx_prg_nr()

Hinweis:

Bei FXDTIMETYPE- und FXINVTYPE-Feldern muss der Qualifier der neuen Hostvariable gleich 0 sein oder mit dem Qualifier des Feldes übereinstimmen.

Passt die mfn_bind-Struktur nicht zur Maske, erfolgt ein Programmabbruch durch fastexit().

perf — Maske interpretieren

Definition:

Event perf(maskendatei, usr_control, usr_startup)
char *maskendatei; Event (*usr_control)(/* obj *, Event */); void (*usr_startup)(/* obj * */);

Beschreibung:

bietet das in der Beschreibungsdatei *maskendatei* hinterlegte Objekt, bei dem es sich um eine Maske handeln muss, zur Bearbeitung an. Ist die entsprechende Maske bereits geladen, so wird das geladene Exemplar benutzt, anderenfalls die Maske aus der Datei *maskendatei* zunächst geladen und die Funktion *usr_startup*, sofern ungleich (void (*)(/* obj */))0, mit dem Maskenzeiger als Argument gerufen.

Die Maske wird anschließend mittels perform(…, *usr_control*) zur Bearbeitung angeboten. Handelt es sich um eine Einzelsatz-Maske, wird hierbei eine "erweiterte Standardlogik" benutzt, die vorab eine Anwendungslogik emuliert, wie sie *FIX* für die Maske generieren würde. Dabei wird eine temporäre _modul-Struktur bereitgestellt und die Datenbankzugriffe erfolgen mittels perf_Sql().

Ergebnis:

L_PRVFIELD bei ungeeignetem Objekt,
Ergebnis des perform()-Aufrufs sonst

Siehe auch:

perf_Sql(), perfix(), perform()

Hinweis:

Eine von perf() zu bearbeitende Maske darf zum Zeitpunkt des Aufrufs nicht mehrfach geladen sein. Auch darf die Maske, wenn sie bereits geladen ist, weder eingebettet noch in Bearbeitung sein. In diesen Fällen erfolgt eine Fehlermeldung.

Die Maske sollte keine eigene Anwendungslogik besitzen und ihre Elemente sollten selbst keine Masken einbetten; in diesen Fällen erfolgt eine Warnung.

Die von perf() geladenen Masken bleiben über den Aufruf hinweg geladen.

perfGetModule — perf()-eigene _modul-Struktur ermitteln

Definition:

```
struct modul_struct *perfGetModule(mskp)
mask *mskp;
```

Beschreibung:

liefert - analog zu den generierten *maskenname*_GetModul()-Funktionen - einen Zeiger auf die _modul-Struktur, die perf() der Maske *mskp* zugeordnet hat. Der Zeiger ist nur gültig, solange der Aufruf von perf() andauert (Zeiger auf automatische Variable im Stack).

Ergebnis:

!= (struct modul_struct *)0 - von perf() zugeordnete _modul-Struktur
== (struct modul_struct *)0 - keine _modul-Struktur zugeordnet (z.B. weil kein Aufruf von perf() für die Maske anhängig)

perfix — Maske gemäß Standard-Ablauflogik interpretieren

Definition:

```
Event perfix(maskendatei)
char *maskendatei;
```

Beschreibung:

entspricht perf(*maskendatei*, (Event (*)(/* obj *, Event */))0, (void (*)(/* obj * */))0).

Ergebnis:

wie perf()

set_obj_state — Zustandsinformation einer Maske setzen

Definition:

```
void set_obj_state(objp, nr)
obj *objp; int nr;
```

Beschreibung:

veranlasst *FIX*, am rechten oberen Window-Rand des Objektes *objp* einen Zustand ("Neuaufnahme", "Korrektur", …) einzublenden. *nr* muss eines der Makros NOTHING, ADD, UPDATE, … aus `fix/obj.h` sein.

Hinweis:

objp muss eine Maske sein.

vartobuf — aktuellen Satz in Backup-Puffer retten

Definition:

BOOLEAN vartobuf(mskp, buf, buflen)
mask *mskp; char *buf; int buflen;

Beschreibung:

schreibt die Werte der Felder der Maske *mskp* an die Adresse *buf*, wobei *buf* ein vom Entwickler bereitgestellter Speicherbereich von mindestens *buflen* Bytes ist.

Ergebnis:

TRUE bei Erfolg
FALSE, wenn Speicherbereich unzureichend

Siehe auch:

buftovar()

Hinweis:

berücksichtigt auch Feld-Links.

9 Einzelsatz-Maske

m_put — Maskeninhalt in Memory Relation schreiben

Definition:

int m_put(mskp)
mask *mskp;

Beschreibung:

überträgt die augenblicklichen Werte der Feld-Hostvariablen der Einzelsatz-Maske *mskp* in das Tupel der Memory Relation.

Ergebnis:

0 - Erfolg
!= 0 - Fehler

Siehe auch:

sm_put()

10 Mehrsatz-Maske

present — Mehrsatz-Maske anzeigen †

Definition:

BOOLEAN present(smsp)
submask *smsp;

Beschreibung:

wie m_present(), aber ohne die Möglichkeit, eine Anwendungslogik anzugeben.

Ergebnis:

FALSE, wenn Anzeige-Modus mit EN (L_END) verlassen wurde
TRUE wenn Anzeige-Modus mit RT (L_NXTFIELD) verlassen wurde

Siehe auch:

m_present()

show_act_row — Satzanzeige aktualisieren

Definition:

void show_act_row(smsp)
submask *smsp;

Beschreibung:

Sofern die Mehrsatz-Maske *smsp* am Bildschirm dargestellt und die Satzanzeige nicht unterdrückt ist, wird auf dem oberen Rand der Mehrsatz-Maske hinter der evtl. vorhandenen Überschrift der Text “xx von yy” ausgegeben, wobei xx die Nummer des aktuellen Satzes und yy die Anzahl der insgesamt vorhandenen Sätze ist.

Hinweis:

Die Funktion hat einen wichtigen Nebeneffekt: Enthielt die Mehrsatz-Maske bei der vorangegangenen Darstellung keine Sätze, ist nun aber nicht mehr leer, so wird der erste Satz zum aktuellen Satz und der Maskeninhalt neu dargestellt.

sm_act_row — Position des aktuellen Satzes ermitteln

Definition:

int sm_act_row(smsp)
submask *smsp;

Beschreibung:

liefert die Position des aktuellen Satzes in der Satzmenge der Mehrsatz-Maske *smsp*.

Ergebnis:

> 0 - Position
0 - Mehrsatz-Maske leer
< 0 - Fehler

Siehe auch:

sm_anz_row(), mr_act_row()

sm_add — Maskeninhalt als Satz an die Memory Relation anhängen

Definition:

int sm_add(smsp, actual_values, become_actual)
submask *smsp; BOOLEAN actual_values; BOOLEAN become_actual;

Beschreibung:

fügt in der Mehrsatz-Maske *smsp* einen Satz an letzter Position an, und zwar mit den aktuellen Werten der Feld-Hostvariablen, wenn *actual_values* != FALSE, einen leeren Satz sonst. Der angefügte Satz wird zum aktuellen Satz, wenn *become_actual* != FALSE ist.

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

sm_insert(), sm_delete()

Hinweis:

Ein angelegter Leersatz enthält in Feldern den Defaultwert, in COPY-Feldern den letztmaligen Wert.

sm_anz_row — Anzahl der Sätze ermitteln

Definition:

int sm_anz_row(smsp)
submask *smsp;

Beschreibung:

liefert die Anzahl der Sätze in der Satzmenge der Mehrsatz-Maske *smsp*.

Ergebnis:

≥ 0 - Anzahl
< 0 - Fehler

Siehe auch:

sm_act_row(), mr_num_of_rows()

sm_browse — in Mehrsatz-Maske blättern

Definition:

int sm_browse(smsp, back)
submask *smsp; int back;

Beschreibung:

rollt die Mehrsatz-Maske *smsp* um einen Satz zurück (*back* != 0) oder vorwärts (*back* == 0), d.h. besitzt der aktuelle Satz einen Vorgänger/Nachfolger, so wird dieser zum aktuellen Satz (SUCCESS), sonst bleibt der aktuelle Satz erhalten (DBR_END/DBR_START).

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

sm_cleanup — Satzmenge aufräumen

Definition:

```
void sm_cleanup(smsp)
submask *smsp;
```

Beschreibung:

ruft die cleanup-Funktion der Mehrsatz-Maske *smsp* auf, wobei ihr *smsp* als Argument übergeben wird.

Falls die cleanup-Funktion nicht mittels sm_install_cleanup() umgesetzt wurde, löscht eine interne cleanup-Funktion alle Leersätze der Mehrsatz-Maske; als Leersatz gilt ein Satz, der in allen nicht gelinkten Feldern den Defaultwert enthält. Gehört der aktuelle Satz nicht dazu, so bleibt er aktueller Satz, andernfalls wird der erste verbleibende Satz zum aktuellen Satz bzw. alle Felder werden auf den Defaultwert gesetzt.

Der Maskeninhalt wird anschließend neu dargestellt.

Siehe auch:

perform(), sm_install_cleanup()

Hinweis:

FIX ruft sm_cleanup() automatisch am Ende der Abarbeitung einer Mehrsatz-Maske (perform()) auf.

sm_colno_to_fnr — zu einer Spalte gehörendes Maskenelement ermitteln †

Definition:

```
int sm_colno_to_fnr(smsp, colno)
submask *smsp; int colno;
```

Beschreibung:

liefert die Element-Position des Elements der Mehrsatz-Maske *smsp*, zu dem die *colno*-te Spalte ($1 \leq colno \leq$ Anzahl Spalten) der Memory Relation gehört.

Ergebnis:

≥ 0 - Element-Position
-2 - unzulässiges Argument
-1 - kein Maskenelement gefunden

Siehe auch:

m_colno_to_fnr(), sm_fnr_to_colno()

Hinweis:

Statt sm_colno_to_fnr() sollte m_colno_to_fnr() verwendet werden.

sm_delete — aktuellen Satz löschen

Definition:

```
int sm_delete(smsp)
submask *smsp;
```

Beschreibung:

löscht den aktuellen Satz der Mehrsatz-Maske *smsp*. Handelt es sich um den einzigen Satz, nehmen alle Felder ihren Defaultwert an, anderenfalls wird, wenn ein weiterer Satz folgt, dieser zum aktuellen Satz, sonst der vorausgehende.

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

sm_empty()

Hinweis:

Die Feldwerte des aktuellen Satzes werden zuvor auf das Clipboard kopiert.

sm_empty — alle Sätze löschen

Definition:

void sm_empty(smsp)
submask *smsp;

Beschreibung:

löscht alle Sätze der Mehrsatz-Maske *smsp*; alle Felder nehmen ihren Defaultwert an.

Der Maskeninhalt wird anschließend neu dargestellt.

Siehe auch:

sm_delete()

Hinweis:

Der Inhalt des Clipboards ist anschließend undefiniert.

sm_exchange — Satzmenge austauschen

Definition:

int sm_exchange(smsp, newmrp, oldmrp_adr)
submask *smsp; MemRelType newmrp; MemRelType *oldmrp_adr;

Beschreibung:

ersetzt die augenblickliche Satzmenge der Mehrsatz-Maske *smsp* durch die Satzmenge *newmrp*. Ist *newmrp* == (MemRelType)0, wird eine leere Memory Relation angelegt, anderenfalls müssen die Sätze in *newmrp* hinsichtlich Spaltenzahl, -typ und -länge bzw. -Qualifier zur Mehrsatz-Maske passen.

kein Fehler, *oldmrp_adr* == 0:
die ursprüngliche Satzmenge wird freigegeben.

kein Fehler, *oldmrp_adr* != 0:
die ursprüngliche Satzmenge wird nicht freigegeben, an der Adresse *oldmrp_adr* wird ihr Deskriptor hinterlegt.

Fehler, *oldmrp_adr* == 0:
die Satzmenge bleibt unverändert.

Fehler, *oldmrp_adr* != 0:
die Satzmenge bleibt unverändert, an der Adresse *oldmrp_adr* wird (MemRelType)0 hinterlegt.

Findet ein Austausch statt, wird der erste Satz zum aktuellen Satz der Mehrsatz-Maske; enthält die neue Memory Relation keine Sätze, nehmen alle Felder ihren Defaultwert an. Das aktuelle Feld wird nicht gewechselt.

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

sm_find — Satz auswählen

Definition:

int sm_find(smsp, indikator, value)
submask *smsp; int indikator; char *value;

Beschreibung:

sucht den Satz entsprechend *indikator* in der Mehrsatz-Maske *smsp* und transportiert den Inhalt in die korrespondierenden Feld-Hostvariablen. *indikator* muss eines der unten aufgeführten Makros sein, *value* ein Zeiger: im Falle FIRST, LAST, NEXT, PREVIOUS usw. wird er ignoriert, im Falle EQUAL, GTEQ und GREATER muss er auf den Vergleichswert zeigen, im Falle INDEXED (finde den i-ten Satz, i ≥ 1), RELATIVE (finde den i-ten Satz vom aktuellen aus gesehen, vorwärts oder rückwärts (negatives i)) oder BY_ROWID (finde den Satz mit ROWID i) enthält er den in einen Zeiger umgewandelten Wert i, also (char *)i. Der so bestimmte Satz wird zum aktuellen Satz.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

sm_hold(), sm_restore(), sm_selfield()

Hinweis:

EQUAL, GTEQ und GREATER setzen die Auswahl einer Spalte mittels sm_selfield() voraus.

Ist ein Vergleich von Werten erforderlich, müssen bei den Typen FXDTIMETYPE und FXINVTYPE die Qualifier von Spalte und Vergleichswert im Sinne von fxvalcmp() kompatibel sein. Beim Typ FXGRAPHICSTYPE müssen die Länge von Spalte und Vergleichswert übereinstimmen.

Wird der Satz nicht gefunden, bleibt der bisherige Satz aktueller Satz. Ein Transport in die Feld-Hostvariablen erfolgt in diesem Fall nicht.

Eine Synchronisation der Darstellung am Bildschirm findet erst bei einer Neuausgabe des aktuellen Satzes bzw. der Maske statt.

sm_find_extended — Satz auswählen

Definition:

int sm_find_extended(smsp, indicator, acceptor, usr_data)
submask *smsp; int indicator; BOOLEAN (*acceptor)(/* MemRelType, TupelType, char * */); char *usr_data;

Beschreibung:

sucht mittels mr_get_tupel_variadic() den Satz in der Mehrsatz-Maske *smsp*. Wird ein Satz gefunden, wird sein Inhalt in die korrespondierenden Feld-Hostvariablen transportiert.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

mr_get_tupel_variadic(), mr_fetch_row()

sm_fnr_to_colno — zu einem Maskenelement gehörende Spalte ermitteln †

Definition:

int sm_fnr_to_colno(smsp, fnr)
submask *smsp; int fnr;

Beschreibung:

liefert die Nummer der Spalte in der Memory Relation der Mehrsatz-Maske *smsp*, die zu dem Maskenelement an Element-Position *fnr* (0 ≤ *fnr* < Anzahl Maskenelemente) gehört.

Ergebnis:

> 0 - Nummer der Spalte
-2 - unzulässiges Argument
-1 - keine entsprechende Spalte vorhanden

Siehe auch:

m_fnr_to_colno(), sm_colno_to_fnr()

Hinweis:

Statt sm_fnr_to_colno() sollte m_fnr_to_colno() verwendet werden.

sm_forall — alle Sätze bearbeiten

Definition:

int sm_forall(smsp, fct, update, display)
submask *smsp; void (*fct)(/* submask *, int */); int update; int display;

Beschreibung:

führt für alle Sätze der Mehrsatz-Maske *smsp*, beginnend beim ersten, (**fct*)(*smsp*, i) aus, wobei i (i ≥ 0) mit jedem Aufruf erhöht wird. Beim Aufruf mit Argument i ist der (i + 1)-te Satz aktueller Satz, d.h. stehen dessen Werte in den korrespondierenden Feld-Hostvariablen. Ist *update* != 0, werden die Feld-Hostvariablen nach Ausführung von *fct*() in das aktuelle Tupel der Memory Relation zurückgeschrieben.

Ist *display* != 0, so wird der Maskeninhalt anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Hinweis:

update == 0 bedeutet nicht, dass der Inhalt des aktuellen Tupels der Memory Relation nicht bei Verwendung von *FIX*-Funktionen innerhalb von *fct* überschrieben wird.

Innerhalb von *fct* darf weder der aktuelle Satz gewechselt werden noch dürfen Sätze vertauscht, gelöscht oder hinzugefügt werden.

Die Funktion versucht, im Anschluss wieder auf den Satz zu positionieren, der beim Aufruf aktuell war.

sm_hold — Position des aktuellen Satzes merken

Definition:

int sm_hold(smsp)
submask *smsp;

Beschreibung:

vermerkt in der Mehrsatz-Maske *smsp* die Position des aktuellen Satzes für ein späteres sm_restore().

Ergebnis:

≥ 0 - Position des gemerkten Satzes
0 - *smsp* keine Mehrsatz-Maske oder keine Sätze vorhanden

Siehe auch:

sm_restore()

sm_insert — Satz einfügen

Definition:

int sm_insert(smsp, actual_values)
submask *smsp; BOOLEAN actual_values;

Beschreibung:

fügt vor dem aktuellen Satz einen Satz in die Mehrsatz-Maske *smsp* ein, und zwar mit den aktuellen Werten der Feld-Hostvariablen, wenn *actual_values* != FALSE, sonst einen leeren Satz. Bei Erfolg wird der neue Satz zum aktuellen Satz.

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

sm_add(), sm_delete()

Hinweis:

Ein angelegter Leersatz enthält in allen nicht gelinkten Feldern den Defaultwert, in COPY-Feldern den letztmaligen Wert.

sm_install_cleanup — cleanup-Funktion umsetzen

Definition:

void (*sm_install_cleanup(smsp, fct))(/* submask * */)
submask *smsp; void (*fct)(/* submask * */);

Beschreibung:

ersetzt die vorhandene (ggf. Default-)Cleanup-Funktion der Mehrsatz-Maske *smsp* durch die Funktion *fct*. Falls *fct* == (void (*)(/* submask * */))0, wird die vorhandene Cleanup-Funktion beseitigt.

Ergebnis:

Zeiger auf die zuvor installierte Cleanup-Funktion

Siehe auch:

sm_cleanup()

sm_mark_new — Statusinformation des aktuellen Satzes ändern

Definition:

int sm_mark_new(smsp)
submask *smsp;

Beschreibung:

markiert den aktuellen Satz der Mehrsatz-Maske *smsp* als “von Hand” angelegt, d.h. nicht durch eine Standardmethode zur Datenbeschaffung gelesen.

Ergebnis:

0 - Satz markiert
< 0 - Fehler

Siehe auch:

sm_mark_old(), sm_read_rows()

sm_mark_old — Statusinformation des aktuellen Satzes ändern

Definition:

int sm_mark_old(smsp)
submask *smsp;

Beschreibung:

markiert den aktuellen Satz der Mehrsatz-Maske *smsp* als “alt”, d.h. durch eine Standardmethode zur Datenbeschaffung gelesen.

Ergebnis:

0 - Satz markiert
< 0 - Fehler

Siehe auch:

sm_old(), sm_mark_new(), sm_read_rows()

sm_old — Statusinformation des aktuellen Satzes auswerten

Definition:

BOOLEAN sm_old(smsp)
submask *smsp;

Beschreibung:

gibt Auskunft über die Entstehung des aktuellen Satzes der Mehrsatz-Maske *smsp*.

Ergebnis:

FALSE - Satz wurde “von Hand” (durch den Anwender) angelegt
TRUE - Satz wurde über eine Standardmethode zur Datenbeschaffung gelesen

Siehe auch:

sm_mark_old(), sm_mark_new(), sm_read_rows()

sm_paste — aktuellen Satz mit dem Inhalt des Clipboard überschreiben

Definition:

int sm_paste(smsp)
submask *smsp;

Beschreibung:

überschreibt den aktuellen Satz der Maske *smsp* mit dem Satz, der sich auf dem Clipboard der Maske befindet.

Ergebnis:

0 - Erfolg
-1 - Mehrsatz-Maske leer

-2 - kein Satz auf dem Clipboard
< -2 - interner Fehler

sm_put — Maskeninhalt in Memory Relation zurückschreiben

Definition:

int sm_put(smsp)
submask *smsp;

Beschreibung:

überträgt die augenblicklichen Werte der Feld-Hostvariablen der Maske *smsp* in das aktuelle Tupel der Memory Relation.

Ergebnis:

0 - Erfolg
!= 0 - Fehler

Siehe auch:

m_put()

sm_read_rows — Sätze lesen †

Definition:

int sm_read_rows(smsp, sqcursornr, sqfct, flush)
submask *smsp; int sqcursornr; long (*sqfct)(/* int, int, char * */); BOOLEAN flush;

Beschreibung:

ruft zunächst die Funktion *sqfct* mit dem Argumenten *sqcursornr* und OPEN und dann wiederholt mit den Argumenten *sqcursornr* und FETCH, solange der Aufruf SUCCESS oder SQL_OMIT zurückgibt. Nach jedem erfolgreichen FETCH-Aufruf wird an die Memory Relation der Mehrsatz-Maske *smsp* ein neues Tupel angefügt, dessen Spaltenwerte den zugehörigen Feld-Hostvariablen entnommen werden. Das Tupel erhält den Status “alt”. Abschließend wird die Funktion *sqfct* mit den Argumenten *sqcursornr* und CLOSE aufgerufen.

Bei einem Fehler (außer SQLNOTFOUND oder SQL_OMIT) wird die Funktion abgebrochen.

Ist *flush* == TRUE, werden alle in der Mehrsatz-Maske *smsp* vorhandenen Sätze vor dem OPEN-Aufruf zunächst gelöscht.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode von *sqfct* oder Tupelzugriff

Siehe auch:

sm_write_rows(), sm_old(), sm_mark_old()

Hinweis:

In mit Versionen bis 2.8 generierten Programmen wurde diese Funktion, versorgt mit einer geeigneten Routine, benutzt, um die Satzmenge einer Mehrsatz-Maske aus der Datenbank zu lesen oder zu erweitern (Makros sm_read(), sm_addread()).

sm_restore — Satz an vermerkter Position zum aktuellen Satz machen

Definition:

int sm_restore(smsp)
submask *smsp;

Beschreibung:

macht den Satz zum aktuellen Satz, der die Position einnimmt, die bei einem vorangegangenen sm_hold() in der Mehrsatz-Maske *smsp* vermerkt wurde.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

sm_hold(), sm_find()

Hinweis:

Eine Synchronisation der Darstellung am Bildschirm findet erst bei einer Neuausgabe des aktuellen Satzes bzw. der Maske statt.

sm_rewind — ersten Satz zum aktuellen Satz machen

Definition:

```
int sm_rewind(smsp)
submask *smsp;
```

Beschreibung:

macht den ersten Satz der Mehrsatz-Maske *smsp* zum aktuellen Satz.

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

sm_wind()

sm_rowid — ROWID des aktuellen Tupels abfragen

Definition:

```
long sm_rowid(smsp)
submask *smsp;
```

Beschreibung:

liefert die ROWID des aktuellen Tupels der Mehrsatz-Maske *smsp*.

Ergebnis:

> 0 - ROWID
0 - Mehrsatz-Maske leer
< 0 - Fehler (z.B. keine Memory Relation vorhanden)

Siehe auch:

mr_rowid()

sm_selfield — Satzkomponente auszeichnen

Definition:

```
int sm_selfield(smsp, fnr)
submask *smsp; int fnr;
```

Beschreibung:

zeichnet das Element an Element-Position *fnr* der Mehrsatz-Maske *smsp*, bei dem es sich weder um einen Feld-Link noch um eine eingebettete Maske handeln darf, als Such- oder Sortierfeld aus.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

sm_sort()

sm_sort — Sätze einer Mehrsatz-Maske sortieren

Definition:

```
int sm_sort(smsp)
submask *smsp;
```

Beschreibung:

sortiert die Sätze der Mehrsatz-Maske *smsp* gemäß fxvalcmp() bzgl. der Spalte, die dem Feld entspricht, das zuvor durch sm_selfield() ausgezeichnet wurde. Nach dem Sortieren ist der erste Satz aktueller Satz.

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

sm_selfield(), fxvalcmp(), mr_qsort()

sm_sum — Werte einer Satzkomponente summieren

Definition:

```
int sm_sum(smsp, fnr, result)
submask *smsp; int fnr; char *result;
```

Beschreibung:

summiert die Werte der Spalte, die zu dem Element der Mehrsatz-Maske *smsp* an Element-Position *fnr* gehört, und speichert das Ergebnis an der Adresse *result*. Es wird angenommen, dass *result* auf eine Variable des gleichen Datentyps wie Feld *fnr* zeigt.

Ergebnis:

0 - Erfolg
< 0 - Fehler
> 0 - es waren entsprechend viele NULL-Werte beteiligt, die ignoriert worden sind

sm_wind — auf letztem Satz positionieren

Definition:

```
int sm_wind(smsp)
submask *smsp;
```

Beschreibung:

macht den letzten Satz der Mehrsatz-Maske *smsp* zum aktuellen Satz.

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

sm_rewind()

sm_write_rows — Sätze schreiben †

Definition:

```
int sm_write_rows(smsp, sqcursornr, sqfct, keep)
submask *smsp; int sqcursornr; long (*sqfct)(/* int, int, char * */); BOOLEAN keep;
```

Beschreibung:

ruft zunächst die Funktion *sqfct* mit den Argumenten *sqcursornr* und OPEN und dann wiederholt mit den Argumenten *sqcursornr* und WFETCH, solange der Aufruf SUCCESS oder SQL_OMIT zurückgibt. Nach dem i-ten WFETCH-Aufruf werden, sofern die Memory Relation der Mehrsatz-Maske *smsp* ein i-tes Tupel enthält, dessen Spaltenwerte in die zugehörigen Variablen transportiert und die Funktion *sqfct* mit den Argumenten sqcursornr und UPDATE gerufen, andernfalls wird die Funktion *sqfct* mit den Argumenten *sqcursornr* und DELETE gerufen.

Bricht die Schleife mit dem Fehler SQLNOTFOUND ab, sind aber noch weitere Tupel in der Memory Relation der Mehrsatz-Maske vorhanden, so werden für jedes die Spaltenwerte in die zugehörigen Variablen transportiert und die Funktion *sqfct* mit den Argumenten *sqcursornr* und INSERT gerufen.

Abschließend wird die Funktion *sqfct* mit den Argumenten *sqcursornr* und CLOSE aufgerufen.

Bei einem Fehler (außer SQLNOTFOUND oder SQL_OMIT) wird die Funktion abgebrochen.

Ist *keep* == FALSE, wird ein Tupel unmittelbar nach der Bearbeitung gelöscht.

Ergebnis:

SUCCESS bei Erfolg
Fehlercode von *sqfct* oder Tupelzugriff bei Fehler

Siehe auch:

sm_read_rows()

Hinweis:

Innerhalb von *sqfct* darf weder der aktuelle Satz gewechselt werden noch dürfen Sätze vertauscht, gelöscht oder hinzugefügt werden.

Eine Synchronisation der Darstellung am Bildschirm findet erst bei einer Neuausgabe des aktuellen Satzes bzw. der Maske statt.

In mit Versionen bis 2.8 generierten Programmen wurde diese Funktion, versorgt mit einer geeigneten Routine, benutzt, um die Sätze einer Mehrsatz-Maske wieder in die Datenbank zu schreiben (Makros sm_write(), sm_keepwrite()).

11 Tabellenmaske

t_backbrowse — eine Seite rückwärts blättern

Definition:

int t_backbrowse(t_p)
table *t_p;

Beschreibung:

blättert die Tabellenmaske *t_p*, falls möglich, um eine Seite zurück, d.h. (angenommen, dass die Tabellenmaske n Sätze darstellen kann):

1. Gehen den augenblicklich dargestellten Sätzen mindestens n Sätze voraus, so werden diese dargestellt, anderenfalls die ersten n (oder weniger) Sätze.
2. Besitzt der aktuelle Satz einen n-ten Vorgänger, so wird dieser zum aktuellen Satz (0), anderenfalls der erste Satz (DBR_START).

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

t_forwbrowse()

Hinweis:

Enthält die Maske keine Sätze, ist die Funktion wirkungslos (DBR_START).

t_bring — Satz sichtbar machen und darauf positionieren

Definition:

int t_bring(t_p, tup_nr)
table *t_p; int tup_nr;

Beschreibung:

bringt den *tup_nr*-ten Satz der Tabellenmaske *t_p* in den sichtbaren Bereich. Ist der Satz bereits sichtbar, so wird er nur zum aktuellen Satz, andernfalls wird versucht, den Satz möglichst an die Position zu bringen, die der zuvor aktuelle Satz einnahm.

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
DBR_START - *tup_nr* < 1
DBR_END - *tup_nr* > Anzahl Sätze
DBR_NODATA - Maske enthält keine Sätze

Siehe auch:

t_bring_to(), t_max_in_window()

Hinweis:

entspricht t_bring_to(*t_p*, *tup_nr*, 0).

t_bring_to — Satz positionieren

Definition:

int t_bring_to(t_p, tup_nr, row_nr)
table *t_p; int tup_nr, row_nr;

Beschreibung:

bringt den *tup_nr*-ten Satz der Tabellenmaske *t_p* an die *row_nr*-te Position und macht ihn zum aktuellen Satz. *row_nr* == 0 wird verschieden interpretiert: ist der Satz bereits sichtbar, so wird er nur zum aktuellen Satz, andernfalls wird versucht, den Satz möglichst an die Position zu bringen, die den bislang aktuellen Satz enthält.

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
DBR_START - *tup_nr* < 1
DBR_END - *tup_nr* > Anzahl Sätze
DBR_NODATA - Maske enthält keine Sätze

Siehe auch:

t_bring(), t_max_in_window()

Hinweis:

Ist *row_nr* ungültig oder nicht möglich (eine Tabellenmaske wird immer von oben gefüllt), so wird *row_nr* auf einen vernünftigen Wert korrigiert.

t_discriminate — Feld für Zeilentypbestimmung definieren

Definition:

BOOLEAN t_discriminate(t_p, fnr, fct)
table *t_p; int fnr; int (*fct)(/* field */);

Beschreibung:

zeichnet, falls es sich bei *t_p* um eine Tabellenmaske mit Zeilentypen handelt, das Element an Element-Position *fnr* als dasjenige Element aus, das den Zeilentyp bestimmt. Die Funktion *fct* muss - bei einem Feld i. Allg. in Abhängigkeit von dessen Wert -, wird sie auf die Adresse des Elements angewendet, die Nummer des passenden Zeilentyps liefern (0 entspricht Default).

Ergebnis:

TRUE bei Erfolg
FALSE sonst

t_forwbrowse — eine Seite vorwärts blättern

Definition:

int t_forwbrowse(t_p)
table *t_p;

Beschreibung:

blättert die Tabellenmaske *t_p*, falls möglich, um eine Seite nach vorn, d.h. (angenommen, dass die Tabellenmaske n Sätze darstellen kann):

1. Folgen auf die augenblicklich dargestellten Sätze n oder weniger weitere Sätze, so werden diese dargestellt.
2. Besitzt der aktuelle Satz einen n-ten Nachfolger, so wird dieser zum aktuellen Satz (0), anderenfalls der letzte Satz (DBR_END).

Der Maskeninhalt wird anschließend neu dargestellt.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

t_backbrowse()

Hinweis:

Enthält die Maske keine Sätze, ist die Funktion wirkungslos (DBR_END).

t_go_into — in Tabellenmaske positionieren

Definition:

int t_go_into(t_p, row_nr)
table *t_p; int row_nr;

Beschreibung:

macht in der Tabellenmaske *t_p* den Satz an *row_nr*-ter Position (1 ≤ *row_nr* ≤ Anzahl dargestellter Sätze) zum aktuellen Satz.

Ergebnis:

0 - Erfolg
DBR_START - *row_nr* < 1
DBR_END - *row_nr* > Anzahl dargestellter Sätze

Siehe auch:

t_max_in_window()

t_max_in_window — Maximalzahl der anzeigbaren Sätze einer Tabellenmaske bestimmen

Definition:

int t_max_in_window(t_p)
table *t_p;

Beschreibung:

ermittelt die Anzahl der Sätze, die in der Tabellenmaske *t_p* gleichzeitig dargestellt werden können.

Ergebnis:

Anzahl der darstellbaren Sätze

Siehe auch:

t_pos_in_window()

t_move - Tabelle mit Zeilenvarianten verschieben

Definition:

void t_move(objp, nz, ns)
obj *objp; int nz; int ns;

Beschreibung:

Verschiebt die linke obere Ecke einer Tabelle objp und aller Zeilentypvarianten an eine neue Position nz (Zeile),ns (Spalte).

Hinweis:

Die Funktion führt keinerlei Plausibilitätsprüfung bezüglich der Koordinaten durch.

t_pos_in_window — Anzeigeposition in Tabellenmaske bestimmen

Definition:

int t_pos_in_window(t_p)
table *t_p;

Beschreibung:

ermittelt, an welcher Position, d.h. als wievielter Satz der aktuelle Satz der Tabellenmaske *t_p* im Maskenwindow dargestellt wird.

Ergebnis:

> 0 - Position in der Tabellenmaske
== 0 - Maske enthält keine Sätze

Siehe auch:

t_max_in_window()

Hinweis:

Der aktuelle Satz sollte zu den angezeigten Sätzen gehören.

t_variant_forall — Zeilenvariantenumschaltung auf alle Varianten

Definition:

BOOLEAN t_variant_forall(t_p, fct);
table *t_p; void (*fct)(/* table *t_p, int i */);

Beschreibung:

Die Funktion führt für alle Zeilentypen der Tabelle t_p die Funktion fct aus. Die Funktion beginnt beim ersten Zeilentyp und schaltet für jeden Aufruf von fct auf den nächsten Zeilentyp. Der Wert von i beginnt bei 0 und wird mit jedem Aufruf von fct um 1 erhöht.

Ergebnis:

Als Rückgabewert liefert die Funktion TRUE bei Erfolg und FALSE wenn der übergebene Maskenpointer keine Tabelle ist, die Tabelle keine Zeilentypen hat oder es sich bei der Tabelle nicht um die erste Maskenvariante handelt.

Hinweis:

Vor der unmittelbaren Rückkehr der Funktion wird auf den zu Beginn aktiven Zeilentyp zurückgeschaltet. Eine Änderung des Wertes des Umschaltfeldes wird nicht berücksichtigt und ist innerhalb von fct nicht zu empfehlen, da es dadurch zu inkonsistenten Zuständen kommen kann.

12 Selo

loadselo — Selo laden

Definition:

selo *loadselo(name)
char *name;

Beschreibung:

lädt das in der Beschreibungsdatei *name* definierte Selo.

Ergebnis:

Zeiger auf die Interndarstellung bei Erfolg
(selo *)0 bei Fehler

Siehe auch:

fx_bindfield()

Hinweis:

Beim Laden erkannte Fehler werden gemeldet.

Sofern das Selo nicht mit einem Feld verknüpft ist, kann es durch o_free() wieder freigegeben werden.

enableSeloAsTable - Tabellendarstellung für Selos aktivieren

Definition:

BOOLEAN enableSeloAsTable(delim, id1, id2)
unsigned char *delim; long id1; long id2;

Beschreibung:

schaltet die Darstellung von Selozellen auf Tabellenzellen um. Dazu ist in dem Bereich, auf den *delim* zeigt, eine Zeichenkette abzulegen, die die Semigrafikzeichen für die Feldbegrenzer einer Selozelle enthält. Die Zeichenkette muss genau 6 Zeichen lang sein. Die ersten beiden Zeichen werden für eine Zelle in der ersten Spalte verwendet, die letzen beiden für eine Zelle in der letzten Spalte und die mittleren beiden für die Zellen in den mittleren Spalten. In *id1* und *id2* sind die Werte zur Darstellung von Tabellenköpfen abzulegen. Diese werden durch Paintareas dargestellt, wobei die Werte *id1* und *id2* als Wert für den Parameter longval1 verwendet werden. Je nachdem, ob es sich um eine der ersten Spalten handelt oder um die letzte Spalte, die anderes gezeichnet werden muss, wird *id1* oder *id2* verwendet. *FIX/Win* geht davon aus, dass hier die Werte PA_TABLEHEADER_FIRST und PA_TABLEHEADER_LAST verwendet werden. Wenn für *FIX/Win* eine spezielle Funktion zum Zeichnen von Überschriften für Tabellenspalten hinterlegt wird, dann kann es notwendig sein, hier andere Werte zu verwenden.

Die Funktion liefert als Ergebnis TRUE, wenn sie erfolgreich war. Wenn FALSE zurückgeliefert wird, dann wird die Anwendung nicht mit *FIX/Win* bedient oder die Länge von *delim* ist ungleich 6.

Um die Tabellendarstellung zu deaktivieren, ist die gleiche Funktion nochmals aufzurufen. Mindestens einer der Parameter muss zum Deaktivieren den Wert 0 enthalten.

Hinweis:

Die Funktion aktiviert die Tabellendarstellung nur, wenn die Anwendung mit *FIX/Win* bedient wird.

fxse_init — fxse-API reservieren

Definition:

int fxse_init(sep, parammask_addr, outmrp_addr, fxse_cb_pointer_addr)
selo *sep; mask **parammask_addr; MemRelType *outmrp_addr; struct fxse_cb **fxse_cb_pointer_addr;

Beschreibung:

reserviert das fxse-API für das Selo *sep*. Im Erfolgsfall wechselt das API vom Zustand “unbenutzt” in den Zustand “suchbereit”; in ∗*parammask_addr*, ∗*outmrp_addr* und ∗*fxse_cb_pointer_addr* sind dann Zeiger auf eine vom API bereitgestellte Einzelsatz-Maske zur Hinterlegung von Startwerten, eine Memory Relation zur Aufnahme der Treffersätze und eine Info-Struktur hinterlegt.

Ergebnis:

0 - Erfolg
FXSE_ILLEGAL_ARGUMENT - fxse-API bereits reserviert oder Selo ungeeignet
FXSE_SETUP_FAILED - Initialisierung fehlgeschlagen
FXSE_ALLOC_FAILED - Speicheranforderung fehlgeschlagen

Siehe auch:

fxse_finalize()

Hinweis:

Während des Gebrauchs des fxse-API kann keine Selobearbeitung mittels perform() stattfinden.

Die Kontrollstruktur struct fxse_cb ist und die Fehlercodes des fxse-API sind in der Header-Datei `fxse.h` deklariert.

fxse_start_search — mittels fxse-API in der Datenbank suchen

Definition:

int fxse_start_search(sep, ev)
selo ∗sep; Event ev;

Beschreibung:

Bei *sep* muss es sich um das Selo handeln, für das das fxse-API reserviert und in den Zustand “suchbereit” versetzt wurde. *ev* muss eines der Events K_f8 oder K_RT sein: bei K_f8 wird eine **restrict**-Angabe im Selo ignoriert, bei K_RT berücksichtigt. *FIX* führt die Datenbeschaffungsvorschrift des Selos aus und versucht, einen ersten Treffersatz zu lesen.

Ergebnis:

0 - Erfolg
FXSE_ILLEGAL_ARGUMENT - unzulässiges Argument, unzulässiger API-Zustand
FXSE_QUERY_FAILED - Fehler bei der Datenbanksuche
FXSE_NODATA - keine Treffer gefunden
FXSE_UNEXPECTED_ERROR - sonstiger Fehler

Siehe auch:

fxse_init()

Hinweis:

In den drei letztgenannten Fällen nimmt das API den Zustand “Suche erfolgt” an. Bei Erfolg verwaltet das fxse-API intern einen geöffneten Cursor.

fxse_supply — mittels fxse-API Treffersätze bereitstellen

Definition:

int fxse_supply(sep, rowsneeded)
selo ∗sep; int rowsneeded;

Beschreibung:

Bei *sep* muss es sich um das Selo handeln, für das das fxse-API den Zustand “Suche erfolgt” oder “Auswahl erfolgt” angenommen hat. Die Datenbanksuche muss erfolgreich gewesen sein (d.h. der interne Cursor des fxse-API muss ge-

öffnet sein). *FIX* versucht, die ersten *rowsneeded* Treffersätze zu lesen und in die hierfür bereitgestellte Memory Relation zu übertragen.

Ergebnis:

0 - Erfolg
FXSE_ILLEGAL_ARGUMENT - unzulässiges Argument, unzulässiger API-Zustand, Datenbanksuche nicht erfolgreich
FXSE_FETCH_FAILED - Fehler beim Lesen aus der Datenbank
FXSE_NODATA - rowsneeded > #Treffersätze

Siehe auch:

fxse_init(), fxse_start_search()

Hinweis:

Wird FXSE_NODATA zurückgegeben, enthält die Memory Relation die gesamte Ergebnismenge der Suche.

fxse_choose — mittels fxse-API im Selo auswählen

Definition:

int fxse_choose(sep, row, export)
selo *sep; int row; BOOLEAN export;

Beschreibung:

Bei *sep* muss es sich um das Selo handeln, für das das fxse-API den Zustand “Suche erfolgt” oder “Auswahl erfolgt” angenommen hat. Die Datenbanksuche muss erfolgreich verlaufen sein und es müssen zumindest die ersten *row* Treffersätze gelesen worden sein. *FIX* übernimmt den *row*-ten Treffersatz aus der Memory Relation in den Auswahlpuffer des Selos und das API wechselt in den Zustand “Auswahl erfolgt”. Ist *export* ungleich FALSE, werden zusätzlich die im Selo hinterlegten Vorschriften zur Wertübernahme abgearbeitet.

Ergebnis:

0 - Erfolg
FXSE_ILLEGAL_ARGUMENT - unzulässiges Argument, unzulässiger API-Zustand, Datenbanksuche nicht erfolgreich
FXSE_GETTUPEL_FAILED - kein *row*-ter Treffersatz
FXSE_NO_EXPORTS - keine Wertübernahmen erfolgt (nur bei *export* ungleich FALSE)
FXSE_EXPORT_FAILED - Wertübernahme gescheitert (nur bei *export* ungleich FALSE)

Siehe auch:

fxse_init(), fxse_start_search(), fxse_supply(), fx_selo_fld_adr()

fxse_finalize — fxse-API freigeben

Definition:

int fxse_finalize(sep)
selo *sep;

Beschreibung:

Bei *sep* muss es sich um das Selo handeln, für das das fxse-API reserviert wurde. Das API gibt die für den Gebrauch bereitgestellten Datenstrukturen (Info-Struktur etc.) frei und wird durch Annahme des Zustands “unbenutzt” wieder verfügbar.

Ergebnis:

0 - Erfolg
FXSE_ILLEGAL_ARGUMENT - unzulässiges Argument, API nicht reserviert

Siehe auch:

fxse_init()

13 Choice

assign_mrel — einer Choice eine Datenmenge zuweisen

Definition:

BOOLEAN assign_mrel(chp, mrp)
choice *chp; MemRelType mrp;

Beschreibung:

bindet die Memory Relation *mrp* als Datenmenge an die Choice *chp*. Besitzt die Memory Relation keine ausgezeichnete Spalte, wird die gleiche Spalte wie in der bisherigen Datenmenge bzw., wenn dies nicht möglich ist, die erste Spalte ausgezeichnet.

Ergebnis:

TRUE bei Erfolg
FALSE bei unzulässigem Argument

Siehe auch:

ch_exchange()

Hinweis:

Bei Choices mit Datenbeschaffungsangabe ruft assign_mrel() die Funktion ch_exchange() auf.

Die Markierung der Tupel bleibt erhalten.

Der Entwickler ist selbst dafür verantwortlich, dass eine vorhandene, nicht länger benötigte Memory Relation ggf. freigegeben wird.

Ist die Choice am Bildschirm dargestellt, muss ihre Darstellung anschließend neu aufgebaut werden.

Die Choice sollte zum Zeitpunkt des Funktionsaufrufes nicht in Bearbeitung sein, da von perform() zu Beginn der Bearbeitung verifizierte Eigenschaften bei der neuen Datenmenge nicht mehr gegeben sein könnten.

chGetNumOfRows — Tupelanzahl ermitteln

Definition:

int chGetNumOfRows(chp)
choice *chp;

Beschreibung:

liefert die Anzahl der Tupel in der Datenmenge der Choice *chp*.

Ergebnis:

Anzahl der Tupel

Hinweis:

Bei einer Choice, die keine Datenmenge besitzt, wird 0 zurückgegeben.

chUnifySelection — Markierung aller Tupel vereinheitlichen

Definition:

int chUnifySelection(chp, state)
choice *chp; BOOLEAN state;

Beschreibung:

weist allen Tupeln der Datenmenge der Choice *chp* die Markierung “selektiert” zu (state != 0) bzw. löscht die Markierung.

Ergebnis:

0

Siehe auch:

mr_tp_set_mark()

Hinweis:

Ist die Choice am Bildschirm dargestellt, muss ihre Darstellung anschließend neu aufgebaut werden.

Bei einer Choice, die keine Datenmenge besitzt, ist die Funktion wirkungslos.

ch_clear_selection — Markierungen aller Tupel löschen †

Definition:

void ch_clear_selection(chp)
choice *chp;

Beschreibung:

löscht bei allen Tupeln der Datenmenge der Choice *chp* die Markierung “selektiert”.

Siehe auch:

chUnifySelection()

Hinweis:

entspricht (void)chUnifySelection(chp, FALSE).

ch_count_selections — markierte Tupel zählen

Definition:

int ch_count_selections(chp)
choice *chp;

Beschreibung:

zählt die als “selektiert” markierten Tupel in der Datenmenge der Choice *chp*.

Siehe auch:

tp_get_mark()

Ergebnis:

Anzahl der markierten Tupel

Hinweis:

Bei einer Choice, die keine Datenmenge besitzt, wird 0 zurückgegeben.

ch_display_columns — Selo-artige Display-Funktion

Definition:

char *ch_display_columns(chp, tp, n)
choice *chp; TupelType tp; int n;

Beschreibung:

wird von *FIX* bei einer Choice, deren Datenbeschaffung mittels einer Pseudo-select-Anweisung erfolgt, zur Gewinnung der Darstellung eines Tupels verwendet, wenn der Entwickler keine eigene Routine spezifiziert hat. Der durch das Tupel *tp* repräsentierte Satz wird in der gleichen Weise wie im Selo dargestellt.

Parameter:

chp Zeiger auf Choice

tp Tupeldeskriptor

n lfd. Nummer des Tupels innerhalb der Memory Relation (*n* > 0)

Ergebnis:

Zeiger (!= (char *)0) auf Text, den *FIX* zur Darstellung des Tupels verwenden soll

Siehe auch:

stddisplay()

Hinweis:

Die Darstellung ist auf auf eine Zeichenfolge mit maximal 512 Bytes beschränkt.

Ein FXGRAPHICSTYPE-Wert wird durch eine der Spaltenbreite entsprechende Zahl von Leerzeichen wiedergegeben.

Wird in Source ausgeliefert (`$FXDIR/src`).

ch_exchange — Datenmenge einer Choice austauschen

Definition:

int ch_exchange(chp, newmrp, oldmrp_adr)
choice *chp; MemRelType mrp; MemRelType *oldmrp_adr;

Beschreibung:

ersetzt die augenblickliche Datenmenge der Choice *chp* durch die Datenmenge *newmrp*. Ist *newmrp* == (MemRelType)0, wird eine leere Memory Relation angelegt, anderenfalls muss die Datenmenge *newmrp* hinsichtlich Spaltenzahl, -typ und -länge bzw. -Qualifier mit der Satzstruktur der Choice übereinstimmen bzw. zur bisherigen Datenmenge passen.

kein Fehler, *oldmrp_adr* == 0:
die ursprüngliche Datenmenge wird freigegeben.

kein Fehler, *oldmrp_adr* != 0:
die ursprüngliche Datenmenge wird nicht freigegeben, an der Adresse *oldmrp_adr* wird ihr Deskriptor hinterlegt.

Fehler, *oldmrp_adr* == 0:
die Datenmenge bleibt unverändert.

Fehler, *oldmrp_adr* != 0:
die Datenmenge bleibt unverändert, an der Adresse *oldmrp_adr* wird (MemRelType)0 hinterlegt.

Findet ein Austausch statt, wird der erste Satz zum aktuellen Satz der Choice.

Ergebnis:

0 - Erfolg
!= 0 - Fehlercode

Siehe auch:

assign_mrel()

Hinweis:

Besitzt die neue Satzmenge keine Spaltenauszeichnung, versucht *FIX*, die gleiche Spalte wie in der ursprünglichen Satzmenge auszuzeichnen. Wenn dies nicht möglich ist, wird die erste Spalte ausgezeichnet.

Die Auswahlmarkierungen werden nicht verändert.

ch_fill_by_args — Datenmenge konstruieren †

Definition:

int ch_fill_by_args(chp, argc, argv, empty)
choice *chp; int argc; char **argv; BOOLEAN empty;

Beschreibung:

füllt aus dem Programm heraus die Choice *chp*, deren Datenmenge aus einer Memory Relation mit einer FXCHARTYPE-Spalte bestehen muss, mit Strings. *argv* ist ein Vektor von *argc* char-Pointern. Die Choice wird zuvor geleert, wenn *empty* != FALSE.

Ergebnis:

0 - Erfolg
1 - *chp* ist keine Choice oder besitzt keine Datenmenge
2 - *argc* == 0 oder *argv* == (char **)0

ch_moveto — auf Tupel positionieren

Definition:

BOOLEAN ch_moveto(chp, n)
choice *chp; int n;

Beschreibung:

positioniert auf das (*n*+1)-te Tupel in der Datenmenge der Choice *chp* (0 ≤ *n* < Anzahl Tupel) und bringt dieses ggf. in den sichtbaren Bereich. Besitzt die Datenmenge der Choice kein (*n*+1)-tes Tupel, ist die Funktion wirkungslos.

Ergebnis:

TRUE bei Erfolg
FALSE, wenn die Datenmenge von *chp* kein (*n*+1)-tes Tupel besitzt

Hinweis:

Ist die Choice am Bildschirm dargestellt, aber nicht aktiv, muss ihre Darstellung neu aufgebaut werden.

get_current_tupel — Deskriptor des aktuellen Tupels bestimmen

Definition:

TupelType get_current_tupel(chp)
choice *chp;

Beschreibung:

liefert, sofern *chp* aktiv ist, den Deskriptor des aktuellen Tupels der Choice *chp*.

Ergebnis:

Tupeldeskriptor bei Erfolg
(TupelType)0 bei Fehler

last_selected — markiertes Tupel einer Memory Relation ermitteln †

Definition:

BOOLEAN last_selected(mrp, tptr, nptr)
MemRelType mrp; TupelType *tptr; int *nptr;

Beschreibung:

testet, ob die Memory Relation *mrp* - die die Datenmenge einer Choice bilden sollte - ein als "selektiert" markiertes Tupel enthält. Bei Erfolg wird in **tptr* der Deskriptor des letzten so markierten Tupels und in **nptr* dessen Index ($0 \leq$ **nptr* < Anzahl Tupel) hinterlegt.

Ergebnis:

TRUE, wenn die Memory Relation *mrp* ein als "selektiert" markiertes Tupel enthält
FALSE sonst

Siehe auch:

ch_count_selections()

Hinweis:

(MemRelType)0 als Argument wird wie eine Memory Relation ohne Tupel behandelt.

Mit der Einführung der Operationen auf Memory Relations ist diese Funktion entbehrlich geworden.

loadchoice — Choice laden

Definition:

choice *loadchoice(mname, mrp)
char *mname; MemRelType mrp;

Beschreibung:

lädt die in der Beschreibungsdatei *mname* definierte Choice.

Falls *mrp* != (MemRelType)0 ist, so wird *mrp* als Deskriptor einer Memory Relation interpretiert und diese der Choice als initiale Datenmenge zugeordnet. Die Datenmenge *newmrp* muss hinsichtlich Spaltenzahl, -typ und -länge bzw. -Qualifier mit der Satzstruktur der Choice übereinstimmen bzw. zur bisherigen Datenmenge passen.

Ergebnis:

Zeiger auf die Interndarstellung bei Erfolg
(choice *)0 bei Fehler

Siehe auch:

loadmask(), loadmenue(), ch_exchange()

Hinweis:

Beim Laden erkannte Fehler werden mittels rt_msg() gemeldet.

stddisplay — Default-Display-Funktion

Definition:

char *stddisplay(chp, tp, n)
choice *chp; TupelType tp; int n;

Beschreibung:

wird von *FIX* bei einer Choice, deren Datenbeschaffung nicht mittels einer Pseudo-select-Anweisung erfolgt, zur Gewinnung der Darstellung eines Tupels verwendet, wenn der Entwickler keine eigene Routine spezifiziert hat. Beginnt

das Tupel mit einer FXCHARTYPE-Spalte, wird deren Inhalt (bis zum ersten nicht darstellbaren Zeichen), sonst die Konstante “<data>” zurückgegeben.

Parameter:

chp Zeiger auf Choice

tp Tupeldeskriptor

n lfd. Nummer des Tupels innerhalb der Memory Relation (*n* > 0)

Ergebnis:

Zeiger (!= (char ∗)0) auf Text, den *FIX* zur Darstellung des Tupels verwenden soll

Siehe auch:

ch_display_columns()

Hinweis:

Die Darstellung ist auf auf eine Zeichenfolge mit maximal 512 Bytes beschränkt.

Wird in Source ausgeliefert (`$FXDIR/src`).

stdexport — Standard-Wertübernahme aus Choice-Auswahl in Feld

Definition:

```
BOOLEAN stdexport(f)
field ∗f;
```

Beschreibung:

f muss eine Choice besitzen, in deren Datenmenge die erste Spalte entweder denselben Typ wie *f* oder den Typ FXCHARTYPE hat. Als Wert von *f* wird - mittels fputval() bzw. fputstring() - der Wert der ersten Spalte des *ersten* als “selektiert” markierten Tupels übernommen, sofern ein solches existiert, anderenfalls wird das Feld gelöscht. stdexport() ist die Default-Funktion für die Wertübernahme (**export** ohne anwendungsspezifische Routine).

Ergebnis:

TRUE bei Erfolg
FALSE bei Fehler

Siehe auch:

stdimport(), addfct(), getfct()

Hinweis:

Eine Choice, die keine Datenmenge besitzt, wird wie eine mit leerer Datenmenge behandelt.

Wird in Source ausgeliefert (`$FXDIR/src`).

stdimport — Standard-Abgleich von Choice-Auswahl mit Feldwert

Definition:

```
BOOLEAN stdimport(f)
field ∗f;
```

Beschreibung:

f muss eine Choice besitzen. In deren Datenmenge werden genau die Tupel als “selektiert” markiert, bei denen der Wert der ersten Spalte mit dem des Feldes übereinstimmt. Werden Tupel markiert, wird die Choice auf das erste markierte Tupel positioniert, anderenfalls auf das erste Tupel, sofern vorhanden. stdimport() ist die Default-Funktion für den Wertabgleich (**import** ohne anwendungsspezifische Routine).

Ergebnis:

TRUE bei Erfolg
FALSE bei Fehler

Siehe auch:

stdexport(), addfct(), getfct()

Hinweis:

Eine Choice, die keine Datenmenge besitzt, wird wie eine mit leerer Datenmenge behandelt.

Wird in Source ausgeliefert (`$FXDIR/src`).

tp_get_mark — Tupel-Markierung abfragen

Definition:

BOOLEAN tp_get_mark(tp)
TupelType tp;

Beschreibung:

testet, ob das Tupel *tp* als “selektiert” markiert ist.

Ergebnis:

TRUE - selektiert
FALSE - nicht selektiert

Siehe auch:

mr_tp_set_mark()

14 Memory Relation

fx_mrgetobject — Objekt bestimmen, an das die Memory Relation gebunden ist

Definition:

obj *fx_mrgetobject(mrp)
MemRelType mrp;

Beschreibung:

liefert das Objekt, an das die Memory Relation *mrp* gebunden ist.

Ergebnis:

Zeiger auf ein Objekt (Maske oder Choice)
(obj *)0 - keine Bindung vorhanden

mr_act_row — Position des aktuellen Tupels ermitteln

Definition:

int mr_act_row(mrp)
MemRelType mrp;

Beschreibung:

ermittelt die Position des aktuellen Tupels innerhalb der Memory Relation *mrp*.

Ergebnis:

> 0 - Position
0 - kein Tupel vorhanden
< 0 - Fehler

Siehe auch:

sm_act_row()

mr_act_tup — Deskriptor des aktuellen Tupels einer Memory Relation bestimmen

Definition:

TupelType mr_act_tup(mrp)
MemRelType mrp;

Beschreibung:

bestimmt den Deskriptor des aktuellen Tupels der Memory Relation *mrp*.

Ergebnis:

Tupeldeskriptor bei Erfolg
(TupelType)0 bei Fehler

mr_activate — Memory Relation öffnen

Definition:

int mr_activate(mrp)
MemRelType mrp;

Beschreibung:

macht die mittels mr_create() erzeugte Memory Relation *mrp* zur Benutzung bereit (initialisiert Puffergrößen, allokiert Puffer etc).

Ergebnis:

SUCCESS - Erfolg
DBR_DBOPEN - Memory Relation bereits aktiviert
LOGIC_ERR - Argument unzulässig, z.B. Spalten unvollständig definiert

mr_addListener — Memory Relation Listener registrieren

Definition:

int mr_addListener(mrp, mrlp)
MemRelType mrp; mrlistener_t *mrlp;

Beschreibung:

registriert *mrlp* als Memory Relation Listener an der Memory Relation *mrp*.

Ergebnis:

SUCCESS - Erfolg
-1 - Fehler (bereits Listener registriert)

Siehe auch:

mr_getListener(), mr_removeListener()

mr_add_row — Tupel anfügen

Definition:

int mr_add_row(mrp)
MemRelType mrp;

Beschreibung:

hängt ein Tupel an die Memory Relation *mrp* an. Für Spalten, für die eine Bindung besteht, wird der Wert von der entsprechenden Adresse übernommen, die übrigen Spalten sind leer. Bei Erfolg wird das neue Tupel zum aktuellen Tupel, bei einem Fehler bleibt das aktuelle Tupel erhalten.

Ergebnis:

SUCCESS - Erfolg
DBR_NOTOPEN - Memory Relation nicht aktiviert
NOSPACE - keinen Speicherplatz erhalten
< 0 - sonstiger Fehler

Siehe auch:

sm_add(), mr_insert_row()

Hinweis:

Bei Spalten, die NULL nicht unterstützen, wird NULL bei der Übernahme geeignet konvertiert.

mr_bind_column — Spalte mit Variable binden

Definition:

int mr_bind_column(mrp, colno, adr)
MemRelType mrp; int colno; char *adr;

Beschreibung:

bindet die *colno*-te Spalte der Memory Relation *mrp* ($1 \leq$ *colno* $\leq$ Anzahl Spalten) mit der Adresse *adr* (muss Adresse einer Variablen sein, die Spaltenwert aufnehmen kann) bzw. löst die bestehende Bindung, falls *adr* == (char *)0.

Ergebnis:

SUCCESS - Erfolg
LOGIC_ERR - unzulässiges Argument

Siehe auch:

mr_define_column()

Hinweis:

Ist der Typ der *colno*-ten Spalte gleich FXDTIMETYPE oder FXINVTYPE, so muss, ist *adr* ungleich (char *)0, der an dieser Adresse vorgefundene Qualifier gleich 0 sein oder mit dem der Spalte übereinstimmen.

Die Memory Relation braucht zum Zeitpunkt des Aufrufs noch nicht aktiviert zu sein.

mr_clearCellUpdateMark — Änderungsmarkierung zu einer Tupelzelle zurücksetzen

Definition:

void mr_clearCellUpdateMark(mrp, tp, n)
MemRelType mrp; TupelType tp; int n;

Beschreibung:

setzt die Änderungsmarkierung zu der Zelle im Datenteil des Tupels *tp*, die den Wert der *n*-ten Spalte ($1 \leq n \leq$ Anzahl Spalten) hält, zurück.

Siehe auch:

mr_isCellUpdateMarkSet()

Hinweis:

Das Zurücksetzen der Änderungsmarkierung ist der Anwendung vorbehalten; *FIX*-intern wird diese Funktion nie aufgerufen.

mr_colval — Spaltenwert ermitteln

Definition:

```
char *mr_colval(mrp, n, tp)
MemRelType mrp; int n; TupelType tp;
```

Beschreibung:

ermittelt einen Zeiger in den Datenteil des Tupels *tp*, wo der Wert der *n*-ten Spalte ($1 \leq n \leq$ Anzahl Spalten) gespeichert ist.

Ergebnis:

Adresse des Spaltenwertes bei Erfolg
(char *)0 bei Fehler

Siehe auch:

mr_explain_column()

mr_create — Memory Relation erzeugen

Definition:

```
int mr_create(mrp_adr, anz)
MemRelType *mrp_adr; int anz;
```

Beschreibung:

erzeugt eine Memory Relation für Tupel mit *anz* Spalten und hinterlegt deren Deskriptor an der Adresse *mrp_adr*. Beim Aufruf muss dort (MemRelType)0 vorgefunden werden. Die Attribute der Spalten (Name, Typ etc.) müssen anschließend mittels mr_define_column() definiert werden. Bevor Tupel verwaltet werden können, muss die Memory Relation mittels mr_activate() aktiviert werden.

Ergebnis:

SUCCESS - Erfolg
DBR_DBOPEN - unzulässiges Argument (**mrp_adr* != (MemRelType)0)
LOGIC_ERR - unzulässiges Argument (*anz* < 0)
NOSPACE - keinen Speicherplatz erhalten

Siehe auch:

mr_drop()

mr_create_by_query — Satzmenge aus Datenbank lesen

Definition:

long mr_create_by_query(mrp_adr, stmt)
MemRelType *mrp_adr; char *stmt;

Beschreibung:

hinterlegt an der Adresse *mrp_adr* den Deskriptor einer Memory Relation, deren Tupel die durch die Pseudo-select-Anweisung *stmt* gefundenen Sätze enthalten.

Ergebnis:

0 bei Erfolg (**mrp_adr* ist Deskriptor neuer Memory Relation)
!= 0 bei Fehler (**mrp_adr* nicht geändert)

Hinweis:

Steht in *mrp_adr* bereits der Deskriptor einer Memory Relation, so wird diese, sofern die Datenbeschaffung erfolgreich verlief, freigegeben.

Eine into-Klausel als Teil der Pseudo-select-Anweisung wird nicht unterstützt.

mr_define_column — Spalte definieren

Definition:

int mr_define_column(mrp, colno, buf)
MemRelType mrp; int colno; struct colspec *buf;

Beschreibung:

setzt die Attribute der *colno*-ten Spalte (1 ≤ *colno* ≤ Anzahl Spalten) der Memory Relation *mrp* entsprechend *buf*. Dies muss genau einmal pro Spalte geschehen. *buf* muss auf eine Struktur folgender Form zeigen (`fix/mrel.h`):

```
struct colspec {
    char *colname;    /* Name (!= (char *)0; wird in Memory Relation kopiert) */
    int coltype;      /* Typ kodiert gemäß fix/fxtypes.h */
    int collen;       /* FXCHARTYPE, FXGRAPHICSTYPE: Länge der Spalte (in Bytes) */
                      /* FXDTIMETYPE, FXINVTYPE: Qualifier */
    int colnull;      /* == 0 bedeutet “keine NULL-Werte” */
    char *colbinding; /* Bindung an Variablenadresse (siehe mr_bind_column()) */
    int offset;       /* reserviert, bleibt unausgewertet */
}
```

Ergebnis:

SUCCESS - Erfolg
DBR_DBOPEN - Memory Relation bereits aktiviert
LOGIC_ERR - unzulässiges Argument oder Spalte bereits definiert
NOSPACE - keinen Speicherplatz erhalten

Siehe auch:

mr_bind_column()

Hinweis:

Ist *buf*->coltype gleich FXDTIMETYPE oder FXINVTYPE, so muss, ist *buf*->colbinding ungleich (char *)0, der an dieser Adresse vorgefundene Qualifier gleich 0 sein oder mit *buf*->collen übereinstimmen.

mr_delete_row — Tupel löschen

Definition:

int mr_delete_row(mrp)
MemRelType mrp;

Beschreibung:

löscht das aktuelle Tupel der Memory Relation *mrp*. Sofern diese weitere Tupel enthält, wird das auf das gelöschte folgende Tupel zum aktuellen Tupel, sonst das ihm vorausgehende.

Ergebnis:

SUCCESS - Erfolg
DBR_NOTOPEN - Memory Relation nicht aktiviert
DBR_NODATA - kein Tupel vorhanden

Siehe auch:

sm_delete()

Hinweis:

Die Werte an den gebundenen Adressen werden nur geändert, wenn nach dem Löschen noch ein (aktuelles) Tupel existiert.

mr_drop — Memory Relation freigeben

Definition:

int mr_drop(mrp_adr)
MemRelType *mrp_adr;

Beschreibung:

gibt die Memory Relation, deren Deskriptor an der Adresse mrp_adr hinterlegt ist, und alle von ihr verwalteten Tupel frei und ersetzt den Deskriptor durch (MemRelType)0.

Ergebnis:

SUCCESS - Erfolg
LOGIC_ERR - unzulässiges Argument (**mrp_adr* == (MemRelType)0)

Siehe auch:

mr_create()

mr_explain_column — Spaltendefinition abfragen

Definition:

int mr_explain_column(mrp, colno, buf)
MemRelType mrp; int colno; struct colspec *buf;

Beschreibung:

liest die Attribute der *colno*-ten Spalte ($1 \leq colno \leq$ Anzahl Spalten) der Memory Relation *mrp* nach *buf*. *buf* muss auf eine Variable vom Typ struct colspec (`fix/mrel.h`) zeigen.

Ergebnis:

SUCCESS - Erfolg
LOGIC_ERR - unzulässiges Argument oder Spalte nicht definiert

Siehe auch:

mr_define_column(), mr_colval()

Hinweis:

Die Memory Relation braucht zum Zeitpunkt des Aufrufs noch nicht aktiviert zu sein.

Im Fehlerfall ist der Wert von *buf* anschließend undefiniert.

mr_fetch_row — aktuelles Tupel wählen

Definition:

int mr_fetch_row(mrp, indicator, value, tp_adr)
MemRelType mrp; int indicator; char *value; TupelType *tp_adr;

Beschreibung:

sucht in der Memory Relation *mrp* nach dem durch *indicator* (und ggf. *value*) beschriebenen Tupel. Wird das Tupel gefunden, wird es zum aktuellen Tupel und die Werte der Spalten, für die Bindungen bestehen, werden an die entsprechenden Adressen kopiert; ansonsten bleibt das aktuelle Tupel erhalten.

Zulässige *indicator*, *value*-Paare sind:

FIRST	beliebig	erstes Tupel
LAST	beliebig	letztes Tupel
CURRENT	beliebig	das aktuelle Tupel
INDEXED	(char *)*n*	das *n*-te Tupel (1 ≤ *n* ≤ Anzahl Tupel)
BY_ROWID	(char *)*n*	das Tupel mit ROWID *n*
NEXT	beliebig	das auf das aktuelle folgende Tupel
PREVIOUS	beliebig	das dem aktuellen vorausgehende Tupel
RELATIVE	(char *)*n*	das *n*-nächste (bzw. *n*-vorangehende, falls *n* < 0) Tupel

Wurde mittels mr_select_column() eine Spalte ausgezeichnet, stehen weitere Möglichkeiten zur Verfügung (*p* muss auf einen Wert vom gleichen Typ wie die ausgezeichnete Spalte zeigen):

EQUAL	(char *)*p*	das erste Tupel, das in der ausgezeichneten Spalte den gleichen Wert enthält, auf den *p* zeigt
GTEQ	(char *)*p*	das erste Tupel, das in der ausgezeichneten Spalte einen Wert enthält, der größer oder gleich dem ist, auf den *p* zeigt
GREATER	(char *)*p*	das erste Tupel, das in der ausgezeichneten Spalte einen Wert enthält, der größer als der ist, enthält, auf den *p* zeigt

Bei der vergleichenden Suche kann durch Hinzusetzen des Attributs SRCH_BEHIND (EQUAL|SRCH_BEHIND) die Suche auf Nachfolger des aktuellen Satzes beschränkt werden.

Ergebnis:

SUCCESS - Tupel gefunden
DBR_NOTOPEN - Memory Relation nicht aktiviert
DBR_NODATA - kein Tupel vorhanden
!= 0 - Tupel nicht gefunden

Siehe auch:

sm_find(), mr_get_tupel()

Hinweis:

Ist ein Vergleich von Werten erforderlich, müssen bei den Typen FXDTIMETYPE und FXINVTYPE die Qualifier von Spalte und Vergleichswert im Sinne von fxvalcmp() kompatibel sein. Beim Typ FXGRAPHICSTYPE müssen die Längen von Spalte und Vergleichswert übereinstimmen.

Der Parameter *tp_adr* soll die Migration erleichtern: Ist er ungleich (TupelType *)0, wird hier im Erfolgsfall der Deskriptor des aktuellen Tupel hinterlegt.

mr_first_tup — Deskriptor des ersten Tupels einer Memory Relation bestimmen

Definition:

TupelType mr_first_tup(mrp)
MemRelType mrp;

Beschreibung:

bestimmt den Deskriptor des ersten Tupels der Memory Relation *mrp*.

Ergebnis:

Tupeldeskriptor bei Erfolg
(TupelType)0 bei Fehler

mr_get_id — ID der Memory Relation abfragen

Definition:

unsigned long mr_get_id(mrp)
MemRelType mrp;

Beschreibung:

liefert die ID der Memory Relation *mrp*.

Ergebnis:

> 0 - ID der Memory Relation *mrp*
0 - Fehler

mr_getListener — Memory Relation Listener abfragen

Definition:

mrlistener_t *mr_getListener(mrp)
MemRelType mrp;

Beschreibung:

liefert den an der Memory Relation *mrp* registrierten Memory Relation Listener.

Ergebnis:

Adresse des Listeners bei Erfolg
(mrlistener_t *)0 - kein Listener registriert

Siehe auch:

mr_addListener(), mr_removeListener()

mr_get_props — anwendungsspezifische Information zur Memory Relation abfragen

Definition:

PrivatePropertyType mr_get_props(mrp)
MemRelType mrp;

Beschreibung:

liefert den initialen bzw. den mittels mr_set_props() zur Memory Relation *mrp* gespeicherten (anwendungsspezifischen) Wert.

Ergebnis:

augenblicklicher Wert

Siehe auch:

mr_set_props()

Hinweis:

Die Struktur PrivatePropertyType ist in `fix/props.h` definiert.

Die Memory Relation braucht zum Zeitpunkt des Aufrufs noch nicht aktiviert zu sein.

mr_get_selected_column — ausgezeichnete Spalte ermitteln

Definition:

int mr_get_selected_column(mrp)
MemRelType mrp;

Beschreibung:

liefert die Nummer der als Such- oder Sortierspalte ausgezeichneten Spalte der Memory Relation *mrp*.

Ergebnis:

> 0 - Nummer der ausgezeichneten Spalte
0 - Memory Relation besitzt keine Spaltenauszeichnung
LOGIC_ERR - unzulässiges Argument

Siehe auch:

mr_select_column()

Hinweis:

Die Memory Relation braucht zum Zeitpunkt des Aufrufs noch nicht aktiviert zu sein.

mr_get_tupel — Tupel auffinden

Definition:

int mr_get_tupel(mrp, indicator, value, tp_adr, act_row_adr)
MemRelType mrp; int indicator; char *value; TupelType *tp_adr; int *act_row_adr;

Beschreibung:

sucht in der Memory Relation *mrp* nach dem durch *indicator* (und ggf. *value*) beschriebenen Tupel.

Wird ein solches Tupel gefunden, wird sein Deskriptor an der Adresse *tp_adr*, seine Position an der Adresse *act_row_adr* hinterlegt, sofern die jeweiligen Adressen ungleich 0 sind.

Ergebnis:

SUCCESS - Tupel gefunden
DBR_NOTOPEN - Memory Relation nicht aktiviert
DBR_NODATA - kein Tupel vorhanden
!= 0 - Tupel nicht gefunden

Siehe auch:

mr_get_tupel_variadic()

Hinweis:

mr_get_tupel_variadic() ist zu bevorzugen: hier sind keine überzähligen Argumente oder technische Casts notwendig.

mr_get_tupel_variadic — Tupel auffinden

Definition:

int mr_get_tupel_variadic(mrp, indicator, ...)
MemRelType mrp; int indicator; /* ...; TupelType *tp_adr; int *act_row_adr; */

Beschreibung:

sucht in der Memory Relation *mrp* nach dem durch *indicator* (und ggf. zusätzlichen Argumenten) beschriebenen Tupel.

Die beiden letzten Argumente müssen vom Typ (TupelType *) und (int *) sein. Wird ein Tupel gefunden, wird sein Deskriptor an der als vorletztes Argument übergebenen Adresse, seine Position an der als letztes Argument übergebenen Adresse hinterlegt, sofern die jeweiligen Adressen ungleich 0 sind.

Zulässig sind:

FIRST, LAST, PREVIOUS, NEXT, CURRENT: kein weiteres Argument

INDEXED: long - absolute Position

RELATIVE: long - relative Position zum aktuellen Satz

COMPARISON, EQUAL, GTEQ, GREATER: char * - typgleicher Vergleichswert

BY_ROWID: long - ROWID

BY_ACCEPTOR: BOOLEAN (*)(/* MemRelType, TupelType, void * */) und void *

Ergebnis:

SUCCESS - Tupel gefunden
DBR_NOTOPEN - Memory Relation nicht aktiviert
DBR_NODATA - kein Tupel vorhanden
!= 0 - Tupel nicht gefunden

Siehe auch:

mr_fetch_row()

Hinweis:

Ist ein Vergleich von Werten erforderlich, müssen bei den Typen FXDTIMETYPE und FXINVTYPE die Qualifier von Spalte und Vergleichswert im Sinne von fxvalcmp() kompatibel sein. Beim Typ FXGRAPHICSTYPE müssen die Längen von Spalte und Vergleichswert übereinstimmen.

Das aktuelle Tupel bleibt erhalten.

mr_insert_row — Tupel einfügen

Definition:

int mr_insert_row(mrp)
MemRelType mrp;

Beschreibung:

fügt vor dem aktuellen ein Tupel in die Memory Relation *mrp* ein. Für Spalten, für die eine Bindung besteht, wird der Wert von der entsprechenden Adresse übernommen, die übrigen Spalten sind leer. Bei Erfolg wird das neue Tupel zum aktuellen Tupel, bei einem Fehler bleibt das aktuelle Tupel erhalten.

Ergebnis:

SUCCESS - Erfolg
DBR_NOTOPEN - Memory Relation nicht aktiviert
NOSPACE - keinen Speicherplatz erhalten
< 0 - sonstiger Fehler

Siehe auch:

sm_insert()

Hinweis:

Bei Spalten, die NULL nicht unterstützen, wird NULL bei der Übernahme geeignet konvertiert.

mr_isCellUpdateMarkSet — Änderungsmarkierung zu einer Tupelzelle abfragen

Definition:

BOOLEAN mr_isCellUpdateMarkSet(mrp, tp, n)
MemRelType mrp; TupelType tp; int n;

Beschreibung:

liefert die Änderungsmarkierung zu der Zelle im Datenteil des Tupels *tp*, die den Wert der *n*-ten Spalte ($1 \leq n \leq$ Anzahl Spalten) hält. Die Änderungsmarkierung wird von *FIX*-Funktionen gesetzt, wenn sie einen Wert in die Zelle schreiben, der sich von dem dort vorgefundenen Wert unterscheidet, und wenn ein neu angelegter Satz mit Werten belegt wird.

Ergebnis:

TRUE - Update-Markierung gesetzt
FALSE - Update-Markierung nicht gesetzt / unzulässiges Argument

Siehe auch:

mr_clearCellUpdateMark()

mr_last_tup — Deskriptor des letzten Tupels einer Memory Relation bestimmen

Definition:

TupelType mr_last_tup(mrp)
MemRelType mrp;

Beschreibung:

bestimmt den Deskriptor des letzten Tupels der Memory Relation *mrp*.

Ergebnis:

Tupeldeskriptor bei Erfolg
(TupelType)0 bei Fehler

mr_nth_tup — Deskriptor des n-ten Tupels einer Memory Relation bestimmen

Definition:

TupelType mr_nth_tup(mrp, n)
MemRelType mrp; int n;

Beschreibung:

bestimmt den Deskriptor des *n*-ten Tupels der Memory Relation *mrp* ($1 \leq n \leq$ Anzahl Tupel).

Ergebnis:

Tupeldeskriptor bei Erfolg
(TupelType)0 bei Fehler

Siehe auch:

mr_first_tup(), mr_last_tup(), mr_act_tup(), tp_previous(), tp_next()

mr_num_of_columns — Spaltenzahl ermitteln

Definition:

int mr_num_of_columns(mrp)
MemRelType mrp;

Beschreibung:

ermittelt die Anzahl der Spalten der Memory Relation *mrp*.

Ergebnis:

> 0 - Anzahl Spalten
≤ 0 - Fehler

mr_num_of_rows — Anzahl Tupel ermitteln

Definition:

int mr_num_of_rows(mrp)
MemRelType mrp;

Beschreibung:

ermittelt die Anzahl der in der Memory Relation *mrp* enthaltenen Tupel.

Ergebnis:

≥ 0 - Anzahl Tupel
< 0 - Fehler

Siehe auch:

sm_anz_row()

mr_qsort — Memory Relation sortieren

Definition:

BOOLEAN mr_qsort(mrp, colno, dir, …)
MemRelType mrp; int colno, dir, …;

Beschreibung:

sortiert die Memory Relation *mrp* gemäß fxvalcmp() aufsteigend auf den angegebenen Spalten (1 ≤ *colno* ≤ Anzahl Spalten); anschließend ist das erste Tupel das aktuelle Tupel. *dir* muss eines der Makros ASCENDING oder DESCENDING sein.

Die Argumentliste muss mit einer 0 hinter dem letzten *colno*, *dir*-Paar abgeschlossen werden.

Ergebnis:

TRUE bei Erfolg
FALSE sonst

Siehe auch:

sm_sort()

Hinweis:

Es werden maximal 32 Sortierspalten ausgewertet.

Die Sortierung erfolgt stabil, d.h. Sätze mit gleichem Schlüssel werden nicht umgeordnet.

mr_removeListener — Memory Relation Listener deregistrieren

Definition:

int mr_removeListener(mrp, mrlp)
MemRelType mrp; mrlistener_t *mrlp;

Beschreibung:

deregistriert *mrlp* als Memory Relation Listener an der Memory Relation *mrp*.

Ergebnis:

SUCCESS - Erfolg
-1 - Fehler (*mrlp* nicht als Listener registriert)

Siehe auch:

mr_addListener(), mr_getListener()

mr_rowid — ROWID des aktuellen Tupels abfragen

Definition:

long mr_rowid(mrp)
MemRelType mrp;

Beschreibung:

ermittelt die ROWID des aktuellen Tupels der Memory Relation *mrp*.

Ergebnis:

> 0 - ROWID des aktuelles Tupels
0 - kein (aktuelles) Tupel vorhanden
< 0 - Fehler

Siehe auch:

sm_rowid()

mr_rowstate — Eigenschaften des aktuellen Tupels ermitteln/ändern

Definition:

long mr_rowstate(mrp, mode, state)
MemRelType mrp; int mode; long state;

Beschreibung:

ermittelt oder ändert die Eigenschaften des aktuellen Tupels der Memory Relation *mrp*. Das Verhalten der Funktion wird gesteuert vom Parameter *mode*. *mode* == P_ADD verleiht, *mode* == P_DEL entzieht dem Tupel die Eigenschaften *state*. Mit *mode* == P_GET kann abgefragt werden, welche der Eigenschaften in *state* das Tupel besitzt. Für Tupel werden nur die Eigenschaften TP_USR1, TP_USR2, TP_USR3, TP_USR4 sowie TOUCHED unterstützt.

Ergebnis:

≥ 0 - die zusätzlich gesetzten Eigenschaften (*mode* == P_ADD)
die nicht länger gesetzten Eigenschaften (*mode* == P_DEL)
die dem Tupel eigenen Eigenschaften aus *state* (*mode* == P_GET)
DBR_NOTOPEN - Memory Relation nicht aktiviert
DBR_NODATA - kein Tupel vorhanden

Siehe auch:

tp_state()

Hinweis:

Mittels P_GET mit *state* ==(unsigned long)~0 können alle unterstützten Eigenschaften des aktuellen Tupels abgefragt werden.

mr_select_column — Spalte auszeichnen

Definition:

int mr_select_column(mrp, colno)
MemRelType mrp; int colno;

Beschreibung:

zeichnet die *colno*-te Spalte der Memory Relation mrp (1 ≤ *colno* ≤ Anzahl Spalten) als Suchspalte aus, bzw. löscht, für *colno* == 0, eine bestehende Auszeichnung als Suchspalte.

Ergebnis:

SUCCESS - Erfolg
LOGIC_ERR - unzulässiges Argument

Siehe auch:

mr_fetch_row(), sm_selfield(), mr_get_selected_column()

Hinweis:

Die Memory Relation braucht zum Zeitpunkt des Aufrufs noch nicht aktiviert zu sein.

mr_set_props — anwendungsspezifische Information zur Memory Relation hinterlegen

Definition:

void mr_set_props(mrp, value)
MemRelType mrp; PrivatePropertyType value;

Beschreibung:

speichert zur Memory Relation *mrp* den (anwendungsspezifischen) Wert *value*.

Siehe auch:

mr_get_props()

Hinweis:

Die Struktur PrivatePropertyType ist in `fix/props.h` definiert.

Die Memory Relation braucht zum Zeitpunkt des Aufrufs noch nicht aktiviert zu sein.

mr_tp_set_mark — Tupel-Markierung setzen

Definition:

void mr_tp_set_mark(mrp, tp, state)
MemRelType mrp; TupelType tp; BOOLEAN state;

Beschreibung:

markiert das Tupel *tp* der Memory Relation *mrp* als “selektiert” (*state* == TRUE) bzw. löscht dessen Markierung (*state* == FALSE).

Siehe auch:

tp_get_mark()

mr_update_row — Tupel modifizieren

Definition:

int mr_update_row(mrp)
MemRelType mrp;

Beschreibung:

modifiziert das aktuelle Tupel der Memory Relation *mrp*. Für Spalten, für die eine Bindung besteht, wird der Spaltenwert durch den Wert an der entsprechenden Adresse ersetzt, die übrigen Spalten bleiben unverändert.

Ergebnis:

SUCCESS Erfolg
DBR_NOTOPEN Memory Relation nicht aktiviert
DBR_NODATA kein Tupel vorhanden

Siehe auch:

sm_put()

Hinweis:

Bei Spalten, die NULL nicht unterstützen, wird NULL bei der Übernahme geeignet konvertiert.

tp_data — Zeiger auf Datenteil eines Tupels ermitteln

Definition:

char *tp_data(tp)
TupelType tp;

Beschreibung:

liefert die Adresse des Datenteils des Tupels *tp*.

Ergebnis:

Zeiger auf Datenteil

Siehe auch:

mr_colval()

Hinweis:

Die Verwendung der Funktion macht i. Allg. nur Sinn, wenn die Tupel genau eine Spalte umfassen.

tp_next — Nachfolger-Tupel ermitteln

Definition:

TupelType tp_next(tp)
TupelType tp;

Beschreibung:

liefert den Deskriptor des auf das Tupel *tp* folgenden Tupels.

Ergebnis:

Tupeldeskriptor
(TupelType)0, wenn tp keinen Nachfolger besitzt

Siehe auch:

tp_previous()

Hinweis:

Das aktuelle Tupel wird nicht geändert.

tp_previous — Vorgänger-Tupel ermitteln

Definition:

TupelType tp_previous(tp)
TupelType tp;

Beschreibung:

liefert den Deskriptor des dem Tupel *tp* vorausgehenden Tupels.

Ergebnis:

Tupeldeskriptor
(TupelType)0, wenn *tp* keinen Vorgänger besitzt

Siehe auch:

tp_next()

Hinweis:

Das aktuelle Tupel wird nicht geändert.

tp_rowid — ROWID eines Tupels abfragen

Definition:

long tp_rowid(tp)
TupelType tp;

Beschreibung:

liefert die ROWID des Tupels *tp*.

Ergebnis:

ROWID

Siehe auch:

mr_rowid()

tp_state — Tupeleigenschaften abfragen oder ändern

Definition:

unsigned long tp_state(f, mode, state)
TupelType tp; int mode; unsigned long state;

Beschreibung:

dient zur Behandlung der Eigenschaften des Tupels *tp*. Das Verhalten der Funktion wird bestimmt vom Parameter *mode*. *state* muss eine sinnvolle Kombination von Tupeleigenschaften sein. *mode* == P_ADD verleiht, *mode* == P_DEL entzieht dem Tupel *tp* die Eigenschaften *state*. Mit *mode* == P_GET kann abgefragt werden, welche der Eigenschaften in *state tp* besitzt. Für Tupel werden nur die Eigenschaften TP_USR1, TP_USR2, TP_USR3, TP_USR4 sowie TOUCHED unterstützt.

Ergebnis:

die zusätzlich gesetzten Eigenschaften (*mode* == P_ADD)
die nicht länger gesetzten Eigenschaften (*mode* == P_DEL)
die *tp* eigenen Eigenschaften aus *state* (*mode* == P_GET)

Siehe auch:

mr_rowstate()

Hinweis:

Mittels P_GET mit *state* == (unsigned long)~0 können alle unterstützten Eigenschaften von *tp* abgefragt werden.

15 Hilfetexte

chelp — privaten Hilfetext zeigen

Definition:

```
void chelp(f)
field *f;
```

Beschreibung:

zeigt den Hilfetext zum Feld *f*, wenn er im Verzeichnis `$CHLPPATH` existiert.

Siehe auch:

n_help(), help()

entry_help — Hilfetext zu Menüpunkt zeigen

Definition:

```
void entry_help()
```

Beschreibung:

zeigt den Hilfetext zum aktuellen Menüpunkt, wenn er im Verzeichnis `$HLPPATH` existiert.

Siehe auch:

n_help(), Makros ACT_MP_NAM, ACT_MP_PTR

Hinweis:

sinnvoll, wenn ein Menüpunkt nur dazu dienen soll, einen Hilfetext zu zeigen; dieser wird dann sowohl über Taste HP als auch durch Menü-Auswahl (RT) erreicht.

help — anwendungsbezogenen Hilfetext zeigen

Definition:

```
void help(f)
field *f;
```

Beschreibung:

zeigt den Hilfetext zum Feld *f*, wenn er im Verzeichnis `$HLPPATH` existiert.

Siehe auch:

n_help(), chelp()

n_help — beliebigen anwendungsbezogenen Hilfetext ausgeben

Definition:

```
void n_help(f, name)
field *f; char *name;
```

Beschreibung:

zeigt den Hilfetext *name*, wenn er im Verzeichnis `$HLPPATH` existiert. *f* dient nur zur Positionierung des Windows.

Siehe auch:

help(), chelp()

t_help — Tastenhilfe anzeigen

Definition:

```
void t_help(f)
field *f;
```

Beschreibung:

“Tasten-Hilfe”: zeigt den Hilfetext `HP_TASTEN` im Verzeichnis `$FXDIR/runtime/hpd`. Dieser erklärt die standardmäßige Bedeutung der Sondertasten.

16 Layout

layout_getattr — Attribut im Layout ermitteln

Definition:

```
int layout_getattr(objp, y, x)
obj *objp; int y; int x;
```

Beschreibung:

liefert das Attribut in Zeile *y*, Spalte *x* des Layouts des Objekts *objp*.

Ergebnis:

Attribut bei Erfolg,
< 0 - Fehler (kein Layout, ungültige Position)

Siehe auch:

layout_setattr(), layout_getchar(), layout_getcharset(), layout_getvideo()

layout_getchar — Zeichencode im Layout ermitteln

Definition:

```
int layout_getchar(objp, y, x)
obj *objp; int y; int x;
```

Beschreibung:

liefert den Zeichencode in Zeile *y*, Spalte *x* des Layouts des Objekts *objp*.

Ergebnis:

Zeichencode bei Erfolg,
< 0 - Fehler (kein Layout, ungültige Position)

Siehe auch:

layout_setchar(), layout_getattr(), layout_getcharset(), layout_getvideo()

layout_getcharset — Zeichensatz eines Zeichens im Layout ermitteln

Definition:

int layout_getcharset(objp, y, x)
obj *objp; int y; int x;

Beschreibung:

liefert den Zeichensatz des Zeichens in Zeile *y*, Spalte *x* des Layouts des Objekts *objp*.

Ergebnis:

CHARSET1 - Standard-Zeichensatz,
CHARSET2 - Grafik-Zeichensatz,
< 0 - Fehler (kein Layout, ungültige Position)

Siehe auch:

layout_getchar(), layout_getvideo(), layout_putstr()

Hinweis:

CHARSET1 und CHARSET2 sind Makros aus `video.h`.

layout_getvideo — Videoattribute eines Zeichens im Layout ermitteln

Definition:

int layout_getvideo(objp, y, x)
obj *objp; int y; int x;

Beschreibung:

liefert die Videoattribute des Zeichens in Zeile *y*, Spalte *x* des Layouts des Objekts *objp*.

Ergebnis:

Bitmaske mit Bits aus VIDEOMASK bei Erfolg,
< 0 - Fehler (kein Layout, ungültige Position)

Siehe auch:

layout_getchar(), layout_getcharset(), layout_putstr()

Hinweis:

VIDEOMASK ist ein Makro aus `video.h`.

layout_setattr — Code für ein Attribut ins Layout eintragen

Definition:

int layout_setattr(objp, y, x, attr)
obj *objp; int y; int x; int attr;

Beschreibung:

trägt als Attribut in Zeile *y*, Spalte *x* des Layouts des Objekts *objp* den Wert *attr* ein.

Ergebnis:

0 - Erfolg,
< 0 - Fehler (kein Layout, ungültige Position)

Siehe auch:

layout_getattr(), layout_setchar()

Hinweis:

attr wird nicht auf Gültigkeit geprüft.

layout_setchar — Code für ein Zeichen ins Layout eintragen

Definition:

int layout_setchar(objp, y, x, c)
obj *objp; int y; int x; int c;

Beschreibung:

trägt als Zeichencode in Zeile *y*, Spalte *x* des Layouts des Objekts *objp* den Wert *c* ein.

Ergebnis:

0 - Erfolg,
< 0 - Fehler (kein Layout, ungültige Position)

Siehe auch:

layout_getchar(), layout_setattr()

Hinweis:

c wird nicht auf Gültigkeit geprüft.

17 Paintarea

pa_declare — Erstellen einer Paintarea

Definition:

HPAINTAREA pa_declare(pa)
paintarea *pa;

Beschreibung:

Diese Funktion erstellt eine Paintarea aus der Struktur **pa*. Die Struktur ist in der Header-Datei `fix/fixwin.h` definiert. Der Bereich, den die Paintarea belegt, wird im virtuellen Bildschirm mit dem Attribut PAINTAREA markiert. Die Paintarea-Beschreibung wird an *FIX/Win* übertragen.

Ergebnis:

Die Funktion gibt bei Erfolg ein Handle auf die erstellte Paintarea zurück. Im Fehlerfall wird der Wert des Makros HPAINTAREA_INVALID zurückgegeben.

Siehe auch:

pa_put()

Hinweis:

Vgl. “Programmieren mit Paintareas” auf Seite 219.

pa_declare_type — Erstellen einer Paintarea

Definition:

HPAINTAREA pa_declare_type(type, objp, pos_y, pos_x, token, text, tooltiptext, attr, align, bt_mask,
longval1, longval2, longval3, longval4, ptrval1, ptrval2)
short type; obj *objp; int pos_y; int pos_x; char *token; char *text; char *tooltiptext; int attr; int align; int bt_mask; long longval1; long longval2; long longval3; long longval4; char *ptrval1; char *ptrval2;

Beschreibung:

Diese Funktion bekommt statt eines Zeigers auf eine paintarea Struktur die einzelnen Werte. Sie erzeugt dann eine Struktur und kopiert die Werte in die entsprechenden Komponenten. Die Länge der Paintarea (pa_len) wird dabei mit der Länge der Zeichenkette *token* belegt. Danach wird die Struktur als Parameter für einen Aufruf von pa_declare() verwendet.

Ergebnis:

Die Funktion gibt bei Erfolg ein Handle auf die eben erstellte Paintarea zurück. Im Fehlerfall wird der Wert des Makros HPAINTAREA_INVALID zurückgegeben.

Siehe auch:

pa_put(), pa_declare(), pa_put_type()

Hinweis:

Vgl. “Programmieren mit Paintareas” auf Seite 219.

pa_delete — Löschen einer Paintarea

Definition:

BOOLEAN pa_delete(h_pa)
HPAINTAREA h_pa;

Beschreibung:

Zum Löschen einer Paintarea ist die Funktion pa_delete() aufzurufen. Sie löscht die Paintarea, indem sie den entsprechenden Bereich mit Leerzeichen im Attribut NORMAL überschreibt.

Ergebnis:

Bei Erfolg liefert sie TRUE zurück. Wenn h_pa keinem gültigen Cache-Eintrag zugeordnet ist, dann liefert die Funktion FALSE zurück.

Hinweis:

Vgl. “Programmieren mit Paintareas” auf Seite 219.

pa_dump_all - Daten von Paintareas ausgeben

Definition:

void pa_dump_all(fout)
FILE *fout;

Beschreibung:

Gibt die Daten der Paintareas aller Objekte auf dem Ausgabe-Kanal *fout* aus.

Vgl. “Ausgabe von Informationen zu Debugzwecken” auf Seite 228.

pa_dump_for_obj - Daten der Paintareas eines Objektes ausgeben

Definition:

void pa_dump_for_obj(fout, objp)
FILE *fout; obj *objp;

Beschreibung:

Gibt die Daten der Paintareas des Objektes *objp* auf dem Ausgabe-Kanal *fout* aus.

Vgl. “Ausgabe von Informationen zu Debugzwecken” auf Seite 228.

pa_get — Beschaffen von Informationen zu einer Paintarea

Definition:

BOOLEAN pa_get(h_pa, pa)
HPAINTAREA h_pa; paintarea *pa;

Beschreibung:

Durch Aufruf der Funktion können die Werte der Paintarea, die durch das Paintarea-Handle h_pa identifiziert wird, aus dem Cache-Eintrag gelesen werden. pa zeigt auf eine Struktur vom Typ paintarea, in der die Informationen abgelegt werden.

Ergebnis:

Die Funktion gibt bei Erfolg TRUE zurück. Ist der zugeordnete Cache-Eintrag ungültig geworden, weil das Objekt vom Bildschirm entfernt wurde, dann liefert die Funktion FALSE.

Siehe auch:

pa_put(), pa_declare(), pa_put_type()

Hinweis:

Es ist möglich, dass die Funktion TRUE zurückliefert, obwohl das Objekt vom Bildschirm entfernt wurde. Dieser Fall kommt dann vor, wenn der Cache-Eintrag mittlerweile durch ein anderes Element belegt wird. Handles dürfen nach dem Entfernen des Objekts vom Bildschirm nicht mehr benutzt werden, da sie *FIX*-intern wiederverwendet werden. Vgl. “Programmieren mit Paintareas” auf Seite 219.

pa_get_clicked — Ermitteln der angeklickten Paintarea

Definition:

HPAINTAREA pa_get_clicked()

Beschreibung:

Die Anwendung kann dann mit Hilfe der Funktion pa_get_clicked() ein Handle der Paintarea ermitteln, die angeklickt wurde.

Ergebnis:

Ein Handle auf die zuletzt angeklickte Paintarea bei Erfolg. Wurde keine Paintarea angeklickt wird HPAINTAREA_EMPTY zurückgegeben. Konnte die Paintarea nicht ermittelt werden, gibt die Funktion HPAINTAREA_INVALID zurück.

set_activeobj_list — Paintareas anderer Objekte auf anklickbar schalten

Definition:

void set_activeobj_list(objp, obj_list)
obj *objp; obj **obj_list;

Beschreibung:

Da weder *FIX* noch *FIX/Win* den Aufbau der Anwendung kennen und wissen, ob sie das Anklicken von Paintareas in nicht aktiven Masken behandeln, muss die Anwendung *FIX/Win* mitteilen, in welchen Masken Paintareas angeklickt werden können. Dazu ist die Funktion set_activeobj_list() aufzurufen. Sie definiert für das Objekt *objp* eine Liste von zusätzlichen Objekten, in denen Paintareas anklickbar sind, wenn die Maske *objp* aktiv ist. Die Liste ist als Array in *obj_list* zu übergeben. Das letzte Element des Arrays muss als Endemarkierung den Wert NULL besitzen.

Hinweis:

Es ist zu beachten, dass *FIX* sich den übergebenen Zeiger auf das Array zu dem Objekt merkt und wiederverwendet. Das Array sollte auf keinen Fall als automatische Variable auf dem Stack angelegt werden oder für andere Objekte umdefiniert werden.

pa_next — Beschaffen von Informationen zu einer Paintarea

Definition:

HPAINTAREA pa_next(h_paStart, pa_check, validmask)
HPAINTAREA h_paStart; paintarea *pa_check; unsigned long validmask;

Beschreibung:

Wenn das Handle zu einem Eintrag nicht bekannt ist, dann kann es durch Aufruf von pa_next() ermittelt werden. Dazu können alle Handles eines Objektes ermittelt werden, die einem bestimmten Suchmuster in *pa_check* entsprechen. Es fließen nur die Werte von *pa_check* in die Suche ein, die durch den Wert von *validmask* dazu bestimmt werden. Folgende Werte und deren Verknüpfungen sind für *validmask* zulässig:

- PA_TYPE - Der Wert in *pa_check*->pa_type wird auf Übereinstimmung geprüft.
- PA_POS - Die Position wird auf Überlappung geprüft. Der in Frage kommende Eintrag muss in seinem Bereich eine Zelle besitzen, die an der Position *pa_check*->pa_pos_y, *pa_check*->pa_pos_x liegt.
- PA_TOKEN - Der Wert in *pa_check*->pa_token wird auf Übereinstimmung geprüft.
- PA_ATTR - Der Wert in *pa_check*->pa_attr wird auf Übereinstimmung geprüft.
- PA_ALIGN - Der Wert in *pa_check*->pa_align wird auf Übereinstimmung geprüft.
- PA_BT_MASK - Der Wert in *pa_check*->pa_bt_mask wird auf Übereinstimmung geprüft.
- PA_LONGVAL1 - Der Wert in *pa_check*->pa_props.longval1 wird auf Übereinstimmung geprüft.
- PA_LONGVAL2 - Der Wert in *pa_check*->pa_props.longval2 wird auf Übereinstimmung geprüft.
- PA_LONGVAL3 - Der Wert in *pa_check*->pa_props.longval3 wird auf Übereinstimmung geprüft.

- PA_LONGVAL4 - Der Wert in *pa_check*->pa_props.longval4 wird auf Übereinstimmung geprüft.
- PA_PTRVAL1 - Der Wert in *pa_check*->ptrval1 wird auf Übereinstimmung geprüft.
- PA_PTRVAL2 - Der Wert in *pa_check*->ptrval2 wird auf Übereinstimmung geprüft.

Soll auf keinen der Werte geprüft werden, dann ist PA_NONE als Wert für den Parameter *validmask* zu verwenden. Es werden nur Paintareas berücksichtigt, die sich in der aktuellen (und damit zur Zeit angezeigten) Variante des Objektes in *pa_check*->pa_objp befinden.

Wenn zu Beginn kein Handle für *h_paStart* bekannt ist, kann HPAINTAREA_EMPTY angegeben werden. In diesem Fall beginnt die Suche bei dem ersten Cache-Eintrag des Objekts. Wird ein konkreter Wert für *h_paStart* angegeben, dann beginnt die Suche hinter dem Eintrag, der durch *h_paStart* definiert wird.

Ergebnis:

Wird ein passender Eintrag gefunden, dann wird das Handle zu diesem Eintrag zurückgeliefert. Bei einem Fehler liefert die Funktion HPAINTAREA_INVALID zurück. Wenn kein weiterer Eintrag gefunden wird, der zu den Bedingungen passt, dann wird HPAINTAREA_EMPTY zurückgeliefert.

Hinweis:

Vgl. “Programmieren mit Paintareas” auf Seite 219.

pa_put — Erstellen einer Paintarea

Definition:

HPAINTAREA pa_put(pa);
paintarea *pa;

Beschreibung:

Diese Funktion erstellt eine Paintarea aus der Struktur **pa*. Die Struktur ist in der Header-Datei `fix/fixwin.h` definiert. Das Token wird in das Layout des Objektes eingetragen und mit dem Attribut PAINTAREA markiert. Die Paintarea-Beschreibung wird an *FIX/Win* übertragen.

Ergebnis:

Die Funktion gibt bei Erfolg ein Handle auf die eben erstellte Paintarea zurück. Im Fehlerfall wird der Wert des Makros HPAINTAREA_INVALID zurückgegeben.

Siehe auch:

pa_declare()

Hinweis:

Vgl. “Programmieren mit Paintareas” auf Seite 219.

pa_put_type — Erstellen einer Paintarea

Definition:

HPAINTAREA pa_put_type(type, objp, pos_y, pos_x, token, text, tooltiptext, attr, align, bt_mask, longval1, longval2, longval3, longval4, ptrval1, ptrval2)
short type; obj *objp; int pos_y; int pos_x; char *token; char *text; char *tooltiptext; int attr; int align; int bt_mask; long longval1; long longval2; long longval3; long longval4; char *ptrval1; char *ptrval2;

Beschreibung:

Diese Funktion bekommt statt eines Zeigers auf eine paintarea Struktur die einzelnen Werte. Sie erzeugt dann eine Struktur und kopiert die Werte in die entsprechenden Komponenten. Die Länge der Paintarea (pa_len) wird dabei mit der Länge der Zeichenkette *token* belegt. Danach ruft sie pa_put() auf.

Ergebnis:

Die Funktion gibt bei Erfolg ein Handle auf die eben erstellte Paintarea zurück. Im Fehlerfall wird der Wert des Makros HPAINTAREA_INVALID zurückgegeben.

Siehe auch:

pa_put(), pa_declare(), pa_declare_type()

Hinweis:

Vgl. “Programmieren mit Paintareas” auf Seite 219.

pa_update — Aktualisieren einer Paintarea

Definition:

BOOLEAN pa_update(h_pa, pa, validmask)
HPAINTAREA h_pa; paintarea *pa; unsigned long validmask;

Beschreibung:

Durch Aufruf der Funktion pa_update() können die Daten einer Paintarea aktualisiert werden. In *h_pa* ist dazu das Handle des Cache-Eintrags anzugeben. *pa* zeigt auf eine Struktur mit den neuen Daten. Der Wert in *validmask* bestimmt, welche Daten aus der Struktur in die Paintarea übernommen werden. Er ist durch eine oder-Verknüpfung aus den folgenden Werten zu bilden:

- PA_TYPE - Der Typ wird auf *pa*->type geändert.
- PA_TOKEN - Der Wert in *pa*->pa_token wird als neuer Kurztext verwendet. Dadurch ändert sich jedoch nicht automatisch der übersetzte Text. Es wird der bei pa_declare()/pa_put() festgesetzte Text verwendet. Wenn der übersetzte Text geändert werden soll, dann ist zusätzlich der Wert PA_TEXT anzugeben.
- PA_TEXT - Der Wert in *pa*->pa_text wird als neuer Text verwendet. Beinhaltet die Komponente den Wert NULL, dann wird der Text durch Aufruf der Übersetzungsfunktion ermittelt.
- PA_TOOLTIPTEXT - Der Wert in *pa*->pa_tooltiptext wird als neuer Text für den Tooltip verwendet.
- PA_ATTR - Der Wert in *pa*->pa_attr wird als neues Bildschirmattribut verwendet.
- PA_ALIGN - Der Wert in *pa*->pa_align wird übernommen.
- PA_BT_MASK - Der Wert in *pa*->pa_bt_mask wird übernommen.
- PA_LONGVAL1 - Der Wert in *pa*->pa_props.longval1 wird übernommen.
- PA_LONGVAL2 - Der Wert in *pa*->pa_props.longval2 wird übernommen.
- PA_LONGVAL3 - Der Wert in *pa*->pa_props.longval3 wird übernommen.
- PA_LONGVAL4 - Der Wert in *pa*->pa_props.longval4 wird übernommen.
- PA_PTRVAL1 - Der Wert in *pa*->ptrval1 wird übernommen.
- PA_PTRVAL2 - Der Wert in *pa*->ptrval2 wird übernommen.

Nach der Übernahme der Werte in den Cache-Eintrag, wird geprüft, ob das Layout und der virtuelle Bildschirm von der Änderung betroffen sind. Dies ist bei der Angabe von PA_TOKEN oder PA_ATTR der Fall. Die geänderten Werte werden dann ins Layout und in den virtuellen Bildschirm übernommen. Bei der Angabe von PA_TYPE, PA_TOKEN, PA_TEXT, PA_ALIGN, PA_TOOLTIPTEXT oder PA_LONGVAL1, PA_LONGVAL2, PA_LONGVAL3, PA_LONGVAL4 müssen die neuen Werte auch an *FIX/Win* übertragen werden. Bei der Angabe von PA_TEXT wird zur Bestim-

mung des Textes die Übersetzungsfunktion aufgerufen, wenn eine solche definiert wurde und der übersetzte Text nicht in der Komponente *pa_text* mitgegeben wurde.

Ergebnis:

Die Funktion gibt bei Erfolg TRUE zurück. Ist ein Fehler aufgetreten, dann liefert die Funktion FALSE.

Hinweis:

Vgl. “Programmieren mit Paintareas” auf Seite 219.

18 Meldungen

clrmsgln — Message-Window löschen und verschwinden lassen

Definition:

void clrmsgln()

Beschreibung:

löscht den Inhalt des Message-Windows und bringt es ggf. zum Verschwinden.

fflash — Meldung ausgeben (ohne Positionierung und Quittung)

Definition:

void fflash(type, fmt, …)
int type; char *fmt;

Beschreibung:

präsentiert eine kurzlebige Meldung im Message-Window. Kurzlebig deshalb, weil *FIX* sie löscht, sobald die nächste Eingabe gelesen worden ist.

type steuert Vortext und Warnton, *fmt* ist ein Formatstring im Sinne von printf(), die weiteren Parameter Argumente für die in *fmt* enthaltenen Formatelemente.

Siehe auch:

oflash()

Hinweis:

Die Position der Schreibmarke wird nach Ausgabe der Meldung wieder restauriert.

findkey - Ermittelt den Eventcode zu einer zweistelligen Tastenbeschreibung

Definition:

Event findkey(key)
char *key;

Beschreibung:

Die Funktion ermittelt den Event zu einer Tastenbeschreibung. Für den Parameter *key* ist die zweistellige Bezeichnung der Taste, so wie sie in der Meldungsdatei hinter dem # verwendet wird, zu übergeben.

Ergebnis:

Die Event-Repräsentation der Taste (Logical Key), oder 0, wenn es keine Event-Repräsentation des zweistelligen Tastenbezeichners gibt.

Siehe auch:

get_keyhelp_text()

flash — Meldung ausgeben (mit Positionierung, ohne Quittung)

Definition:

int flash(type, txt, rzeile, rspalte)
int type; char *txt; int rzeile; int rspalte;

Beschreibung:

Ähnlich wie fflash(), allerdings kann die Position am Bildschirm angegeben werden, an die die Schreibmarke platziert werden soll, bis Eingabe erfolgt. Diese Eingabe wird von der Funktion gelesen, aber sofort wieder in den Eingabe-Puffer zurückgestellt und das Message-Window anschließend zum Verschwinden gebracht.

type steuert Vortext und Warnton, *txt* ist der auszugebende Text.

Ergebnis:

das Event, das gelesen wurde

Siehe auch:

fflash(), move()

Hinweis:

Vorsicht: Bei der Eingabe kann es sich auch um zurückgestellte Zeichen handeln. Sofern nur die Positionierung erforderlich ist, bietet sich als Alternative an:

move(*rzeile*, *rspalte*); fflash(*art*, *txt*);

fxlapptxt — Zeiger auf anwendungsbezogenen Meldungstext ermitteln

Definition:

char *fxlapptxt(n)
long n;

Beschreibung:

liefert einen Zeiger auf den Text, der unter der Nummer *n* in `$FXHOME/messages` abgelegt ist. Der Text wird beim ersten Zugriff in den Speicher geladen.

Ergebnis:

Zeiger auf den zur Nummer *n* gehörigen Text, oder, wenn der Text nicht gelesen werden kann oder nicht existiert, auf einen (statischen) Leerstring.

Siehe auch:

Makro fxapptxt()

Hinweis:

Aufrufe mit derselben Nummer liefern dieselbe Adresse.

fxlsystxt — Zeiger auf *FIX*-Meldungstext ermitteln

Definition:

char *fxlsystxt(n)
long n;

Beschreibung:

liefert einen Zeiger auf den Text, der unter der Nummer *n* in `$FXDIR/runtime/messages` abgelegt ist. Der Text wird beim ersten Zugriff in den Speicher geladen.

Ergebnis:

Zeiger auf den zur Nummer *n* gehörigen Text, oder, wenn der Text nicht gelesen werden kann oder nicht existiert, auf einen (statischen) Bereich mit dem Inhalt “fix system text *n*”.

Siehe auch:

fxlapptxt(), Makro fxsystxt()

Hinweis:

Aufrufe mit derselben Nummer liefern dieselbe Adresse.

Diese Funktion wird in generierten Programmen verwendet; sonst sollte der Entwickler nicht auf *FIX*-Meldungstexte zugreifen, da diese stark versionsabhängig sind.

get_keyhelp_text — Ermitteln des Textes, der von der Tastenhilfe angezeigt wird

Definition:

char *get_keyhelp_text();

Beschreibung:

Der Text der Tastenhilfe kann mit Hilfe dieser Funktion ausgelesen werden.

Ergebnis:

Sie liefert einen Zeiger auf den gerade angezeigten Meldungstext, der je nach Nummer des Prompttextes mit der Funktion fxlsystxt() oder fxlapptxt() ermittelt wurde, oder, wenn der Text nicht gelesen werden kann oder nicht existiert, auf einen (statischen) Leerstring.

Siehe auch:

fxlapptxt(), Makro fxsystxt()

Hinweis:

Enthält der Text Tastenlabels, dann haben diese folgendes Format:

```
\020\024(xx)\025\021
```

Die oktal angegebenen Zeichen sind Steuerzeichen für den Bildschirm. \020 speichert den aktuellen Zustand des Bildschirms. \021 stellt ihn wieder her. \024 markiert den Start des Tastenlabels. \025 markiert das Ende. Zwischen den Klammern steht die Eventbezeichnung des Tastenlabels (xx).

jn_msg — im Message-Window Frage stellen

Definition:

BOOLEAN jn_msg(type, fmt, …)
int type; char *fmt;

Beschreibung:

wie msg(), hinter dem Meldungstext erscheint jedoch ein Fragezeichen und die Meldung muss mit einem der zur Darstellung von S_yes oder S_no hinterlegten Zeichen quittiert werden.

Ergebnis:

TRUE bei S_yes
FALSE bei S_no

Siehe auch:

msg(), rt_msg()

Hinweis:

Auch die den für S_yes und S_no hinterlegten Kleinbuchstaben entsprechenden Großbuchstaben werden akzeptiert.

l_msg — langlebige Meldung ausgeben (ohne Positionierung und Quittung)

Definition:

void l_msg(type, fmt, …)
int type; char *fmt;

Beschreibung:

präsentiert eine langlebige Meldung im Message-Window. Langlebig deshalb, weil sie nicht nach dem Lesen der nächsten Eingabe verschwindet wie bei oflash(). Die Meldung wird erst durch eine nachfolgende Meldung oder durch Aufruf von clrmsgln() zum Verschwinden gebracht.

Siehe auch:

oflash(), msg()

msg — Meldung ausgeben (mit Quittung)

Definition:

Event msg(type, fmt, …)
int type; char *fmt;

Beschreibung:

präsentiert eine Meldung im Message-Window und wartet anschließend auf Eingabe. Nach dem Lesen der Eingabe wird die Meldung zum Verschwinden gebracht.

type steuert Vortext und Warnton, *fmt* ist ein Formatstring im Sinne von printf(), die weiteren Parameter sind Argumente für die in *fmt* enthaltenen Formatelemente.

Ergebnis:

das Event, das gelesen wurde

Siehe auch:

rt_msg(), jn_msg()

Hinweis:

Vorsicht: Bei der Eingabe kann es sich auch um zurückgestellte Zeichen handeln.

oflash — Meldung ausgeben (ohne Positionierung und Quittung)

Definition:

void oflash(objp, type, fmt, …)
obj *objp; int type; char *fmt;

Beschreibung:

präsentiert eine kurzlebige Meldung im Message-Window. Kurzlebig deshalb, weil *FIX* sie löscht, sobald die nächste Eingabe gelesen worden ist.

objp gibt an, auf welches Objekt sich die Meldung bezieht, *type* steuert Vortext und Warnton, *fmt* ist ein Formatstring im Sinne von printf(), die weiteren Parameter Argumente für die in *fmt* enthaltenen Formatelemente.

Siehe auch:

l_msg()

Hinweis:

Die Position der Schreibmarke wird nach Ausgabe der Meldung wieder restauriert.

rt_msg — Meldung ausgeben (mit Quittung)

Definition:

Event rt_msg(art, fmt, …)
int art; char *fmt;

Beschreibung:

wie msg(), hinter dem Meldungstext erscheint jedoch die Beschriftung der Taste RT und die Meldung muss durch Eingabe von RT quittiert werden.

Ergebnis:

das Event K_RT

Siehe auch:

msg(), jn_msg()

Hinweis:

Diese Funktion kann auch benutzt werden, wenn die Bildschirmverwaltung nicht aktiv ist, und entspricht dann

```
fprintf(stderr, fmt, txt2, …);
if (isatty(fileno(stdin))) while (c=getc(stdin)) != EOF && c != '\n') ;
```

show_keyhelp —Anzeige der Tastenhilfe umschalten

Definition:

void show_keyhelp(on);
BOOLEAN on;

Beschreibung:

Diese Funktion kann die Anzeige der Tastenhilfe anschalten (Wert für on = TRUE) oder ausschalten (Wert für on = FALSE). Sie darf nicht bei Hintergrundprozessen benutzt werden, da sie den virtuellen Bildschirm modifiziert.

19 Fehlerbehandlung

fx_SC — Fehlercode umsetzen

Definition:

int fx_SC(code, op)
long code; int op;

Beschreibung:

dient dazu, den Fehlercode *code* (in der Regel den Wert von SQLCODE) in einen vom Datenbanksystem unabhängigen Code umzusetzen:

code	Ergebnis
0	SUCCESS
SQLNOTFOUND	FX_SQLNOTFOUND
FCT_NOT_IMPLEMENTED	FCT_NOT_IMPLEMENTED
SQLLOCKED (-233)	FX_SQLLOCKED
-243	FX_SQLLOCKED
-244	FX_SQLLOCKED
-245	FX_SQLLOCKED
-246	FX_SQLLOCKED
SQLUPDLOCK (-250)	FX_SQLLOCKED
sonst	IOERROR

Ergebnis:

normierter Fehlercode

Hinweis:

Das Argument *op* wird nicht ausgewertet.

Wird in Source ausgeliefert (`$FXDIR/src`).

fx_sql_error — Fehler in fx_sql() melden †

Definition:

void fx_sql_error(errcode, cursornr, fct, txt)
long errcode; int cursornr; int fct; char *txt;

Beschreibung:

wird von *FIX* - indirekt über Makros - benutzt, um Fehler in der Funktion fx_sql() zu melden. Die letzten drei Parameter entsprechen i. Allg. denen der Funktion fx_sql().

- *errcode* == 0 (Makro gibts_nicht):
  die Operation *fct* ist für den Cursor Nr. *cursornr* nicht definiert.
- *errcode* < 0 (Makro FX_SQL_MSG):
  nach Abarbeitung der Operation *fct* enthält SQLCODE einen Wert < 0.

Hinweis:

Wird in Source ausgeliefert (`$FXDIR/src`).

In mit Versionen bis 2.8 generierten Programmen übernahm die Funktion fx_sql() die Aufgabe, Sätze aus der Datenbank zu lesen oder sie darin zurückzuschreiben.

fx_sql_undef — Fehler in fx_sql() melden †

Definition:

void fx_sql_undef(cursornr, fct, txt)
int cursornr; int fct; char *txt;

Beschreibung:

Mit dieser Funktion meldet *FIX* in der Funktion fx_sql() eine nicht vorgesehene Operation; aufgerufen wird fx_sql_error().

Siehe auch:

fx_sql_error()

Hinweis:

In mit Versionen bis 2.8 generierten Programmen übernahm die Funktion fx_sql() die Aufgabe, Sätze aus der Datenbank zu lesen oder sie darin zurückzuschreiben.

sql_error — ESQL/C-Fehler melden

Definition:

void sql_error(sqlcode, txt)
long sqlcode; char *txt;

Beschreibung:

wird von *FIX*, insbesondere der Funktion fx_sql_test(), benutzt, um Fehler bei der Abarbeitung von ESQL/C-Anweisungen zu melden (fx_database_connect(), fx_commit_transaction() etc.).

Parameter:

sqlcode: i. Allg. gleich SQLCODE
txt: Text für die Fehlermeldung

Siehe auch:

fx_sql_test()

Hinweis:

Wird in Source ausgeliefert (`$FXDIR/src`).

20 E/A

beep — akustisches Signal geben

Definition:

void beep()

Beschreibung:

erzeugt ein akustisches Signal.

blink — Videoattribut BLINK einstellen

Definition:

void blink()

Beschreibung:

stellt das Video-Attribut BLINK ein.

Siehe auch:

standend(), putstr()

clear — Bildschirm löschen

Definition:

void clear()

Beschreibung:

setzt alle Zeichen des (virtuellen) Bildschirms auf ' ' und bewegt die Schreibmarke in die linke obere Ecke.

Hinweis:

Für die Ausgabe wird stets das Videoattribut NORMAL benutzt.

Siehe auch:

clrline(), clrtoeol()

clrline — Bildschirmzeile löschen

Definition:

void clrline()

Beschreibung:

setzt alle Zeichen der Zeile, in der sich die Schreibmarke befindet, auf ' ' und bewegt die Schreibmarke nach Spalte 0.

Hinweis:

Für die Ausgabe wird stets das Videoattribut NORMAL benutzt.

Siehe auch:

clear(), clrtoeol()

clrtoeol — Bildschirmzeile ab Schreibmarke löschen

Definition:

void clrtoeol()

Beschreibung:

clrtoeol() setzt alle Zeichen ab der Position der Schreibmarke bis zum Ende der Zeile auf ' '.

Hinweis:

Für die Ausgabe wird stets das Videoattribut NORMAL benutzt.

Siehe auch:

clear(), clrline()

fx_discard_input — anstehende Eingabe verwerfen

Definition:

void fx_discard_input()

Beschreibung:

verwirft (bei Terminalbetrieb: am Eingabekanal 0) anstehende Eingabe.

Siehe auch:

Systemroutinen ioctl(), tcflush()

Hinweis:

Die Funktion ist gelegentlich sinnvoll, um nach einem Programmwechsel das Lesen vorab getippter Zeichen zu verhindern.

getpos — Position der Schreibmarke ermitteln

Definition:

int getpos(yadr, xadr)
int *yadr; int *xadr;

Beschreibung:

hinterlässt die Position der Schreibmarke des virtuellen Bildschirms in **yadr* und **xadr*.

Ergebnis:

TRUE bei Erfolg
FALSE sonst

Siehe auch:

move()

gr_end — in Standard-Zeichensatz umschalten

Definition:

void gr_end()

Beschreibung:

schaltet den *FIX*-Standard-Zeichensatz ein.

Siehe auch:

gr_start(), putstr()

gr_start — in Grafik-Zeichensatz umschalten

Definition:

void gr_start()

Beschreibung:

schaltet den *FIX*-Grafik-Zeichensatz ein.

Siehe auch:

gr_end(), putstr()

lowint — Videoattribut LOW einstellen

Definition:

void lowint()

Beschreibung:

stellt das Video-Attribut LOW ein.

Siehe auch:

standend(), putstr()

move — Schreibmarke positionieren

Definition:

void move(y, x)
int y; int x;

Beschreibung:

setzt die Schreibmarke des virtuellen Bildschirms auf Zeile *y*, Spalte *x*.

Siehe auch:

moverelm()

Hinweis:

Die Positionierung wird erst durch eine Bildschirmaktualisierung (→ sync_refresh()) sichtbar.

moverelm — Schreibmarke relativ zu Objekt positionieren

Definition:

void moverelm(objp, relz, rels)
obj *objp; int relz; int rels;

Beschreibung:

positioniert die Schreibmarke auf die Koordinaten *relz* und *rels* relativ zum Objekt *objp*.

Siehe auch:

move()

moveright — Schreibmarke positionieren †

Definition:

void moveright()

Beschreibung:

bewegt die Schreibmarke um eine Stelle nach rechts.

Siehe auch:

move(), putstr()

putstr — *FIX*-String ausgeben

Definition:

```
void putstr(str)
char *str;
```

Beschreibung:

schreibt den String *str* an die Stelle, an der sich die Schreibmarke befindet. Um innerhalb des Strings Videoattribut und Zeichensatz wechseln zu können, interpretiert putstr() folgende nicht darstellbare Zeichen:

'\001' - eingeschleustes standend(): NORMAL
'\002' - eingeschleustes standout(): INVERS (invertiert)
'\003' - eingeschleustes underln(): UNDERLINE (unterstrichen)
'\004' - eingeschleustes lowint(): LOW (halbhell)
'\005' - eingeschleustes blink(): BLINK (blinkend)
'\016' - Grafik-Zeichensatz
'\017' - Standard-Zeichensatz
'\020' - eingeschleustes save_video(): Videoattribut und Zeichensatz merken
'\021' - eingeschleustes restore_video(): Videoattribut und Zeichensatz wiederherstellen

Ebenfalls interpretiert werden:

'\b' - Schreibmarke eine Position nach links
'\r' - Schreibmarke zum Anfang der Zeile
'\n' - Schreibmarke zum Anfang der nächsten Zeile
'\t' - Schreibmarke zum nächsten Tabulator
'\f' - Schreibmarke in die linke obere Ecke des Bildschirms

putstr() ignoriert die nicht darstellbaren Zeichen '\024' (DC4) und '\025' (NAK).

Siehe auch:

sync_refresh(), putstrwidth()

Hinweis:

Die Ausgabe wird erst durch eine Bildschirmaktualisierung (→ sync_refresh()) sichtbar.

refresh — Bildschirminhalt ausgeben

Definition:

```
void refresh()
```

Beschreibung:

entspricht sync_refresh("refresh", FALSE).

Siehe auch:

sync_refresh(), touchscreen()

restore_video — Videoattribut und Zeichensatz restaurieren

Definition:

```
int restore_video()
```

Beschreibung:

setzt das Videoattribut und den Zeichensatz, die durch den letzten Aufruf von save_video() gespeichert wurden.

Ergebnis:

Kodierung für die neu gesetzten Werte

Siehe auch:

save_video(), putstr()

Hinweis:

keine Schachtelung möglich.

save_video — Videoattribut und Zeichensatz speichern

Definition:

int save_video()

Beschreibung:

speichert das aktuelle Videoattribut und den Zeichensatz in einem statischen Bereich, der bei jedem Aufruf überschrieben wird.

Ergebnis:

interne Kodierung für die gespeicherten Werte.

Siehe auch:

restore_video(), putstr()

standend — Videoattribut NORMAL einstellen

Definition:

void standend()

Beschreibung:

stellt das Video-Attribut NORMAL ein.

Siehe auch:

putstr()

standout — Videoattribut INVERS einstellen

Definition:

void standout()

Beschreibung:

stellt das Video-Attribut INVERS ein.

Siehe auch:

standend(), putstr()

sync_refresh — Bildschirminhalt ausgeben

Definition:

void sync_refresh(char *dbgmsg, BOOLEAN force)

Beschreibung:

gleicht i.d.R den Terminal-Bildschirm an den virtuellen Bildschirm an, d.h. macht die seit dem vorausgegangenen Aufruf erfolgte Ausgabe sichtbar. Es werden nur die Positionen am Bildschirm überschrieben, die seit der letzten Bildschirmanzeige verändert wurden.

Der Parameter *force* bezeichnet die Wichtigkeit der Bildschirmaktualisierung: bei Aufrufen mit FALSE kann die Bildschirmaktualisierung unterdrückt werden.

Siehe auch:

refresh(), touchscreen()

touchscreen — vollständigen Bildschirmneuaufbau erzwingen

Definition:

void touchscreen()

Beschreibung:

informiert *FIX*, dass beim nächsten Anzeigen des Bildschirminhalts der Bildschirm vollständig neu aufgebaut werden soll.

Siehe auch:

sync_refresh()

Hinweis:

Die Funktion kann beispielsweise benutzt werden, wenn der Bildschirm unter Umgehung der Bildschirm-Verwaltung verändert wurde.

tty_clear — Bildschirm (phys.) löschen

Definition:

void tty_clear()

Beschreibung:

löscht *unter Umgehung der Bildschirmverwaltung* Bildschirm (Termcap-Einträge cm, cl) und Statuszeile (Termcap-Einträge sl, ce).

Siehe auch:

tty_clrtoeol()

tty_clrtoeol — Bildschirmzeile hinter (phys.) Cursorposition löschen

Definition:

void tty_clrtoeol()

Beschreibung:

löscht *unter Umgehung der Bildschirmverwaltung* die Zeile hinter der Position des Cursors (Termcap-Eintrag ce).

Siehe auch:

tty_clear()

tty_move — (phys.) Cursor positionieren

Definition:

```
void tty_move(y, x)
int y; int x;
```

Beschreibung:

bewegt *unter Umgehung der Bildschirmverwaltung* den Cursor nach Zeile *y* in Spalte *x* (Termcap-Einträge cm, sl).

underln — Videoattribut UNDERLINE einstellen

Definition:

```
void underln()
```

Beschreibung:

stellt das Video-Attribut UNDERLINE ein.

Siehe auch:

standend(), putstr()

writetty — Text an angegebene Position schreiben

Definition:

```
void writetty(y, x, str, clflg)
int y; int x; char *str; int clflg;
```

Beschreibung:

schreibt *unter Umgehung der Bildschirmverwaltung* den String *str* in Spalte *y*, Zeile *x*. Ist *clflg* von 0 verschieden, wird zuvor die Zeile gelöscht. Anschließend wird der Cursor an die Position bewegt, wo *FIX* ihn zuvor vermutete, und touchscreen() aufgerufen.

Hinweis:

Die Funktion eignet sich für kurze Anzeigen, die beim nächsten Neuaufbau des Bildschirms verschwinden können. *str* sollte keine nicht darstellbaren Zeichen enthalten.

21 Frontend

frontend_show_caret — Schreibmarke anzeigen

Definition:

```
void frontend_show_caret(on)
int on;
```

Beschreibung:

Die Funktion kann aufgerufen werden, um die Schreibmarke unter *FIX/Win* explizit anzuschalten (on==TRUE) oder auszuschalten (on==FALSE).

fx_client_id — Frontend-Arbeitsplatz identifizieren

Definition:

char *fx_client_id()

Beschreibung:

identifiziert den Arbeitsplatz, an dem die Ein- und Ausgabe für das Programm erfolgt, durch einen String von höchstens 15 Zeichen. Voraussetzung ist, dass zuvor ein Aufruf von fxinit() erfolgt ist.

Ergebnis:

Zeiger auf statischen Bereich mit einer Zeichenfolge, die den Arbeitsplatz identifiziert
(char *)0 bei Fehler (z.B. Programm nicht durch Frontend gestartet oder Frontend noch im “Textmodus”)

Hinweis:

Ein Aufruf, der einen Fehler zurückgibt, ist ansonsten ohne Folgen.

Der String beinhaltet gegenwärtig die Internet-Adresse des *FIX/Win*-Rechners bzw. Rechners, auf dem das JAVA-Applet läuft, in ‘.’-Notation.

Die Funktion kann anstelle der C-Routine ttyname() genutzt werden, die bei durch ein Frontend gestarteten Programmen (char *)0 liefert.

fx_client_type — Art des Frontends ermitteln

Definition:

int fx_client_type()

Beschreibung:

liefert einen numerischen Wert (Makro aus `fix/basics.h`), der das Frontend kennzeichnet. Voraussetzung ist, dass zuvor ein Aufruf von fxinit() erfolgt ist.

Ergebnis:

FRONT_FIXWIN - *FIX/Win*
FRONT_FIXWEB - *FIX/Web*
0 - kein Frontend oder Fehler (z.B. kein Aufruf von fxinit() vorausgegangen)

fx_clienthost_id — Frontend-Host identifizieren

Definition:

char *fx_clienthost_id()

Beschreibung:

liefert einen Zeiger auf einen statischen Bereich mit einem String der Form “xxx.xxx.xxx.xxx”, der die Internet-Adresse des Hosts enthält, auf dem das Frontend abläuft. Voraussetzung ist, dass zuvor ein Aufruf von fxinit() erfolgt ist.

Ergebnis:

String der Form “xxx.xxx.xxx.xxx” - Internet-Adresse
Leerstring - Fehler

Siehe auch:

fx_client_id()

Hinweis:

Ein Aufruf, der einen Fehler zurückgibt, ist ansonsten ohne Folgen.

fx_exec_frontend — Anweisung für Frontend senden

Definition:

int fx_exec_frontend(str)
char *str;

Beschreibung:

sendet den String *str* an das Frontend. Dieses ruft bei Empfang eine vom Entwickler bereitgestellte, dynamisch eingebundene Funktion mit dem Argument *str* auf, die den String interpretiert (in der Regel als Anweisung, etwas zu tun).

Alle Zeichen in *str* müssen darstellbare Zeichen des US-ASCII-Zeichensatzes sein. *str* darf höchstens 255 Zeichen enthalten.

Ergebnis:

0 - String wurde gesendet
< 0 - Fehler (z.B. Programm nicht durch Frontend gestartet oder Frontend im “Textmodus”)

Hinweis:

Ein Aufruf, der einen Fehler zurückgibt, ist ansonsten ohne Folgen.

FIX kann nicht feststellen, ob das Frontend die Anweisung *str* unterstützt.

fx_fixwin_feature_level — Featurelevel von *FIX/Win* bestimmen

Definition:

int fx_fixwin_feature_level()

Beschreibung:

Im *FIX*-Programm kann der Featurelevel von *FIX/Win* mittels fx_fixwin_feature_level() ermittelt werden. Dieser Wert kann genutzt werden, um ein Programm an verschiedene *FIX/Win*-Versionen anzupassen.

Ergebnis:

Der Featurelevel von *FIX/Win* oder 0 im Fehlerfall.

fx_fwown_get_version_info — Informationen zur Benutzerbibliothek ermitteln

Definition:

char *fx_fwown_get_version_info(feature_level)
int *feature_level;

Beschreibung:

Die Funktion liefert einen Text und einen Featurelevel, der die Version der Benutzerbibliothek definiert. Es handelt sich dabei um die Werte, die von der selbst erstellten Funktion FwownGetVersionInfo() der Benutzerbibliothek geliefert werden. Diese Werte werden unmittelbar nach dem Verbindungsaufbau von *FIX/Win* an *FIX* übertragen.

Ergebnis:

Text, der die Version beschreibt.

fx_iconlist — Iconlist am Frontend einstellen

Definition:

int fx_iconlist(new_iclist_p, old_iclist_p)
unsigned int *new_iclist_p; unsigned int *old_iclist_p;

Beschreibung:

erklärt, falls *new_iclist_p* != (unsigned int *)0, die Iconlist mit ID **new_iclist_p* zur aktuellen Iconlist. Ist o*ld_iclist_p* != (unsigned int *)0, wird an dieser Adresse die ID der zuvor eingestellten Iconlist hinterlegt.

Ergebnis:

0 - Erfolg
< 0 - Fehler (z.B. Programm nicht durch Frontend gestartet)

Hinweis:

Ein Aufruf der Funktion zu einem ungeeigneten Zeitpunkt (Frontend im “Textmodus”) wird von *FIX* gepuffert und bei geeigneter Gelegenheit an das Frontend übermittelt.

Ein Aufruf, der einen Fehler zurückgibt, ist ansonsten ohne Folgen.

FIX kann nicht feststellen, ob das Frontend die Iconlist mit ID **new_iclist_p* unterstützt.

fx_transfer_to_frontend - Binäre Daten senden

Definition:

int fx_transfer_to_frontend(bytes, buf)
int bytes; char *buf;

Beschreibung:

Die Funktion sendet die binären Daten der Größe bytes im Speicherbereich *buf* an das Frontend. Dieses hat innerhalb der Funktion FwownProcessData() die Möglichkeit, die Daten zu bearbeiten und zurück zu senden. An die *FIX*-Anwendung wird beim Eintreffen der Daten das Event K_SB gesendet. Ein Zeiger auf den Speicherbereich kann durch Aufruf der Funktion getFrontendData() ermittelt werden. Die Daten sollten direkt verarbeitet werden oder in einen anwendungseigenen Puffer kopiert werden, da *FIX* den Speicherbereich beim Einlesen des nächsten Events wieder freigibt.

Ergebnis:

Als Rückgabewerte liefert die Funktion 0 oder -1, wenn kein Frontend die Anwendung bedient.

fxExecProgram — Kommando ausführen

Definition:

int fxExecProgram(cmdstring)
char *cmdstring;

Beschreibung:

ähnlich execute_cmd(*cmdstring*, 0), nur dass bei Bedienung mittels eines Frontends dieses zusätzlich angewiesen wird, vor der Ausführung des Kommandos seinen Zustand zu sichern und den Grundzustand anzunehmen. Nach der Ausführung wird es veranlasst, den gesicherten Zustand wieder zu restaurieren.

Ergebnis:

Exitcode des Kommandos
errno nach fehlgeschlagenem fork(), exec() oder wait(), wenn das Kommando nicht ausgeführt werden konnte oder abgebrochen wurde

Siehe auch:

execute_cmd()

Hinweis:

Bei Bedienung durch ein Frontend sind stdin, stdout und stderr während der Ausführung des Programms mit den Kommunikationskanälen zwischen der *FIX*-Anwendung und dem Frontend (Sockets) verbunden.

getFrontendData — Daten des Frontends lesen

Definition:

long getFrontendData(pBuf)
char **pBuf;

Beschreibung:

Die Funktion dient zur Ermittlung des Zeiger auf den Speicherbereich mit Daten, die von *FIX/Win* aufgrund eines fx_transfer_to_frontend() Aufrufs zurück gesendet wurden. Der Wert *pBuf* muss auf eine Variable vom Typ char * zeigen, in der der Zeiger auf die Daten abgelegt wird.

Ergebnis:

Als Rückgabewert liefert die Funktion die Größe des Datenbereichs.

Hinweis:

Die Daten sollten direkt verarbeitet werden oder in einen anwendungseigenen Puffer kopiert werden, da *FIX* den Speicherbereich beim Einlesen des nächsten Events wieder freigibt.

set_tooltip_allowed — Feldtooltips ein- bzw. ausschalten

Definition:

void set_tooltip_allowed(on)
BOOLEAN on;

Beschreibung:

Mit dieser Funktion kann die Ermittlung von Tooltips auf Feldern abgeschaltet werden. Ein Aufruf mit FALSE schaltet die Ermittlung so lange ab, bis ein Aufruf mit TRUE erfolgt. Ein zu diesem Zeitpunkt dargestellter Tooltip ist davon nicht betroffen. Tooltips auf Paintareas sind ebenfalls nicht betroffen. *FIX* nutzt diese Funktion intern selbst zum Abschalten der Feld-Tooltips während Meldungen auf Eingabe warten.

show_context_menu — Kontextmenü an *FIX/Win* senden

Definition:

BOOLEAN show_context_menu(ctx_men);
context_menu *ctx_men;

Beschreibung:

Durch den Aufruf der Funktion kann ein Kontextmenü angezeigt werden. Das Argument vom Typ context_menu * bestimmt den Aufbau des Kontextmenüs. Der Typ wird in der Datei `fix/fixwin.h` definiert, die über die Datei `fix/fix.h` eingebunden wird.

Ergebnis:

TRUE- Erfolg
FALSE- Fehler

Hinweis:

Vgl. “Nutzung von Kontextmenüs” auf Seite 239.

22 Funktionen zur formatierten Darstellung von long und double-Werten

fxrfmtdouble — double-Wert gemäß Format in String schreiben

Definition:

int fxrfmtdouble(dbl, format, outbuf)
double dbl; char *format; char *outbuf;

Beschreibung:

schreibt den Wert *dbl* entsprechend dem Format *format* an die Adresse *outbuf* und fügt ‘\0’ an. *outbuf* muss mindestens |*format*| +1 Bytes aufnehmen können.

Spezifiziert *format* zu wenige Vorkommastellen, wird *outbuf* mit ‘*’ gefüllt. Ist *dbl* NULL, werden entsprechend viele Leerzeichen geschrieben.

format darf maximal 80 Zeichen lang sein; ‘$’ , ‘*’ , ‘&’ , ‘#’ , ‘<’ , ‘,’ , ‘.’ , ‘-’ , ‘+’ , ‘(’ , ‘)’ werden als Formatierzeichen betrachtet.

Als Dezimalpunkt wird das in der Umgebungsvariable FXRADIXCHAR spezifizierte Zeichen (ersatzweise ‘.’) verwendet. Als Tausender-Trennzeichen wird ‘,’ bzw. ‘.’ verwendet je nach Wahl des Dezimalpunktes, als Währungszeichen stets ‘$’.

Ergebnis:

0 - Erfolg
-1211 - kein Speicher verfügbar (vgl. fxdecfcvt())
-1217 - Formatstring zu lang

Siehe auch:

fxrfmtdec(), fxrfmtlong()

Hinweis:

Ist FXRADIXCHAR beim Aufruf noch nicht ausgewertet, so geschieht dies jetzt.

Im Fehlerfall wird der Wert an der Adresse *outbuf* nicht verändert.

fxrfmtlong — long-Wert gemäß Format in String schreiben

Definition:

int fxrfmtlong(lg, format, outbuf)
double lg; char *format; char *outbuf;

Beschreibung:

schreibt den Wert *lg* entsprechend dem Format *format* an die Adresse *outbuf* und fügt ‘\0’ an. *outbuf* muss mindestens |*format*| +1 Bytes aufnehmen können.

Spezifiziert *format* zu wenige Vorkommastellen, wird *outbuf* mit ‘*’ gefüllt. Ist *lg* NULL, werden entsprechend viele Leerzeichen geschrieben.

format darf maximal 80 Zeichen lang sein; ‘$’ , ‘*’ , ‘&’ , ‘#’ , ‘<’ , ‘,’ , ‘.’ , ‘-’ , ‘+’ , ‘(’ , ‘)’ werden als Formatierzeichen betrachtet.

Als Dezimalpunkt wird das in der Umgebungsvariable FXRADIXCHAR spezifizierte Zeichen (ersatzweise '.') verwendet. Als Tausender-Trennzeichen wird ',' bzw. '.' verwendet je nach Wahl des Dezimalpunktes, als Währungszeichen stets '$'.

Ergebnis:

0 - Erfolg
-1211 - kein Speicher verfügbar (vgl. fxdecfcvt())
-1217 - Formatstring zu lang

Siehe auch:

fxrfmtdec(), fxrfmtdouble()

Hinweis:

Ist FXRADIXCHAR beim Aufruf noch nicht ausgewertet, so geschieht dies jetzt.

Im Fehlerfall wird der Wert an der Adresse *outbuf* nicht verändert.

23 dec_t-Funktionen

fxdecadd — dec_t-Werte addieren

Definition:

int fxdecadd(np1, np2, result)
dec_t *np1; dec_t *np2; dec_t *result;

Beschreibung:

addiert die Werte **np1* und **np2* und legt die Summe an der Adresse *result* ab. Ist einer der Summanden NULL, ist auch die Summe NULL.

Ergebnis:

0 bei Erfolg
-1200 - Überlauf (Betrag > 0.9…9e126)
-1201 - Summe nicht von 0 unterscheidbar (Betrag < 1e-129)

Siehe auch:

fxdecsub(), fxdecmul(), fxdecdiv()

Hinweis:

Die Adresse für das Ergebnis darf mit der eines Arguments übereinstimmen.

Im Fehlerfall ist der Wert an der Adresse *result* undefiniert.

fxdeccmp — dec_t-Werte vergleichen

Definition:

int fxdeccmp(np1, np2)
dec_t *np1; dec_t *np2;

Beschreibung:

vergleicht die Werte **np1* und **np2*.

Ergebnis:

-1 - ∗*np1* < ∗*np2*
0 - ∗*np1* == ∗*np2*
1 - ∗*np1* > ∗*np2*
-2 - ∗*np1* oder ∗*np2* ist NULL

fxdeccopy — dec_t-Wert kopieren

Definition:

void fxdeccopy(np, result)
dec_t ∗np; dec_t ∗result;

Beschreibung:

überträgt den Wert ∗*np* an die Adresse *result*.

fxdeccvasc — dec_t-Wert aus String lesen

Definition:

int fxdeccvasc(cp, len, np)
char ∗cp; int len; dec_t ∗np;

Beschreibung:

versucht, in den ersten *len* Zeichen beginnend an der Adresse *cp* eine Zahl zu erkennen. Bei Erfolg wird diese an die Adresse *np* geschrieben. *len* Leerzeichen ergeben den Wert NULL.

Als Dezimalpunkt wird das in der Umgebungsvariable FXRADIXCHAR spezifizierte Zeichen (ersatzweise '.') erwartet.

Ergebnis:

0 bei Erfolg
-1200 - Überlauf (Betrag > 0.9…9e126)
-1201 - Wert nicht von 0 unterscheidbar (Betrag < 1e-129)
-1213 - unzulässige Zeichen oder Exponent zu groß

Siehe auch:

fxdectoasc()

Hinweis:

Außer zum Test auf NULL werden führende und nachgestellte Leerzeichen ignoriert.

Zwischen (optionalem) Vorzeichen und der ersten Ziffer dürfen keine Leerzeichen stehen.

Nur die erste von '0' verschiedene und bis zu 31 darauf folgende Ziffern werden verarbeitet, alle weiteren ignoriert.

Ist FXRADIXCHAR beim Aufruf noch nicht ausgewertet, so geschieht dies jetzt.

Im Fehlerfall ist der Wert an der Adresse *np* undefiniert.

fxdeccvdbl — double-Wert in dec_t-Wert wandeln

Definition:

int fxdeccvdbl(dbl, np)
double dbl; dec_t ∗np;

Beschreibung:

wandelt den Wert *dbl* nach dec_t und schreibt das Ergebnis an die Adresse *np*. Ist *dbl* NULL, wird ∗*np* ebenfalls NULL zugewiesen.

Ergebnis:

0 bei Erfolg
-1200 - Überlauf (Betrag > 0.9…9e126)
-1201 - Wert nicht von 0 unterscheidbar (Betrag < 1e-129)

Siehe auch:

fxdectodbl(), fxdeccvlong(), fxdeccvshort()

Hinweis:

Im Fehlerfall ist der Wert an der Adresse *np* undefiniert.

fxdeccvlong — long-Wert in dec_t-Wert wandeln

Definition:

int fxdeccvlong(lng, np)
long lng; dec_t *np;

Beschreibung:

wandelt den Wert *lng* nach dec_t und schreibt das Ergebnis an die Adresse *np*. Ist *lng* NULL (kodiert als -2147483648), wird **np* ebenfalls NULL zugewiesen.

Ergebnis:

0

Siehe auch:

fxdectolong(), fxdeccvshort(), fxdeccvdbl()

Hinweis:

keine Wertebereichsprüfung.

fxdeccvshort — short-Wert in dec_t-Wert wandeln

Definition:

int fxdeccvshort(i, np)
int i; dec_t *np;

Beschreibung:

wandelt den Wert *i* nach dec_t und schreibt das Ergebnis an die Adresse *np*. Ist *i* NULL (kodiert als -32768), wird **np* ebenfalls NULL zugewiesen.

Ergebnis:

0

Siehe auch:

fxdectoshort(), fxdeccvlong(), fxdeccvdbl()

Hinweis:

keine Wertebereichsprüfung.

fxdecdiv — dec_t-Werte dividieren

Definition:

int fxdecdiv(np1, np2, result)
dec_t *np1; dec_t *np2; dec_t *result;

Beschreibung:

dividiert den Wert **np1* durch **np2* und legt das Ergebnis an der Adresse *result* ab. Ist der Dividend oder Divisor NULL, ist auch das Ergebnis NULL.

Ergebnis:

0 bei Erfolg
-1200 - Überlauf (Betrag > 0.9…9e126)
-1201 - Ergebnis nicht von 0 unterscheidbar (Betrag < 1e-129)
-1202 - Division durch 0

Siehe auch:

fxdecadd(), fxdecsub(), fxdecmul()

Hinweis:

Die Adresse für das Ergebnis darf mit der eines Arguments übereinstimmen.

Im Fehlerfall ist der Wert an der Adresse *result* undefiniert.

fxdececvt — Ziffernfolge eines dec_t-Wertes bestimmen

Definition:

char *fxdececvt(np, prec, decpt, sign)
dec_t *np; int prec; int *decpt; int *sign;

Beschreibung:

liefert einen Zeiger auf einen '\0'-terminierten Speicherbereich, der die signifikanten Ziffern des Wertes **np* mit *prec* Stellen Genauigkeit enthält, d.h. außer für den Wert 0 ist die erste Ziffer von '0' verschieden. An der Adresse *decpt* wird die Position des Dezimalpunktes relativ zum Anfang des Ergebnisstring abgelegt (kleiner 0 bedeutet "links davon"), an der Adresse *sign* 0 bei einem Wert größer-gleich 0, sonst 1.

Ist **np* NULL, stehen an der zurückgegebenen Adresse *prec* Leerzeichen und die Werte an den Adressen *decpt* und *sign* sind undefiniert.

Ergebnis:

Zeiger auf einen Speicherbereich, der gelegentlich wieder überschrieben wird
(char *)0 bei Fehler (kein Speicher allokierbar oder *prec* < 0)

Siehe auch:

fxdecfcvt(), C-Funktion ecvt()

Hinweis:

Der Ergebnisstring enthält stets *prec* Zeichen.

Im Fehlerfall werden die Werte an den Adressen *decpt* und *sign* nicht verändert.

fxdecfcvt — Ziffernfolge eines dec_t-Wertes bestimmen

Definition:

char *fxdecfcvt(np, scale, decpt, sign)
dec_t *np; int scale; int *decpt; int *sign;

Beschreibung:

liefert einen Zeiger auf einen '\0'-terminierten Speicherbereich, der die Ziffern des auf *scale* Nachkommastellen gerundeten Wertes ∗*np* enthält. Dabei sind führende Nullen weggelassen, d.h. die erste Ziffer ist stets von '0' verschieden. An der Adresse *decpt* wird die Position des Dezimalpunktes relativ zum Anfang des Ergebnisstring abgelegt (kleiner 0 bedeutet "links davon"), an der Adresse *sign* 0 bei einem Wert größer-gleich 0, sonst 1.

Ist ∗*np* NULL, stehen an der zurückgegebenen Adresse *scale* Leerzeichen und die Werte an den Adressen *decpt* und *sign* sind undefiniert.

Ergebnis:

Zeiger auf einen Speicherbereich, der gelegentlich wieder überschrieben wird
(char ∗)0 bei Fehler (kein Speicher allokierbar oder *scale* < 0)

Siehe auch:

fxdecfcvt(), C-Funktion fcvt()

Hinweis:

Für einen Wert mit Betrag kleiner 0.5e-*scale* wird ein Leerstring zurückgegeben.

Im Fehlerfall werden die Werte an den Adressen *decpt* und *sign* nicht verändert.

fxdecmul — dec_t-Werte multiplizieren

Definition:

int fxdecmul(np1, np2, result)
dec_t ∗np1; dec_t ∗np2; dec_t ∗result;

Beschreibung:

multipliziert den Wert ∗*np1* mit ∗*np2* und legt das Produkt an der Adresse *result* ab. Ist einer der Faktoren NULL, ist auch das Ergebnis NULL.

Ergebnis:

0 bei Erfolg
-1200 - Überlauf (Betrag > 0.9…9e126)
-1201 - Produkt nicht von 0 unterscheidbar (Betrag < 1e-129)

Siehe auch:

fxdecadd(), fxdecsub(), fxdecdiv()

Hinweis:

Die Adresse für das Ergebnis darf mit der eines Arguments übereinstimmen.

Im Fehlerfall ist der Wert an der Adresse *result* undefiniert.

fxdecround — dec_t-Wert runden

Definition:

void fxdecround(np, scale)
dec_t ∗np; int scale;

Beschreibung:

rundet den Wert an der Adresse ∗*np* auf *scale* Nachkommastellen. NULL bleibt bewahrt.

scale muss größer-gleich 0 sein.

Siehe auch:

fxdectrunc()

Hinweis:

Abgerundet wird stets gegen 0 hin.

fxdecsub — dec_t-Werte subtrahieren

Definition:

int fxdecsub(np1, np2, result)
dec_t *np1; dec_t *np2; dec_t *result;

Beschreibung:

subtrahiert vom Wert **np1* den Wert **np2* und legt die Differenz an der Adresse *result* ab. Ist der Minuend oder Subtrahend NULL, ist auch das Ergebnis NULL.

Ergebnis:

0 bei Erfolg
-1200 - Überlauf (Betrag > 0.9…9e126)
-1201 - Differenz nicht von 0 unterscheidbar (Betrag < 1e-129)

Siehe auch:

fxdecadd(), fxdecmul(), fxdecdiv()

Hinweis:

Die Adresse für das Ergebnis darf mit der eines Arguments übereinstimmen.

Im Fehlerfall ist der Wert an der Adresse *result* undefiniert.

fxdectoasc — dec_t-Wert in String schreiben

Definition:

int fxdectoasc(np, cp, len, right)
dec_t *np; char *cp; int len; int right;

Beschreibung:

versucht, den Wert von **np* als Literal an die Adresse *cp* zu schreiben, wo Raum für *len* Zeichen erwartet wird. Ist *right* größer-gleich 0, werden möglichst *right* Nachkommastellen verwendet, sonst so viele wie notwendig.

Reichen *len* Zeichen nicht aus, wird gerundet oder ggf. die wissenschaftliche Notation verwendet, wenn dadurch mehr signifikante Ziffern bewahrt werden. Ist die Zahl auch so nicht darstellbar, werden *len* '*' hinterlegt.

Werden zur Darstellung weniger als *len* Zeichen benötigt, wird nach rechts mit Leerzeichen aufgefüllt.

Als Dezimalpunkt wird das in der Umgebungsvariable FXRADIXCHAR spezifizierte Zeichen (ersatzweise '.') verwendet.

Ergebnis:

0 bei Erfolg
-1211 - kein Speicher verfügbar
< 0 bei sonstigem Fehler

Siehe auch:

fxdeccvasc()

Hinweis:

Ist FXRADIXCHAR beim Aufruf noch nicht ausgewertet, so geschieht dies jetzt.

Bei der wissenschaftlichen Darstellung umfasst der Exponent mindestens zwei Ziffern.

Das Ergebnis ist nicht mit ‘\0’ abgeschlossen.

Im Fehlerfall ist der Wert an der Adresse *cp* undefiniert.

fxdectodbl — dec_t-Wert in double-Wert wandeln

Definition:

int fxdectodbl(np, dblp)
dec_t *np; double *dblp;

Beschreibung:

wandelt den Wert **np* nach double und schreibt das Ergebnis an die Adresse *dblp*. Ist **np* NULL, wird **dblp* ebenfalls NULL zugewiesen.

Ergebnis:

0 bei Erfolg
-1200 - Betrag von **np* zu groß für double

Siehe auch:

fxdeccvdbl(), fxdectoshort(), fxdectolong()

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *dblp* nicht verändert.

fxdectolong — dec_t-Wert in long-Wert wandeln

Definition:

int fxdectolong(np, lngp)
dec_t *np; long *lngp;

Beschreibung:

wandelt den Wert **np* nach long und schreibt das Ergebnis an die Adresse *lngp*. Ist **np* NULL, wird **lngp* ebenfalls NULL (kodiert als -2147483648) zugewiesen.

Ergebnis:

0 - Erfolg
-1200 - Betrag des ganzzahligen Anteils von **np* größer als 2147483647

Siehe auch:

fxdeccvlong(), fxdectoshort(), fxdectodbl()

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *lngp* nicht verändert.

fxdectoshort — dec_t-Wert in short-Wert wandeln

Definition:

int fxdectoshort(np, ip)
dec_t *np; int *ip;

Beschreibung:

wandelt den Wert ∗*np* nach int und schreibt das Ergebnis an die Adresse *ip*. Ist ∗*np* NULL, wird ∗*ip* ebenfalls NULL (kodiert als -32768) zugewiesen.

Ergebnis:

0 bei Erfolg
-1200 - Betrag des ganzzahligen Anteils der Zahl größer als 32767

Siehe auch:

fxdeccvshort(), fxdectolong(), fxdectodbl()

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *ip* nicht verändert.

fxdectrunc — dec_t-Wert abschneiden

Definition:

void fxdectrunc(np, scale)
dec_t ∗np; int scale;

Beschreibung:

schneidet den Wert an der Adresse *np* nach der *scale*-ten Nachkommastellen ab. NULL bleibt bewahrt.

scale muss größer-gleich 0 sein.

Siehe auch:

fxdecround()

fxrfmtdec — dec_t-Wert gemäß Format in String schreiben

Definition:

int fxrfmtdec(np, format, outbuf)
dec_t ∗np; char ∗format; char ∗outbuf;

Beschreibung:

schreibt den Wert ∗*np* entsprechend dem Format *format* an die Adresse *outbuf* und fügt '\0' an. *outbuf* muss mindestens |*format*| +1 Bytes aufnehmen können.

Spezifiziert *format* zu wenige Vorkommastellen, wird *outbuf* mit '∗' gefüllt. Ist ∗*np* NULL, werden entsprechend viele Leerzeichen geschrieben.

format darf maximal 80 Zeichen lang sein; '$' , '∗' , '&' , '#' , '<' , ',' , '.' , '-' , '+' , '(' , ')' werden als Formatierzeichen betrachtet.

Als Dezimalpunkt wird das in der Umgebungsvariable FXRADIXCHAR spezifizierte Zeichen (ersatzweise '.') verwendet. Als Tausender-Trennzeichen wird ',' bzw. '.' verwendet je nach Wahl des Dezimalpunktes, als Währungszeichen stets '$'.

Ergebnis:

0 - Erfolg
-1211 - kein Speicher verfügbar (vgl. fxdecfcvt())
-1217 - Formatstring zu lang

Siehe auch:

fxdectoasc(), fxrfmtdec(), fxrfmtlong()

Hinweis:

Ist FXRADIXCHAR beim Aufruf noch nicht ausgewertet, so geschieht dies jetzt.

Im Fehlerfall wird der Wert an der Adresse *outbuf* nicht verändert.

fxfracfmtdec — dec_t-Wert gemäß Bruchteil-Format in String schreiben

Definition:

int fxfracfmtdec(np, format, outbuf)
dec_t *np; char *format; char *outbuf;

Beschreibung:

schreibt den Wert **np* entsprechend dem Bruchteil-Format *format* an die Adresse *outbuf* und fügt '\0' an. *outbuf* muss mindestens |*format*| +1 Bytes aufnehmen können.

Spezifiziert *format* zu wenige Vorkommastellen, wird *outbuf* mit '*' gefüllt. Ist **np* NULL, werden entsprechend viele Leerzeichen geschrieben.

Als Tausender-Trennzeichen wird ',' bzw. '.' verwendet je nach Wahl des Dezimalpunktes.

Ergebnis:

0 - Erfolg
-3 - *format* kein Bruchteil-Format
< 0 - sonstiger Fehler

Siehe auch:

fxrfmtdec()

Hinweis:

Ist FXRADIXCHAR beim Aufruf noch nicht ausgewertet, so geschieht dies jetzt.

Im Fehlerfall ist der Wert an der Adresse *outbuf* undefiniert.

24 date_t-Funktionen

atodate — Datum-Literal in Datum konvertieren

Definition:

date_t atodate(dt_str, leng)
char *dt_str; int leng;

Beschreibung:

wandelt das Datum-Literal *dt_str* in ein Datum um; *leng* steuert (zusammen mit dem Wert der Umgebungsvariable FX-DATE) das erwartete Format.

leng	erwartetes Format:	FXDATE="DMY4./"
10	Tag, Monat, vierstelliges Jahr mit Standardtrennzeichen	dd.mm.yyyy
8	Tag, Monat, zweistelliges Jahr mit Standardtrennzeichen	dd.mm.yy
6	Tag, Monat, zweistelliges Jahr ohne Trennzeichen	ddmmyy
7,-7	Monat, vierstelliges Jahr mit alternativem Trennzeichen	mm/yyyy
5,-5	Monat, zweistelliges Jahr mit alternativem Trennzeichen	mm/yy

Besonderheiten:

Bei Formaten mit vierstelligem Jahr wird eine Jahresangabe, die aus exakt zwei Ziffern besteht, auf den aktuellen 100-Jahre-Zeitraum bezogen.

Bei Formaten ohne Tagkomponente wird, ist *leng* negativ, statt des ersten der letzte Tag des Monats zurückgegeben (“06/89” --> 30.06.1989).

Ergebnis:

Datum-Wert
NULL, wenn *dt_str* Leerstring oder fehlerhaft

Siehe auch:

datetoa()

datetoa — Datum in Datum-Literal konvertieren

Definition:

int datetoa(dt_str, jdate, leng)
char *dt_str; date_t jdate; int leng;

Beschreibung:

legt in *dt_str* das Literal des Datums *jdate* ab im Format

leng	erzeugtes Format	FXDATE=“DMY4./”
10	Tag, Monat, vierstelliges Jahr mit Standardtrennzeichen	dd.mm.yyyy
8	Tag, Monat, zweistelliges Jahr mit Standardtrennzeichen	dd.mm.yy
6	Tag, Monat, zweistelliges Jahr ohne Trennzeichen	ddmmyy
7	Monat, vierstelliges Jahr mit alternativem Trennzeichen	mm/yyyy
5	Monat, zweistelliges Jahr mit alternativem Trennzeichen	mm/yy

jeweils abgeschlossen durch ‘\0’. An der Adresse *dt_str* müssen mindestens *leng* + 1 Bytes zur Verfügung stehen.

Ergebnis:

0 - Erfolg
!= 0 - Fehler

Siehe auch:

atodate(), fx_today()

Hinweis:

Für NULL werden *leng* Leerzeichen hinterlegt.

Bei Formaten mit vierstelligem Jahr muss *jdate* zwischen dem 01.01.0001 und dem 31.12.9999, bei solchen mit zweistelligem Jahr im aktuellen 100-Jahre-Zeitraum liegen.

Im Fehlerfall werden *leng* ‘*’ nach *dt_str* geschrieben.

day_of_year — aus Datum Tag im Jahr bestimmen

Definition:

int day_of_year(yyyy, mm, dd)
int yyyy; int mm; int dd;

Beschreibung:

bestimmt die laufende Nummer des Tages im Jahr.

Ergebnis:

laufende Nummer des *dd*-ten Tages des Monats *mm* des Jahres *yyyy* bei Erfolg
-1 bei Fehler

Hinweis:

Die Jahresangabe wird aufgrund von Schaltjahren benötigt.

first_of_month — Monatsanfang ermitteln

Definition:

date_t first_of_month(jdate)
date_t jdate;

Beschreibung:

berechnet das Datum des ersten Tages des Monats, in dem *jdate* liegt.

Ergebnis:

Datum-Wert
NULL, falls *jdate* NULL-Wert oder bei Fehler

Siehe auch:

last_of_month()

fx_today — relative Datumsangabe in Datum-Literal konvertieren

Definition:

BOOLEAN fx_today(dt_str, leng, offset)
char *dt_str; int leng; int offset;

Beschreibung:

legt in *dt_str* das Literal des um *offset* Tage vermehrten aktuellen Datums ab in demselben Format wie datetoa(). An der Adresse *dt_str* müssen mindestens *leng* + 1 Bytes zur Verfügung stehen.

Ergebnis:

TRUE, wenn seit dem letzten Aufruf das aktuelle Datum gewechselt hat
FALSE sonst

Siehe auch:

datetoa()

Hinweis:

benutzt datetoa()

fxrdatestr — date_t-Wert in String schreiben

Definition:

int fxrdatestr(jdate, str)
date_t jdate; char *str;

Beschreibung:

schreibt den date_t-Wert *jdate* im Standard-Datumsformat an die Adresse *str* und fügt '\0' an. Für NULL werden entsprechend viele Leerzeichen geschrieben.

Das Standard-Datumsformat ist “dd.mm.yyyy”, sofern die Umgebungsvariable FXDATE nichts anderes spezifiziert.

Ergebnis:

0 bei Erfolg
-1212 - fehlerhaftes Datumsformat
-1210 - *jdate* unzulässig

Siehe auch:

fxrfmtdate()

Hinweis:

Ist FXDATE beim Aufruf noch nicht ausgewertet, geschieht dies jetzt.

Im Fehlerfall bleibt der Wert an der Adresse *str* unverändert.

fxrdayofweek — Wochentag zu einem date_t-Wert bestimmen

Definition:

int fxrdayofweek(jdate)
date_t jdate;

Beschreibung:

bestimmt den Wochentag des date_t-Wertes *jdate*.

Ergebnis:

>= 0 bei Erfolg: 0 steht für Sonntag, 1 für Montag u.s.w.
-1 - *jdate* NULL
-1210 - *jdate* unzulässig

fxrdefmtdate — date_t-Wert in String erkennen

Definition:

int fxrdefmtdate(jdatep, fmtstring, inbuf)
date_t *jdatep; char *fmtstring; char *inbuf;

Beschreibung:

versucht, im String *inbuf* ein Datum zu erkennen. Voraussetzung ist, dass die Datumsbestandteile darin in der gleichen Reihenfolge auftreten wie die Platzhalter für Tag (dd), Monat (mm, mmm) und Jahr (yy, yyyy) in der Formatangabe *fmtstring* oder dass *inbuf* ausschließlich Leerzeichen enthält. Bei Erfolg wird das Datum an der Adresse *jdatep* abgelegt.

Ergebnis:

0 bei Erfolg
-1212 - fehlerhaftes Datumsformat
-1218 - überzählige Ziffern in *inbuf*
-1204 - Jahresangabe unzulässig
-1205 - Monatsangabe unzulässig
-1206 - Tagesangabe unzulässig

Siehe auch:

fxrstrdate()

Hinweis:

Die Abkürzungen der Monatsnamen entstammen der *FIX*-Meldungsdatei.

Bei einer ein- oder zweiziffrigen Jahresangabe wird - unabhängig vom Platzhalter - das 20. Jahrhundert angenommen; diese Annahme kann durch die Umgebungsvariable FX_RDTCV_YEAR_THRESHOLD übersteuert werden.

Im Fehlerfall bleibt der Wert an der Adresse *jdatep* unverändert.

fxrfmtdate — date_t-Wert gemäß Format in String schreiben

Definition:

int fxrfmtdate(jdate, fmtstring, outbuf)
date_t jdate; char *fmtstring; char *outbuf;

Beschreibung:

schreibt den date_t-Wert *jdate* gemäß der Formatangabe *fmtstring* an die Adresse *outbuf* und fügt '\0' an. *outbuf* muss mindestens |*fmtstring*| + 1 Zeichen aufnehmen können. Die Platzhalter für Wochentag (ddd), Tag (dd), Monat (mm, mmm) und Jahr (yy, yyyy) werden ersetzt, alle sonstigen Zeichen der Formatangabe übernommen. Ist jdate NULL, werden |*fmtstring*| Leerzeichen geschrieben.

Ergebnis:

0 bei Erfolg
-1212 - *fmtstring* unzulässig
-1210 - *jdate* unzulässig

Siehe auch:

fxrdatestr()

Hinweis:

Die Abkürzungen der Monats- und Wochentagsnamen entstammen der *FIX*-Meldungsdatei.

Im Fehlerfall bleibt der Wert an der Adresse *outbuf* unverändert.

fxrjulmdy — Komponenten aus date_t-Wert extrahieren

Definition:

int fxrjulmdy(jdate, mdy)
date_t jdate; short mdy[3];

Beschreibung:

schreibt Monat, Tag und Jahr des date_t-Wertes *jdate* nach *mdy*[0], *mdy*[1] und *mdy*[2]. Ist *jdate* NULL, wird den Elementen von *mdy* 0 zugewiesen.

Ergebnis:

0 bei Erfolg
-1210 - *jdate* unzulässig

Siehe auch:

fxrmdyjul()

Hinweis:

Im Fehlerfall bleibt der Wert der Elemente von *mdy* unverändert.

fxrleapyear — Jahreszahl auf Schaltjahr testen

Definition:

int fxrleapyear(year)
int year;

Beschreibung:

testet, ob es sich bei dem Jahr *year* um ein Schaltjahr handelt.

Ergebnis:

0 - kein Schaltjahr
1 - Schaltjahr
-1204 - *year* unzulässig

fxrmdyjul — date_t-Wert aus Komponenten bilden

Definition:

int fxrmdyjul(mdy, jdatep)
short mdy[3]; date_t *jdatep;

Beschreibung:

bildet aus den Angaben von Monat, Tag und Jahr in *mdy*[0], *mdy*[1] und *mdy*[2] einen date_t-Wert und legt ihn an der Adresse *jdatep* ab. Enthalten alle Elemente von *mdy* 0, wird NULL zugewiesen.

Ergebnis:

0 bei Erfolg
-1204 - Jahresangabe unzulässig
-1205 - Monatsangabe unzulässig
-1206 - Tagesangabe unzulässig

Siehe auch:

fxrjulmdy()

Hinweis:

Im Fehlerfall bleibt der Wert an der Adresse *jdatep* unverändert.

fxrstrdate — date_t-Wert in String erkennen

Definition:

int fxrstrdate(str, jdatep)
char *str; date_t *jdatep;

Beschreibung:

versucht, im String *str* ein Datum zu erkennen. Voraussetzung ist, dass die Datumsbestandteile darin in der gleichen Reihenfolge auftreten wie die Platzhalter für Tag (dd), Monat (mm, mmm) und Jahr (yy, yyyy) im Standard-Datumsformat oder dass *str* ausschließlich Leerzeichen enthält. Bei Erfolg wird das Datum an der Adresse *jdatep* abgelegt.

Das Standard-Datumsformat ist “dd.mm.yyyy”, sofern die Umgebungsvariable FXDATE nichts anderes spezifiziert.

Ergebnis:

0 bei Erfolg
-1212 - fehlerhaftes Datumsformat
-1218 - überzählige Ziffern in *str*
-1204 - Jahresangabe unzulässig
-1205 - Monatsangabe unzulässig
-1206 - Tagesangabe unzulässig

Siehe auch:

fxrdefmtdate()

Hinweis:

Ist FXDATE beim Aufruf noch nicht ausgewertet, geschieht dies jetzt.

Die Abkürzungen der Monatsnamen entstammen der *FIX*-Meldungsdatei.

Bei einer ein- oder zweiziffrigen Jahresangabe wird das 20. Jahrhundert angenommen; diese Annahme kann durch die Umgebungsvariable FX_RDTCV_YEAR_THRESHOLD übersteuert werden.

Im Fehlerfall bleibt der Wert an der Adresse *jdatep* unverändert.

fxrtoday — aktuelles Datum ermitteln

Definition:

void fxrtoday(jdatep)
date_t *jdatep;

Beschreibung:

legt das aktuelle Datum an der Adresse *jdatep* ab.

last_of_month — Monatsende ermitteln

Definition:

date_t last_of_month(jdate)
date_t jdate;

Beschreibung:

berechnet das Datum des letzten Tages des Monats, in dem *jdate* liegt.

Ergebnis:

Datum-Wert
NULL, falls *jdate* NULL-Wert oder bei Fehler

Siehe auch:

first_of_month()

25 dtime_t- und intrvl_t-Funktionen

fxdtaddinv — zu dtime_t-Wert intrvl_t-Wert addieren

Definition:

int fxdtaddinv(dt, inv, res)
dtime_t *dt; intrvl_t *inv; dtime_t *res;

Beschreibung:

addiert zu dem Zeitpunkt **dt* die Zeitspanne **inv* und legt das Ergebnis, das die Genauigkeit von **dt* erbt, an der Adresse *res* ab. **dt* muss zumindest all jene Komponenten enthalten, die in **inv* vorkommen.

Bei MONTH TO …-Zeitpunkten muss das Ergebnis im gleichen Jahr, bei DAY TO …-Zeitpunkten im gleichen Monat und Jahr wie *dt* liegen. Der Aufruf schlägt fehl, wenn bei der Addition eines YEAR(…) TO MONTH- oder MONTH(…) TO MONTH-Intervalls ein im Kalender nicht existierender Tag entstehen würde.

Ergebnis:

0 bei Erfolg
-1266 - Qualifier ungeeignet
-1204, -1267 - Ergebnis außerhalb des Wertebereichs
-1268 - ungültiger Qualifier
< 0 -sonstiger Fehler

Siehe auch:

fxdtsubinv()

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *res* nicht verändert.

fxdtcurrent — augenblicklichen Zeitpunkt bestimmen

Definition:

void fxdtcurrent(dt)
dtime_t *dt;

Beschreibung:

hinterlegt den aktuellen Zeitpunkt an der Adresse *dt*. Enthält *dt*->dt_qual beim Aufruf einen zulässigen Qualifier, erbt das Ergebnis diese Genauigkeit, sonst besitzt es die Genauigkeit YEAR TO FRACTION(3).

fxdtcvasc — dtime_t-Wert in String erkennen

Definition:

int fxdtcvasc(str, dt)
char *str; dtime_t *dt;

Beschreibung:

versucht, im String *str* einen dtime_t-Wert zu erkennen, der bei Erfolg an der Adresse *dt* abgelegt wird. *str* muss genau die der Genauigkeit *dt*->dt_qual entsprechenden Komponenten in Standarddarstellung enthalten.

Ergebnis:

0 bei Erfolg
-1260 - Fehler bei Typkonversion
-1261 - erste Komponente enthält zu viele Ziffern
-1262 - Komponente enthält nicht-numerisches Zeichen
-1263 - Komponente außerhalb des Wertebereichs
-1264 - überzählige Zeichen am Stringende
-1265 - Überlauf bei DATETIME/INTERVAL-Operation
-1266 - Qualifier ungeeignet
-1267 - Ergebnis außerhalb des Wertebereichs
-1268 - ungültiger Qualifier

Siehe auch:

fxdttoasc()

Hinweis:

Führende oder nachgestellte Leerzeichen sind zulässig.

Bei einer ein- oder zweiziffrigen Jahresangabe wird das 20. Jahrhundert angenommen; diese Annahme kann durch die Umgebungsvariable FX_RDTCV_YEAR_THRESHOLD übersteuert werden.

fxdtextend — Genauigkeit eines dtime_t-Wertes wandeln

Definition:

int fxdtextend(dt, res)
dtime_t *dt; dtime_t *res;

Beschreibung:

versucht, den dtime-Wert **dt* mit der Genauigkeit *res*->dt_qual an der Adresse *res* abzulegen. Komponenten im Ergebnis, die in **dt* nicht vorkommen, werden, sofern sie höherwertiger sind, entsprechend dem aktuellen Zeitpunkt, ansonsten mit dem kleinstmöglichen Wert gefüllt.

Ergebnis:

0 bei Erfolg
-1268 - ungültiger Qualifier

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *res* nicht verändert.

fxdtsubdt — zwei dtime_t-Werte voneinander subtrahieren

Definition:

int fxdtsubdt(dt1, dt2, res)
dtime_t *dt1; dtime_t *dt2; intrvl_t *res;

Beschreibung:

subtrahiert von dem Zeitpunkt **dt1* den Zeitpunkt **dt2* und legt das Ergebnis an der Adresse *res* ab. Vor der Subtraktion wird der Subtrahend auf die Genauigkeit des Minuenden erweitert. Die Differenz wird auf die Genauigkeit *res*->in_qual erweitert.

Enthält das Ergebnis Tage, Stunden, Minuten, Sekunden oder Sekundenbruchteile, darf *res*->in_qual nicht INTERVAL YEAR(…) TO MONTH oder INTERVAL MONTH(…) TO MONTH sein.

Ergebnis:

0 bei Erfolg
-1266 - Qualifier ungeeignet
-1268 - ungültiger Qualifier
< 0 - sonstiger Fehler

Siehe auch:

fxdtsubinv()

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *res* nicht verändert.

fxdtsubinv — von dtime_t-Wert intrvl_t-Wert subtrahieren

Definition:

int fxdtsubinv(dt, inv, res)
dtime_t *dt; intrvl_t *inv; dtime_t *res;

Beschreibung:

subtrahiert von dem Zeitpunkt **dt* die Zeitspanne **inv* und legt das Ergebnis, das die Genauigkeit von **dt* erbt, an der Adresse *res* ab. **dt* muss zumindest all jene Komponenten enthalten, die in **inv* vorkommen.

Bei MONTH TO …-Zeitpunkten muss das Ergebnis im gleichen Jahr, bei DAY TO …-Zeitpunkten im gleichen Monat und Jahr wie *dt* liegen. Der Aufruf schlägt fehl, wenn bei der Addition eines YEAR(…) TO MONTH- oder MONTH(…) TO MONTH-Intervalls ein im Kalender nicht existierender Tag entstehen würde.

Ergebnis:

0 bei Erfolg
-1204, -1267 - Ergebnis außerhalb des Wertebereichs
-1268 - ungültiger Qualifier
< 0 - sonstiger Fehler

Siehe auch:

fxdtaddinv()

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *res* nicht verändert.

fxdttoasc — dtime_t-Wert in Standardform in String schreiben

Definition:

int fxdttoasc(dt, str)
dtime_t *dt; char *str;

Beschreibung:

schreibt den dtime_t-Wert **dt* im Standard-DATETIME-Format an die Adresse *str* und fügt '\0' an. An der Adresse *str* muss hinreichend Platz zur Verfügung stehen.

Ergebnis:

0 bei Erfolg
< 0 bei Fehler

Siehe auch:

fxdtcvasc()

Hinweis:

Die längstmögliche Ausgabe entsteht für DATETIME YEAR TO FRACTION(5) und umfasst (einschl. '\0') 26 Zeichen ("yyyy-mm-dd hh:mm:ss,fffff").

fxinvcvasc — intrvl_t-Wert in String erkennen

Definition:

int fxinvcvasc(str, inv)
char *str; intrvl_t *inv;

Beschreibung:

versucht, im String *str* einen intrvl_t-Wert zu erkennen, der bei Erfolg an der Adresse *dt* abgelegt wird. *str* muss genau die der Genauigkeit *inv*->in_qual entsprechenden Komponenten in Standarddarstellung enthalten.

Ergebnis:

0 bei Erfolg
-1260 - Fehler bei Typkonversion
-1261 - erste Komponente enthält zu viele Ziffern
-1262 - Komponente enthält nicht-numerisches Zeichen
-1263 - Komponente außerhalb des Wertebereichs
-1264 - überzählige Zeichen am Stringende
-1265 - Überlauf bei DATETIME/INTERVAL-Operation

-1266 - Qualifier ungeeignet
-1267 - Ergebnis außerhalb des Wertebereichs
-1268 - ungültiger Qualifier

Siehe auch:

fxinvtoasc()

Hinweis:

Führende oder nachgestellte Leerzeichen sind zulässig.

fxinvdivdbl — intrvl_t-Wert durch Zahl dividieren

Definition:

int fxinvdivdbl(inv, dbl, res)
intrvl_t *inv; double dbl; intrvl_t *res;

Beschreibung:

dividiert die Zeitspanne **inv* durch *dbl* und legt das Ergebnis an der Adresse *res* ab. Vor der Division wird der Dividend auf die Genauigkeit *res*->in_qual erweitert. *inv*->in_qual und *res*->in_qual müssen also kompatibel sein.

Ergebnis:

0 bei Erfolg
-1200 - Überlauf
-1201 - Produkt nicht von 0 unterscheidbar
-1202 - Division durch 0
-1265 - Überlauf bei DATETIME/INTERVAL-Operation
-1266 - Qualifier ungeeignet
-1268 - ungültiger Qualifier
< 0 - sonstiger Fehler

Siehe auch:

fxinvmuldbl()

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *res* nicht verändert.

fxinvdivinv — intrvl_t-Wert durch intrvl_t-Wert dividieren

Definition:

int fxinvdivinv(i1, i2, res)
intrvl_t *i1; intrvl_t *i2; double *res;

Beschreibung:

dividiert die Zeitspanne **i1* durch die Zeitspanne **i2* und legt das Ergebnis an der Adresse *res* ab. Vor der Division werden Dividend und Divisor auf die gleiche Genauigkeit erweitert. *i1*->in_qual und *i2*->in_qual müssen also kompatibel sein.

Ergebnis:

0 bei Erfolg
-1200 - Überlauf
-1201 - Quotient nicht von 0 unterscheidbar
-1266 - Qualifier ungeeignet
-1268 - ungültiger Qualifier
< 0 - sonstiger Fehler

Siehe auch:

fxinvdivdbl()

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *res* nicht verändert.

fxinvextend — Genauigkeit eines intrvl_t-Wertes wandeln

Definition:

int fxinvextend(inv, res)
intrvl_t *inv; intrvl_t *res;

Beschreibung:

versucht, den intrvl_t-Wert **inv* mit der Genauigkeit *res*->in_qual an der Adresse *res* abzulegen. *inv*->in_qual und *res*->in_qual müssen also kompatibel sein. Komponenten des Ergebnisses, die in **inv* nicht vorkommen, werden, sofern sie höherwertiger sind, mit gültigen Werten, anderenfalls mit 0 gefüllt.

Ergebnis:

0 bei Erfolg
-1265 - Überlauf bei DATETIME/INTERVAL-Operation
-1266 - Qualifier ungeeignet
-1268 - ungültiger Qualifier

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *res* nicht verändert.

fxinvmuldbl — intrvl_t-Wert mit Zahl multiplizieren

Definition:

int fxinvmuldbl(inv, dbl, res)
intrvl_t *inv; double dbl; intrvl_t *res;

Beschreibung:

multipliziert die Zeitspanne **inv* mit *dbl* und legt das Ergebnis an der Adresse *res* ab. Vor der Multiplikation wird der Multiplikand auf die Genauigkeit *res*->in_qual erweitert. *inv*->in_qual und *res*->in_qual müssen also kompatibel sein.

Ergebnis:

0 bei Erfolg
-1200 - Überlauf
-1201 - Produkt nicht von 0 unterscheidbar
-1265 - Überlauf bei DATETIME/INTERVAL-Operation
-1266 - Qualifier ungeeignet
-1268 - ungültiger Qualifier
< 0 - sonstiger Fehler

Siehe auch:

fxinvdivdbl()

Hinweis:

Im Fehlerfall wird der Wert an der Adresse *res* nicht verändert.

fxinvtoasc — intrvl_t-Wert in Standardform in String schreiben

Definition:

int fxinvtoasc(inv, str)
intrvl_t *inv; char *str;

Beschreibung:

schreibt den intrvl_t-Wert **inv* im Standard-INTERVAL-Format an die Adresse *str* und fügt '\0' an. An der Adresse *str* muss ausreichend Platz bereitstehen.

Ergebnis:

0 bei Erfolg
< 0 bei Fehler

Siehe auch:

fxinvcvasc()

Hinweis:

Die längstmögliche Ausgabe entsteht für INTERVAL DAY(9) TO FRACTION(5) und umfasst (einschl. Vorzeichen und '\0') 26 Zeichen ("-ddddddddd hh:mm:ss,fffff"). Bei einem Wert größer-gleich 0 beginnt das Ergebnis mit einem Leerzeichen.

26 Wahrheitswert-Funktionen

atotruth — Wahrheitswert in String erkennen

Definition:

int atotruth(str)
char *str;

Beschreibung:

wandelt das Wahrheitswert-Literal *str* in einen Wahrheitswert um.

Ergebnis:

S_no, wenn Darstellung von Nein erkannt
S_yes, wenn Darstellung von Ja erkannt
NULL, wenn *str* Leerstring oder fehlerhaft

Siehe auch:

truthtoa()

truthtoa — Wahrheitswert in String schreiben

Definition:

int truthtoa(str, val)
char *str; char *val;

Beschreibung:

legt in *str* das Literal des Wahrheitswertes **val* ab, abgeschlossen durch '\0'.

**val* *str*

'\0' / ' '	“ ”
S_no	“n” (bzw. das in FXTRUTH hinterlegte Zeichen)
S_yes	“j” (bzw. das in FXTRUTH hinterlegte Zeichen)
sonst	“∗”

An der Adresse *str* müssen mindestens 2 Bytes zur Verfügung stehen.

Ergebnis:

0 - Erfolg
!= 0 - Fehler

Siehe auch:

atotruth()

27 Konversion

fxisnull — auf NULL-Wert testen

Definition:

```
int fxisnull(type, adr)
int type; char *adr;
```

Beschreibung:

testet, ob an der Adresse *adr* ein NULL-Wert des Typs *type* (`fix/fxtypes.h`) steht.

Ergebnis:

1, wenn NULL-Wert
!= 1 sonst

Siehe auch:

fxsetnull()

fxsetnull — NULL-Wert ablegen

Definition:

```
int fxsetnull(type, adr)
int type; char *adr;
```

Beschreibung:

schreibt den NULL-Wert des Typs *type* (`fix/fxtypes.h`) an die Adresse *adr*.

Ergebnis:

0 bei Erfolg
!= 0 bei Fehler

Siehe auch:

fxisnull()

fxtocsql — zu *FIX*-Datentyp korrespondierenden ESQL/C-Datentyp bestimmen

Definition:

int fxtocsql(type)
int type;

Beschreibung:

liefert den dem *FIX*-Datentyp *type* entsprechenden ESQL/C-Datentyp (CCHARTYPE, CFIXCHARTYPE, CSHORTTYPE, CLONGTYPE, CFLOATTYPE, CDOUBLETYPE, CDECIMALTYPE, CDATETYPE, CDTIMETYPE, CINVTYPE).

Ergebnis:

ESQL/C-Datentyp bei Erfolg
-1 bei Fehler

Siehe auch:

sql_to_fx()

Hinweis:

Für FXTRUTHTYPE wird CCHARTYPE zurückgegeben.

Für FXMONEYTYPE wird CDOUBLETYPE zurückgegeben.

sql_to_fx — zu ESQL/C-Datentyp korrespondierenden *FIX*-Datentyp bestimmen

Definition:

int sql_to_fx(sqltype)
int sqltype;

Beschreibung:

liefert den dem ESQL/C-Datentyp *sqltype* (SQLCHAR, SQLNCHAR, SQLSMINT, SQLINT, SQLFLOAT, SQLSMFLOAT, SQLDECIMAL, SQLSERIAL, SQLDATE, SQLMONEY, SQLDTIME, SQLINTERVAL) entsprechenden *FIX*-Datentyp.

Ergebnis:

FIX-Datentyp bei Erfolg
-1 bei Fehler

Siehe auch:

fxtocsql()

Hinweis:

FXTRUTHTYPE wird FXGRAPHICSTYPE kommen als Ergebnis nie vor.

28 Vermischtes

editor — Datei mit Texteditor bearbeiten

Definition:

void editor(file)
char *file;

Beschreibung:

Der Inhalt der Umgebungsvariablen EDITOR wird als Pfad eines Editors interpretiert, der - nach Löschen des Bildschirms und Zurücksetzen der Terminal-Parameter - mit dem Argument *file* (über system()) aufgerufen wird. Im Anschluss wird der Bildschirm wieder restauriert.

Siehe auch:

execute_cmd()

Hinweis:

Ersatzweise wird für EDITOR als Pfad **vi** gesetzt.

Die Funktion editor() ist auch bei Einsatz des grafischen Frontends *FIX/Win* nutzbar. In diesem Fall wird neben dem Inhalt der Datei auch eine zu FXCHARSET passende Windows-Codepage-Nummer an das Frontend übermittelt. Nach dem Editvorgang liefert *FIX/Win* den neuen Dateiinhalt (entsprechend dieser Codepage), der dann von *FIX* in die Datei zurückgeschrieben wird.

execute_cmd — Kommando ausführen

Definition:

```
#include <exec.h>
int execute_cmd(cmdstring, flags)
char *cmdstring; int flags;
```

Beschreibung:

Nach - bei Terminalbetrieb - Löschen des Bildschirms und Zurücksetzen der Terminal-Parameter wird *cmdstring* an die Shell zur Ausführung übergeben.

Der Parameter *flags* wird nur im Terminalbetrieb ausgewertet. Wenn *flags* das Bit E_VERBOSE enthält, so wird das Kommando vor der Ausführung angezeigt. Wenn *flags* das Bit E_WAIT enthält oder das Kommando einen Wert ungleich 0 zurückgibt, so verlangt eine Eingabeaufforderung das Drücken von RETURN oder ^D, bevor die Kontrolle an das *FIX*-Programm zurückgegeben wird.

Ergebnis:

Exitcode des Kommandos
errno nach fehlgeschlagenem fork(), exec() oder wait(), wenn das Kommando nicht ausgeführt werden konnte oder abgebrochen wurde.

Siehe auch:

exec_ms_command()

fx_kal_off — Kalender-Window schließen

Definition:

void fx_kal_off()

Beschreibung:

bringt den *FIX*-Monatskalender zum Verschwinden.

Siehe auch:

fx_kal_on()

fx_kal_on — Kalender-Window öffnen

Definition:

void fx_kal_on()

Beschreibung:

lässt den *FIX*-Monatskalender erscheinen (mit hervorgehobenem aktuellen Tag).

Siehe auch:

fx_kal_off()

Hinweis:

benutzt Dummy-Maske im *FIX*-Verzeichnis.

shell — Shell aufrufen

Definition:

void shell(erase)
int erase;

Beschreibung:

realisiert den Aufruf einer Shell. Als Shell wird der Inhalt der Umgebungsvariablen SHELL genommen, falls nicht gesetzt, der der Umgebungsvariablen shell, falls auch diese nicht gesetzt, der Pfad `/bin/sh`. Vor dem Programmstart werden der Bildschirm gelöscht und die Terminal-Parameter zurückgesetzt. Im Anschluss werden Bildschirm und Terminal-Parameter wieder restauriert.

Siehe auch:

execute_cmd()

Hinweis:

Der Parameter *erase* hat keine Bedeutung und existiert nur zur Kompatibilität mit früheren Versionen.

waitalittle — Startup-Hinweis im Message-Window ausgeben

Definition:

void waitalittle()

Beschreibung:

gibt die Meldung "*** Programm wird geladen ***" aus.

Hinweis:

Die Funktion kann benutzt werden, um dem Anwender zu verdeutlichen, dass das Programm eine längere Initialisierungsphase durchläuft, bevor etwas auf dem Bildschirm erscheint.

Der Text wird `$FXDIR/runtime/messages` entnommen.

29 Grundfunktionen

expand_fopen — Datei zum Lesen öffnen

Definition:

FILE *expand_fopen(name, envvar)
char *name; char *envvar;

Beschreibung:

bildet mittels expandfile(*name*, *envvar*) einen Dateinamen und öffnet die Datei zum Lesen ("r").

Ergebnis:

FILE-Deskriptor bei Erfolg
(FILE *)0 bei Fehler

Siehe auch:

expandfile(), fxfopen(), fxfclose()

Hinweis:

Je nach System wird bei einem Scheitern ein zweiter Versuch unternommen, bei dem die letzte Komponente des Pfades auf 14 Zeichen gekürzt wird.

Da die Funktion einen internen Zähler manipuliert, muss eine mittels expand_fopen() geöffnete Datei mittels fxfclose() statt fclose() geschlossen werden.

expandfile — Dateinamen konstruieren

Definition:

char *expandfile(name, envvar)
char *name; char *envvar;

Beschreibung:

bildet einen Dateinamen aus dem Inhalt der Umgebungsvariablen *envvar*, der als Verzeichnis-Pfad interpretiert wird, und dem Datei-Pfad *name*. Ist *name* kein relativer Pfad, bleibt *envvar* unberücksichtigt. Enthält *envvar* keinen absoluten Pfad, wird zusätzlich das Verzeichnis der Anwendung vorangestellt.

"FXHOME", "HLPPATH", "CHLPPATH" werden über die globalen Variablen S_fxhome, S_hlppath, S_chlppath expandiert.

Ergebnis:

Zeiger auf einen statischen Bereich, der den Dateinamen enthält

Siehe auch:

expand_fopen()

Hinweis:

Der Dateiname darf nicht länger als 100 Zeichen werden.

fx_char_copy — Zeichenfolge übertragen

Definition:

char *fx_char_copy(dest, destsize, src, srcsize, trimright, maxcnt, padchar, addzero)
char *dest; int destsize; char *src; int srcsize; BOOLEAN trimright; int maxcnt; int padchar; BOOLEAN addzero;

Beschreibung:

kopiert die Zeichenfolge an der Adresse *src* (partiell) an die Adresse *dest*. *srcsize* gibt an, wieviele Bytes ab *src* maximal gelesen werden dürfen, *destsize* bestimmt, wieviele Bytes ab *dest* maximal geschrieben werden dürfen. Ist *maxcnt* größer-gleich 0, werden höchstens *maxcnt* Zeichen kopiert.

trimright wird in dieser Version nicht berücksichtigt; als Argument sollte TRUE übergeben werden.

Ist *padchar* ungleich '\0' und *src* ungleich dem NULL-Wert, werden an die kopierte Zeichenfolge Bytes mit Wert *padchar* angefügt, bis die Zeichenfolge an der Adresse *dest destsize* Bytes umfasst; ist *maxcnt* größer-gleich 0, allerdings höchstens soviele, dass sie nicht mehr als *maxcnt* Zeichen enthält.

Ist *padchar* gleich '\0' und *src* ungleich dem NULL-Wert, wird an die Zeichenfolge an der Adresse *dest* '\0' angefügt.

Ist *addzero* ungleich FALSE und *src* ungleich dem NULL-Wert, wird die Zeichenfolge '\0'-terminiert; in diesem Fall sollte *dest*[*destsize*] geschrieben werden dürfen.

Ergebnis:

0 bei Erfolg
< 0 bei Fehler,
> 0- Anzahl der Zeichen von *src*, die nicht kopiert werden konnten

Siehe auch:

fgetstring(), fputval()

Hinweis:

Ist *src* gleich dem NULL-Wert (und *destsize* größer 0), wird auch *dest* NULL zugewiesen.

fx_mktemp — eindeutigen Dateinamen erzeugen

Definition:

```
char *fx_mktemp(name)
char *name;
```

Beschreibung:

erzeugt einen dem Muster *name* (z.B. "LOGFXpppppccc") entsprechenden eindeutigen Dateinamen. Die letzten 8 Zeichen von *name* werden durch die Prozessnummer (fünfstellig) und eine fortlaufende Nummer (dreistellig) ersetzt.

Ergebnis:

name bei Erfolg
(char *)0 bei Fehler

fxfclose — Datei schließen

Definition:

```
int fxfclose(fp)
FILE *fp;
```

Beschreibung:

schließt mittels fclose() die mit dem FILE-Pointer *fp* assoziierte Datei. Scheitert fclose(), erfolgt eine Meldung im Message-Window.

Ergebnis:

0 bei Erfolg
Wert von errno nach fclose()

Siehe auch:

fxfopen(), expand_fopen()

Hinweis:

fxfclose() sollte nur benutzt werden, wenn die Datei durch fxfopen() oder expand_fopen() geöffnet wurde, da die Funktion einen internen Zähler manipuliert.

fxfopen — Datei öffnen

Definition:

FILE *fxfopen(fname, mode)
char *fname; char *mode;

Beschreibung:

öffnet mittels fopen() die Datei *fname* im Modus *mode*. Scheitert fopen(), erfolgt eine Meldung im Message-Window.

Ergebnis:

FILE-Deskriptor bei Erfolg
(FILE *)0 bei Fehler

Siehe auch:

expand_fopen(), fxfclose()

Hinweis:

Je nach System wird die letzte Komponente des Pfades zuvor auf 14 Zeichen verkürzt.

Da die Funktion einen internen Zähler manipuliert, sollte eine mittels fxfopen() geöffnete Datei mittels fxfclose() statt fclose() geschlossen werden.

fxisalnum — Zeichenklasse prüfen

Definition:

int fxisalnum(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zu einer der Zeichenklassen UPPER, LOWER oder DIGIT.

Ergebnis:

!= 0, wenn das Zeichen zu einer der Klassen UPPER, LOWER oder DIGIT gehört
0 sonst

Siehe auch:

fxisupper(), fxislower(), fxisalpha(), fxisdigit()

Hinweis:

Die Klassenzugehörigkeit ist abhängig vom benutzten Zeichensatz.

fxisalpha — Zeichenklasse prüfen

Definition:

int fxisalpha(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zu einer der Zeichenklassen UPPER oder LOWER.

Ergebnis:

!= 0, wenn das Zeichen zu einer der Klassen UPPER oder LOWER gehört
0 sonst

Siehe auch:

fxisupper(), fxislower()

Hinweis:

Die Klassenzugehörigkeit ist abhängig vom benutzten Zeichensatz.

fxisblank — Zeichenklasse prüfen

Definition:

int fxisblank(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zur Zeichenklasse BLANK.

Ergebnis:

!= 0, wenn das Zeichen zur Klasse BLANK gehört
0 sonst

Siehe auch:

fxisprint(), fxisspace()

Hinweis:

Die Klassenzugehörigkeit ist abhängig vom benutzten Zeichensatz.

fxiscntrl — Zeichenklasse prüfen

Definition:

int fxiscntrl(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zur Zeichenklasse CONTROL.

Ergebnis:

!= 0, wenn das Zeichen zur Klasse CONTROL gehört
0 sonst

Siehe auch:

fxisprint()

Hinweis:

Die Klassenzugehörigkeit ist abhängig vom benutzten Zeichensatz.

fxisdigit — Zeichenklasse prüfen

Definition:

int fxisdigit(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zur Zeichenklasse DIGIT.

Ergebnis:

!= 0, wenn das Zeichen zur Klasse DIGIT gehört
0 sonst

Siehe auch:

fxisxdigit(), fxisalnum()

fxisgraph — Zeichenklasse prüfen

Definition:

int fxisgraph(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zu einer der Zeichenklassen UPPER, LOWER, DIGIT oder PUNCTUATION.

Ergebnis:

!= 0, wenn das Zeichen zu einer der Klassen UPPER, LOWER, DIGIT oder PUNCTUATION gehört
0 sonst

Siehe auch:

fxisupper(), fxislower(), fxisdigit(), fxisalnum(), fxispunct()

Hinweis:

Die Klassenzugehörigkeit ist abhängig vom benutzten Zeichensatz.

fxislower — Zeichenklasse prüfen

Definition:

int fxislower(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zur Zeichenklasse LOWER.

Ergebnis:

!= 0, wenn das Zeichen zur Klasse LOWER gehört
0 sonst

Siehe auch:

fxisupper(), fxisalpha(), fxtolower()

Hinweis:

Die Klassenzugehörigkeit ist abhängig vom benutzten Zeichensatz.

fxisprint — Zeichenklasse prüfen

Definition:

int fxisprint(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zu einer der Zeichenklassen UPPER, LOWER, DIGIT, PUNCTUATION oder BLANK.

Ergebnis:

!= 0, wenn das Zeichen zu einer der Klassen UPPER, LOWER, DIGIT, PUNCTUATION oder BLANK gehört
0 sonst

Siehe auch:

fxisupper(), fxislower(), fxisdigit(), fxispunct()

Hinweis:

Die Klassenzugehörigkeit ist abhängig vom benutzten Zeichensatz.

fxispunct — Zeichenklasse prüfen

Definition:

int fxispunct(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zur Zeichenklasse PUNCTUATION.

Ergebnis:

!= 0, wenn das Zeichen zur Klasse PUNCTUATION gehört
0 sonst

Siehe auch:

fxisgraph(), fxisprint()

Hinweis:

Die Klassenzugehörigkeit ist abhängig vom benutzten Zeichensatz.

fxisspace — Zeichenklasse prüfen

Definition:

int fxisspace(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zu einer der Zeichenklassen SPACE oder BLANK.

Ergebnis:

!= 0, wenn das Zeichen zu einer der Klassen SPACE oder BLANK gehört
0 sonst

Siehe auch:

fxisblank(), fxisprint()

Hinweis:

Die Klassenzugehörigkeit ist abhängig vom benutzten Zeichensatz.

fxisupper — Zeichenklasse prüfen

Definition:

int fxisupper(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zur Zeichenklasse UPPER.

Ergebnis:

!= 0, wenn das Zeichen zur Klasse UPPER gehört
0 sonst

Siehe auch:

fxislower(), fxisalpha(), fxtoupper()

Hinweis:

Die Klassenzugehörigkeit ist abhängig vom benutzten Zeichensatz.

fxisxdigit — Zeichenklasse prüfen

Definition:

int fxisxdigit(code)
int code;

Beschreibung:

testet die Zugehörigkeit des Zeichens *code* zur Zeichenklasse XDIGIT.

Ergebnis:

!= 0, wenn das Zeichen zur Klasse XDIGIT gehört
0 sonst

Siehe auch:

fxisdigit()

fxtolower — Zeichen in Kleinschreibung umwandeln

Definition:

int fxtolower(c)
int c;

Beschreibung:

konvertiert Großbuchstaben in Kleinbuchstaben.

Ergebnis:

falls *c* ein Großbuchstabe ist, den entsprechenden Kleinbuchstaben
c sonst

Siehe auch:

fxtoupper(), fxislower(), makelower()

Hinweis:

legt den von der Anwendung benutzten Zeichensatz zugrunde.

fxtoupper — Zeichen in Großschreibung umwandeln

Definition:

int fxtoupper(c)
int c;

Beschreibung:

konvertiert Kleinbuchstaben in Großbuchstaben.

Ergebnis:

falls *c* ein Kleinbuchstabe ist, den entsprechenden Großbuchstaben
c sonst

Siehe auch:

fxtolower(), fxisupper(), makeupper()

Hinweis:

legt den von der Anwendung benutzten Zeichensatz zugrunde.

fxreg — regulären Ausdruck in String erkennen

Definition:

BOOLEAN fxreg(info, pattern)
char *info; char *pattern;

Beschreibung:

sucht im String *info* nach einem der in *pattern* enthaltenen, durch '|' getrennten regulären Ausdrücke.

Ergebnis:

TRUE, wenn *info* einen Teilstring enthält, der zu einem der regulären Ausdrücke passt
FALSE sonst

fxvalcmp — Feld-Hostvariablen-Werte vergleichen

Definition:

int fxvalcmp(t1, t2, type, len)
char *t1; char *t2; int type; int len;

Beschreibung:

vergleicht die Werte an den Adressen *t1* und *t2*, an denen Werte vom *FIX*-Typ *type* erwartet werden. Der Parameter *len* wird nur beim Typ FXGRAPHICSTYPE ausgewertet.

Ergebnis:

-2 - Fehler (Vergleich von FXINVTYPE-Werten mit inkompatiblen Qualifiern)
-1 - **t1* < **t2*
0 - **t1* == **t2*
1 - **t1* > **t2*

Siehe auch:

notrailcmp()

Hinweis:

Beim Typ FXCHARTYPE werden Leerzeichen am Ende ignoriert. Die Ordnung der Zeichen entspricht der von notrailcmp().

Beim Typ FXGRAPHICSTYPE werden *len* korrespondierende Codes mittels memcpy() verglichen.

Beim Typ FXTRUTHTYPE gilt “Nein” als kleiner als “Ja”.

Beim Typ FXDTIMETYPE werden, sind die Qualifier von ∗*t1* und ∗*t2* verschieden, beide Werte so erweitert, dass sie die Komponenten Jahr bis Sekunde umfassen.

Beim Typ FXINVTYPE werden, sind die Qualifier kompatibel, beide Werte entweder in Monate oder in Sekunden konvertiert und die sich so ergebenden Werte verglichen.

NULL wird als gleich mit NULL und kleiner als alles andere behandelt.

inint — Wert in int-Array finden

Definition:

int inint(i, ii)
int i; int ∗ii;

Beschreibung:

sucht im int-Array *ii*, das mit einem Element endet, das 0 enthält, nach einem Element, das den Wert *i* enthält (*i* != 0).

Ergebnis:

i, wenn *i* in *ii* vorkommt
0 sonst

Siehe auch:

instring()

instring — Zeichen in String finden

Definition:

int instring(c, str)
int c; char ∗str;

Beschreibung:

sucht im String *str*, der mit ‘\0’ endet, nach dem Zeichen *c* (*c* != '\0').

Ergebnis:

Index des ersten Vorkommens von *c* in *str* (beginnend mit 1)
0, wenn *c* nicht in *str* vorkommt

Siehe auch:

inint()

makelower — String in Kleinschreibung umwandeln

Definition:

void makelower(str)
char ∗str;

Beschreibung:

wandelt Großbuchstaben im String *str* in Kleinbuchstaben um.

Siehe auch:

fxtolower()

Hinweis:

legt den von der Anwendung benutzten Zeichensatz zugrunde.

makeupper — String in Großschreibung umwandeln

Definition:

void makeupper(str)
char *str;

Beschreibung:

wandelt Kleinbuchstaben im String *str* in Großbuchstaben um.

Siehe auch:

fxtoupper()

Hinweis:

legt den von der Anwendung benutzten Zeichensatz zugrunde.

notrailcmp — Strings vergleichen

Definition:

int notrailcmp(s1, s2)
char *s1; char *s2;

Beschreibung:

Wie strcmp(), jedoch werden bei beiden Argumenten Leerzeichen am Ende ignoriert.

Ergebnis:

< 0 - *s1* ist lexikografisch kleiner als *s2*
== 0 - *s1* und *s2* stimmen überein
> 0 - *s1* ist lexikografisch größer als *s2*

Hinweis:

notrailcmp() darf aufgrund der Kodierung von NULL nicht auf die Werte von FXCHARTYPE- oder FXTRUTHTYPE-Feldern angewendet werden.

putstrlen — Länge der Darstellung eines *FIX*-String bestimmen

Definition:

int putstrlen(s)
char *s;

Beschreibung:

zählt die darstellbaren Zeichen im String *s*.

Ergebnis:

Anzahl der darstellbaren Zeichen

Siehe auch:

putstr(), putstrwidth()

Hinweis:

Wo möglich, ist putstrwidth() zu bevorzugen.

putstrwidth — Platz für die Darstellung eines *FIX*-Strings bestimmen

Definition:

```
int putstrwidth(s)
char *s;
```

Beschreibung:

liefert den Platzbedarf (in Bildschirmspalten) zur Darstellung der Zeichen im String *s*.

Ergebnis:

Anzahl der Spalten

Siehe auch:

putstr(), putstrlen(), fxisprint()

Hinweis:

berücksichtigt Videoattribut- oder Zeichensatz-Umschaltung und Zeichenbreite.

strapp — Strings konkatenieren

Definition:

```
char *strapp(s1, s2)
char *s1; char *s2;
```

Beschreibung:

kopiert wie strcpy() *s2* nach *s1*, zurückgegeben wird jedoch nicht *s1*, sondern das "Ende" von *s1*, also ein Zeiger auf das Null-Byte des Ergebnisses. Dieser kann für einen weiteren strapp()-Aufruf benutzt werden, um so Strings zu konkatenieren.

Beispiel:

```
cp = strapp(ziel, str1);      /* kopiert str1 nach ziel ... */
strapp (cp, str2);            /* ... und hängt str2 an */
```

Ergebnis:

&*s1*[strlen(*s2*)]

Hinweis:

strapp() darf aufgrund der Kodierung von NULL nicht auf die Werte von FXCHARTYPE- oder FXTRUTHTYPE-Feldern angewendet werden.

strnswap — Strings vertauschen

Definition:

```
void strnswap(s1, s2, n)
char *s1; char *s2; int n;
```

Beschreibung:

vertauscht die ersten *n* Bytes der Speicherflächen *s1* und *s2*.

strsave — Stringkopie anlegen

Definition:

```
char *strsave(s)
char *s;
```

Beschreibung:

kopiert den String *s* in einen neu allokierten Speicherbereich.

Ergebnis:

Zeiger auf die Kopie von *s* bei Erfolg
(char *)0 bei Fehler

strtruncate — String verkürzen

Definition:

```
char *strtruncate(src)
char *src;
```

Beschreibung:

ersetzt im String *src* Leerzeichen am Ende durch Nullbytes ('\0').

Ergebnis:

src

Hinweis:

strtruncate() darf aufgrund der Kodierung von NULL nicht auf die Werte von FXCHARTYPE- oder FXTRUTHTYPE-Feldern angewendet werden.

trycalloc — dynamisch Speicher allokieren

Definition:

```
char *trycalloc(size)
unsigned int size;
```

Beschreibung:

allokiert mittels calloc() Speicherplatz für *size* Bytes und eine zusätzliche Markierung, durch die Fehler bei Speicherzugriffen leichter zu finden sind.

Ergebnis:

Zeiger auf die vom Programm nutzbaren *size* Bytes bei Erfolg
(char *)0 bei Fehler

Siehe auch:

tryrealloc(), tryfree()

Hinweis:

Im Fehlerfall wird eine Meldung ausgegeben.

Mittels trycalloc() allokierter Speicherplatz darf nur mittels tryfree() freigegeben werden.

tryfree — dynamisch allokierten Speicher freigeben

Definition:

void tryfree(pptr)
char *pptr;

Beschreibung:

gibt mittels trycalloc() allokierten Speicherplatz frei.

Siehe auch:

trycalloc(), tryrealloc()

Hinweis:

Nur mittels trycalloc() oder tryrealloc() allokierter Speicherplatz darf mittels tryfree() freigegeben werden.

tryrealloc — dynamisch allokierten Speicher redimensionieren

Definition:

char *tryrealloc(pptr, size)
char *pptr; unsigned int size;

Beschreibung:

vergrößert oder verkleinert einen durch trycalloc() allokierten Speicherbereich, der dazu u.U. verschoben wird; die darin abgelegten Daten bleiben so weit erhalten, wie es die neue Länge zulässt.

Ergebnis:

Zeiger auf den neuen Speicherbereich der Daten bei Erfolg
(char *)0 bei Fehler

Siehe auch:

trycalloc(), tryfree()

Hinweis:

Im Fehlerfall wird eine Meldung ausgegeben und die Daten an der Adresse *pptr* bleiben erhalten.

Nur mittels trycalloc() oder tryrealloc() allokierter Speicherplatz darf mittels tryrealloc() reallokiert werden.

typecmp — Feld-Hostvariablen vergleichen †

Definition:

int typecmp(t1, t2, type)
char *t1; char *t2; int type;

Beschreibung:

vergleicht die Werte an den Adressen *t1* und *t2*, an denen Werte vom *FIX*-Typ *type* erwartet werden, mittels fxvalcmp().

Ergebnis:

-2 - Fehler (illegaler Typ, Vergleich von FXINVTYPE-Werten mit inkompatiblen Qualifiern)
-1 - **t1* < **t2*
0 - **t1* == **t2*
1 - **t1* > **t2*

Siehe auch:

fxvalcmp()

Hinweis:

Der Typ FXGRAPHICSTYPE wird nicht unterstützt.

Die Verwendung von fxvalcmp() ist deshalb zu bevorzugen.

44 Globale Variablen und Ressourcen

Das Verhalten einer Anwendung kann durch globale Variablen und das Belegen von Ressourcen mit bestimmten Werten gesteuert werden. Ressourcen sind in einer Ressourcedatei zu setzen oder über eine Umgebungsvariable zu definieren. Globale Variablen werden von *FIX* definiert und können in der Anwendung abgefragt werden und in einigen Fällen verändert werden.

1 Globale Variablen

Die jenigen globalen Variablen, die mit **readonly** markiert sind dürfen von der Anwendung nur gelesen werden. Alle mit **r/w** markierten Variablen dürfen auch verändert werden. Die folgenden Abschnitte beschreiben die Variablen in alphabetischer Reihenfolge.

field ∗_f: readonly

Sofern eine Maske mittels perform() bearbeitet wird, hinterlegt *FIX* in dieser Variablen die Adresse des zum Besuch anstehenden (L_ENTERFIELD) bzw. gerade besuchten Elements. Wenn kein aktuelles Maskenelement existiert, hat die Variable den Wert (field ∗)0. Bei einer eingebetteten Maske ist ein geeigneter Cast des Zeigers erforderlich.

short ∗_fnr: r/w

Sofern eine Maske mittels perform() bearbeitet wird und hierbei ein *aktuelles* Element besitzt, enthält diese Variable die Adresse der Komponente fnr der bearbeiteten Maske (mfo-Nummer des aktuellen Elements). Bei Masken, die keine Elemente besitzen, hat die fnr-Komponente den Wert -1. Der Wert von ∗_fnr darf nur mittels der von *FIX* bereitgestellten Makros und Funktionen geändert werden.

BOOLEAN allowAssignFormat:r/w

Diese Variable existiert, um die Kompatibilität mit älteren *FIX*-Versionen zu gewährleisten. Wenn sie auf den Wert 1 gesetzt wird, dann ist ein Aufruf der Funktion f_AssignFormat() erlaubt, obwohl das Bit 1 der Ressource SwapOptions nicht gesetzt ist.

int B_CodesetRequired: readonly

Diese Variable wird bei der Initialisierung auf 0 gesetzt und darf nicht verändert werden.

BOOLEAN B_dev: readonly

wird von der Funktion arginspect() auf TRUE gesetzt, wenn der Schalter “-dev” angegeben wird. Anderenfalls hat die Variable den Wert FALSE.

B_dev == TRUE hat folgende Wirkung:

- Die Taste g8 startet den Editor nicht auf der “privaten” Hilfedatei (in `$CHLPPATH`), sondern auf der Hilfedatei der Anwendung (in `$HLPPATH`).
- Die Event-Testhilfe (vgl. Seite 529) ist verwendbar.

Es ist geplant, den Grad interner Plausibilitätsprüfungen u. ä. zukünftig ebenfalls von B_dev abhängig zu machen.

int B_numeric_logic: r/w

steuert die Art, in der ein numerisches Feld mit Formatangabe erfasst wird. Die Variable ist mit 0 vorbelegt.

Hat B_numeric_logic den Wert 0, so werden die Felder in der auf Seite 42 beschriebenen typabhängigen Weise erfasst. Beim Wert 1 erfolgt auch die Erfassung von FXMONEYTYPE-Felder in der für alle sonstigen Felder benutzten Weise. Beim Wert 2 wird generell die sonst nur für FXMONEYTYPE-Felder verwendete Methode benutzt. Andere Werte sind unzulässig.

BOOLEAN B_pfennig: r/w

wird von der Funktion arginspect() auf den Wert FALSE gesetzt, wenn der Schalter "-DM" angegeben wird. Ist der Schalter nicht angegeben, hat die Variable den Wert TRUE.

Hat B_pfennig den Wert FALSE, wird beim Besuch eines FXMONEYTYPE-Feldes, das keine Formatangabe besitzt, vor dem Verlassen nicht automatisch vor den letzten beiden Ziffern ein Dezimalpunkt eingefügt, wenn der Feldinhalt keinen Dezimalpunkt aufweist.

Eine Änderung des Wertes sollte ggf. zwischen den Aufrufen von arginspect() und fxinit() erfolgen.

BOOLEAN B_service: readonly

wird von der Funktion arginspect() auf den Wert TRUE gesetzt, wenn der Schalter "-service" angegeben wird. Ist der Schalter nicht angegeben, hat die Variable den Wert FALSE.

char *B_shell: r/w

wird von der Funktion arginspect() auf den Wert FALSE gesetzt, wenn der Schalter "-sh" angegeben wird. Ist der Schalter nicht angegeben, hat die Variable den Wert TRUE.

Hat B_shell den Wert FALSE, erlaubt *FIX* kein Verzweigen in die Shell als Standardreaktion auf die Taste K_sh.

Eine Änderung des Wertes kann jederzeit erfolgen.

BOOLEAN B_transaction: readonly

ist mit FALSE vorbelegt. B_transaction erhält den Wert TRUE, wenn die Datenbank, zu der mittels fx_database_connect() eine Verbindung hergestellt wurde, über Transaktionssicherung verfügt. Wird die Datenbankverbindung mittels fx_database_disconnect() beendet, wird der Wert auf FALSE zurückgesetzt.

obj *c_objp: readonly

Diese Variable enthält die Adresse des augenblicklich bearbeiteten Objekts. Wenn kein Objekt aktiv ist, hat die Variable den Wert (obj *)0.

BOOLEAN db_connected: readonly

ist mit FALSE vorbelegt. db_connected erhält den Wert TRUE, wenn mittels fx_database_connect() erfolgreich eine Datenbankverbindung hergestellt wurde. Wird die Datenbankverbindung mittels fx_database_disconnect() beendet, wird der Wert auf FALSE zurückgesetzt.

int F_test: r/w

wird von der Funktion arginspect() entsprechend dem Wert des Schalters "-ftest *n*" gesetzt. Ist der Schalter nicht angegeben, hat die Variable den Wert 0.

Der Entwickler kann diese Variable benutzen, um eigene Test- oder Warnstufen zu realisieren. *FIX* wertet diese Variable niemals aus.

int fxday_tab[2][13]

fxdaytab[0][*i*] ($1 \le i \le 12$) enthält die Anzahl der Tage des *i*-ten Monats eines Nicht-Schaltjahres, fxdaytab[1][*i*] die eines Schaltjahres.

char *fxdbname: readonly

In dieser Variable hinterlegt *FIX* einen Zeiger auf den Namen der Datenbank (Kopie des Arguments von fx_database_connect() bzw. des Wertes der Umgebungsvariablen DBNAME), wenn mittels fx_database_connect() erfolgreich eine Datenbankverbindung hergestellt wurde. Beim Beenden der Datenbankverbindung wird der Wert *nicht* zurückgesetzt.

int IN_transaction: readonly

wird von *FIX* mit -1 vorbelegt. Sobald mittels fx_database_connect() eine Verbindung zu einer Datenbank hergestellt wurde, die über Transaktionssicherung verfügt, erhält IN_transaction den Wert 0 (“in Basistransaktion”). Wird die Datenbankverbindung mittels fx_database_disconnect() beendet, wird der Wert auf -1 zurückgesetzt.

Benutzt die Anwendung virtuelle Transaktionen (vgl. Seite 136), so wird IN_transaction bei jedem Aufruf von fx_database_commit() um 1 erhöht, bei jedem Aufruf von fx_database_commit() bzw. fx_database_rollback() vermindert. Wird hierbei 0 erreicht, wird die aktuelle Datenbanktransaktion bestätigt oder zurückgesetzt und eine neue begonnen.

Achtung:

Über den Wert von IN_transaction entscheidet *FIX*, ob vor dem Beenden der Datenbankverbindung (close database) eine Transaktion beendet werden muss (rollback work).

int NOBJ: r/w

Der Wert dieser Variablen (Vorbelegung: 128) wird von *FIX* zur Dimensionierung von Datenstrukturen benutzt, in denen er über die augenblicklich geladenen Masken (außer Varianten) Buch führt. Mehrfach geladene Masken benötigen auch mehrere Einträge.

Eine Erhöhung des Wertes muss vor dem Aufruf von fxinit() erfolgen.

char *Prg_version: r/w

Der Entwickler kann in dieser Variablen einen Text hinterlegen, den *FIX* ausgibt, wenn das Programm mit dem Schalter -version aufgerufen wird. Hat die Variable den Wert (char *)0 (Vorbelegung), erfolgt keine Ausgabe.

Eine Änderung des Wertes muss vor dem Aufruf von arginspect() erfolgen.

int (* S_ApplicationHelpHandler(/* char *name */): r/w

Diese Variable wird von den *FIX*-Funktionen help() und n_help() ausgewertet. Sofern ihr Wert ungleich (int (*)(char *))0 ist, wird die Funktion, auf die die Variable zeigt, mit dem Namen des Hilfetextes als Argument aufgerufen. Gibt sie 0 zurück, unterbleibt das übliche Anzeigen einer Hilfetext-Datei in einem Window und der Aufruf von help() bzw. n_help() kehrt zurück.

Ergebnis:

0 - Hilfe-Routine verlassen
!= 0 - im Code der Hilfe-Routine fortfahren.

Hinweis:

FIX ruft die Funktion u.U. mit dem Argument (char *)0 auf, wenn aus dem Kontext kein Name für einen Hilfetext bestimmt werden kann.

(*S_ApplicationHelpHandler)(*name*) könnte beispielsweise einen Browser bedienen.

struct clickinfo *S_BtLeftFromField: r/w

Zeigt diese globale Variable (Vorbelegung: (struct clickinfo *)0) auf eine clickinfo-Struktur (vgl. Seite 159), bleibt die Vorgabe, wie die aktive Maske auf BT_LEFT reagieren soll, unbeachtet, und bei der Behandlung des Events wird nicht nur das aktive Objekt berücksichtigt. Vielmehr kommt als Ziel eine an dieser Position sichtbare (d.h. nicht überdeckte) Stelle eines Feldes einer beliebigen Maske bzw. eines Menüpunktes eines beliebigen Menüs in Betracht.

BOOLEAN (*S_ConvertFieldValue)(field *f, int dir, char *from , char *to): r/w

Wenn an der Adresse S_ConvertFieldValue eine Funktion hinterlegt wird, dann wird diese Funktion von *FIX* bei der Felderfassung und bei der Darstellung von Feldwerten aufgerufen.

Sie hat die Aufgabe, den abgespeicherten Wert oder den angezeigten Wert umzurechnen. Wird als dir (direction) der Wert DIR_INFO_TO_WRK angegeben, dann muss eine Umrechnung des angezeigten Wertes (info) in den abgespeicherten Wert (wrk) erfolgen. Bei dem Wert DIR_WRK_TO_INFO muss eine Umrechnung in der umgekehrten Richtung erfolgen.

Der Originalwert steht in dem Parameter from. Der Datentyp ergibt sich aus dem Datentyp des Feldes (f). An der Adresse to muss der umgerechnete Wert in dem gleichen Datentyp hinterlegt werden. *FIX* stellt an der Adresse to vor Aufruf der Funktion eine dem Datentyp entsprechende Speicherfläche bereit. Das Feld f kann zur Ermittlung weiterer Informationen verwendet werden (ob_props).

Die Funktion wird für alle Felder aufgerufen. Soll für ein Feld keine Umrechnung vorgenommen werden, dann kann der Wert an der Adresse to unbesetzt bleiben und mit FALSE zurückgekehrt werden. *FIX* verwendet in diesem Fall den unveränderten Wert an der Adresse from.

int (*S_DefineFieldButton)(field *f, BOOLEAN active): r/w

In der Variablen S_DefineFieldButton kann die Adresse einer selbst erstellten Funktion hinterlegt werden, die beim Zeichnen des Feldes aufgerufen wird und als Parameter den Zeiger auf das Feld und ein Flag bekommt, das besagt ob das Feld gerade aktiv ist. Der Rückgabewert dieser Funktion wird als Index des Buttons bewertet. Ist dieser -1, wird kein Button dargestellt. Ansonsten werden die Zeichencodes verwendet, die mittels defineButtonSet() definiert wurden.

BOOLEAN (*S_GetTableFieldCodes)(field *f, unsigned char *delim): r/w

Zur Darstellung von Feldern als Tabellenzellen kann in dieser Variablen die Adresse einer selbst erstellten Funktion hinterlegt werden. Die Funktion hat die Aufgabe, die Semigrafikzeichen zu definieren, die für ein Feld als Begrenzer und Feldfüllzeichen verwendet werden. Dazu muss sie in dem Speicherbereich auf den delim zeigt eine Zeichenkette mit folgendem Aufbau ablegen:

```
[]_^0
```

wobei die Zeichen die folgende Bedeutung haben:

[- linker Feldbegrenzer
] - rechter Feldbegrenzer
_ - Leerstelle im Feld
^ - Füllstelle im Feld
0 - 0-Byte

Zur Bestimmung der Zeichen kann der Zeiger f auf das Feld ausgewertet werden. Wird der Bereich nicht mit eigenen Codes gefüllt, dann werden die von *FIX* standardmäßig definierten Begrenzer verwendet. Über den Rückgabewert steuert die Funktion, ob das Feld als Tabellenfeld oder als normales Feld gezeichnet werden soll. Je nachdem wird ein anderes Attribut zur Darstellung in *FIX/Win* verwendet. Ein Wert von TRUE definiert ein Tabellenfeld. Die in delim definierten Codes werden jedoch unabhängig davon immer verwendet.

Die Funktion wird nur aufgerufen, wenn *FIX/Win* als Frontend verwendet wird.

Event S_HideSeloKey: r/w

Enthält diese Variable ein Tasten-Event, führt das Erkennen dieses Events in der Auswahlphase der Selobearbeitung dazu, dass *FIX* das Selo bis zum Erkennen eines beliebigen weiteren Events nicht am Bildschirm darstellt. Bei Verwendung der Event-Kodierung eines darstellbaren Zeichens als Wert von S_HideSeloKey greift dieser Mechanismus nicht.

Mehr hierzu in Abschnitt 3 auf Seite 254.

int S_MB_CUR_MAX: readonly

Diese Variable wird bei der Initialisierung auf 1 gesetzt und darf nicht verändert werden.

Event (*S_ReadEvent)(/* void */): r/w

zeigt auf die Funktion, die von dcgetch() benutzt werden soll, um ein Event vom Eingabestrom zu lesen. Die Variable ist mit der (Adresse der) Funktion fxReadEvent() vorbelegt.

void (*S_SetFormatHook)(field *f): r/w

In dieser globalen Variablen kann eine Funktion der Anwendung hinterlegt werden, die *FIX* jedesmal aufruft, bevor ein Format zur Darstellung eines Feldwertes verwendet wird. Die Anwendung hat somit die Möglichkeit, das Format zu ändern.

Dazu sind die Funktionen f_AssignFormat(), f_RemoveFormat() und f_RestoreFormat() zu verwenden.

void (*S_SwapPtrval)(obj *o1, obj *o2)

Wenn in dieser Variable die Adresse einer Funktion hinterlegt wird, dann ist diese Funktion für das Vertauschen der Werte in der Struktur ob_props beim Variantenwechsel verantwortlich. Die Funktion wird für die Maske und für jedes Feld aufgerufen. Sie bekommt die Masken- bzw. die Feldzeiger als Parameter. Voraussetzung für das Vertauschen der Werte ist, dass in der Ressource SwapOptions das Bit 2 gesetzt ist.

Wird in der Variablen der Wert NULL hinterlegt, dann übernimmt *FIX* das Vertauschen der Werte in ob_props.

int S_TrailingNonASCIIBlanksMode: readonly

Diese Variable wird bei der Initialisierung auf 0 gesetzt und darf nicht verändert werden.

BOOLEAN S_backwards_rollback: r/w

Diese Variable (Vorbelegung: TRUE) dient der Kompatibilität von perf() mit älteren Versionen von *FIX*. Gewöhnlich hinterlegt perf() analog zu einer generierten Source beim “Zurückgehen vor das Fetch-Element” den Wert ROLLBACK in der restart-Komponente (vgl. Source-Beispiel). Ist dieses Verhalten nicht zweckmäßig, so sollte S_backwards_rollback der Wert FALSE zugewiesen werden, was bewirkt, dass stattdessen NO_ROLLBACK hinterlegt wird.

Eine Änderung des Wertes von S_backwards_rollback wird beim nächsten Auftreten dieser Situation wirksam.

char *S_charset: readonly

Diese Variable zeigt nach der der Initialisierung auf den Wert der Umgebungsvariablen FXCHARSET (oder ersatzweise auf die Konstante "ISO-15"). *FIX* erwartet, dass sich der referenzierte Wert nicht verändert.

int S_chlen_to_dblen_factor: readonly

Diese Variable wird bei der Initialisierung auf 1 gesetzt und darf nicht verändert werden.

char *S_chlppath: r/w

FIX hält intern Zeiger auf die Werte einiger Umgebungsvariablen, die häufig benötigt werden. Diese werden beim Aufruf von fxinit() belegt. Benutzt werden diese Variablen beispielsweise bei der Bildung von Dateinamen (→ expandfile()), statt die Pfade erneut in der Umgebung nachzuschlagen.

FIX speichert in S_chlppath einen Zeiger auf einen Puffer mit dem Wert von CHLPPATH.

int S_cols: readonly

wird von der Funktion fx_init() entsprechend der Angabe “co” in der Bildschirmbeschreibungsdatei gesetzt.

int S_field_insert_mode: r/w

Diese Variable (Vorbelegung: TRUE) wird zu Beginn der Erfassung eines FXCHARTYPE-Feldes ausgewertet. Besitzt sie den Wert FALSE, ignoriert *FIX* bei dieser Erfassung die Einstellung der Ressource UseFieldInsertMode (vgl. Seite 449) und verwendet den herkömmlichen Overwrite-Modus (vgl. “Bedienungsmodi” auf Seite 34). Es ist Aufgabe der Anwendung, die Variable nach der Felderfassung oder spätestens vor der Erfassung des nächsten Feldes wieder auf den Ursprungswert zu setzen, wenn dies notwendig ist.

Das Setzen von S_field_insert_mode hat nur Wirkung, wenn die Ressource UseFieldInsertMode den Wert TRUE besitzt.

void (*S_fitwindow)(field *f, box_cb *hp_box): r/w

An diesem Zeiger ist standardmäßig die Adresse der Funktion fitwindow() hinterlegt, die für die Positionierung von Hilfetexten und Selos zuständig ist. Soll die seit Version 3.0.0 verbesserte Positionierung genutzt werden, so muss der Zeiger auf die Funktion fitwindow_new() zeigen. Die Positionierung kann auch von der Anwendung selbst durchgeführt werden, wenn an dieser Adresse eine eigene Funktion hinterlegt wird, die die entsprechende Position in die übergebene box_cb-Struktur einträgt.

char *S_fxhome: r/w

Wie S_chlppath, nur dass hier ein Zeiger auf einen Puffer mit dem Wert von FXHOME gespeichert ist. Ist FXHOME nicht gesetzt, zeigt S_fxhome auf einen Leerstring.

Achtung:

Ein Wechsel des Anwendungsverzeichnisses im laufenden Programm ist durch Umsetzen dieser Variablen (und ggf. entsprechender Aufrufe von putenv()) prinzipiell möglich.

char *S_hardcopy: r/w

Über diese Variable kann gesteuert werden, an welchen Filter *FIX* Hardcopies übergibt. Hat S_hardcopy den Wert (char *)0 (Vorbelegung), so erwartet die Hardcopy-Routine den Pfad des Filterprogramms in der Umgebungsvariable HARDCOPY.

Wird S_hardcopy gesetzt, muss die Variable auf den Pfadnamen eines Programms zeigen, das mittels popen() gestartet werden kann und von der Standardeingabe die geräteunabhängige Form einer Hardcopy (vgl. **lpcat**) annimmt.

Wird weder im Programm noch in der Umgebung ein Filterprogramm spezifiziert, so legt *FIX* die Hardcopy in einer Datei mit dem Präfix DMP im aktuellen Verzeichnis ab.

char *S_hlppath: r/w

Wie S_chlppath, nur dass hier ein Zeiger auf einen Puffer mit dem Wert von HLPPATH gespeichert ist.

int S_lines: readonly

wird von der Funktion fx_init() entsprechend der Angabe “li” in der Bildschirmbeschreibungsdatei gesetzt.

char *S_logfile: r/w

Über diese Variable kann gesteuert werden, wohin *FIX* Datenbankfehler protokolliert (→ sql_error(), fx_sql_error()). Hat S_logfile den Wert (char *)0 (Vorbelegung), so erfolgt die Ausgabe in die Datei LOGFILE im Anwendungsverzeichnis.

Soll die Protokollierung in eine andere Datei erfolgen, so muss S_logfile auf den Namen einer Datei zeigen, die *FIX* zum Schreiben öffnen oder anlegen kann. Um die Protokollierung zu unterdrücken, sollte man S_logfile auf die Zeichenkette "/dev/null" zeigen lassen.

Eine Änderung des Wertes kann jederzeit erfolgen, da *FIX* die Datei für jeden Eintrag neu öffnet.

Achtung:

Ein relativer Dateiname wird bzgl. des aktuellen Verzeichnisses, nicht des Anwendungsverzeichnisses interpretiert. Muss die Datei angelegt werden, erhält sie die Zugriffsrechte "rw-rw-rw-".

char S_months[12][4]: readonly

Diese Variable enthält die Abkürzungen für die Monate (Jan - Dez). Diese sind der Datei `$FXDIR/runtime/messages` (bzw. `$FXRUNTIMEDIR/messages`) entnommen.

int S_newtabmode: r/w

Die oberste und unterste Zeile einer Tabellenmaske werden von *FIX* nicht zur Darstellung von Sätzen verwendet, selbst wenn die Tabellenmaske keinen Rahmen besitzt. Ab Version 2.9.0 ist dies für die unterste Zeile jedoch erreichbar, wenn die Variable S_newtabmode (Vorbelegung: 0) zum Zeitpunkt des Ladens der Maske einen Wert ungleich 0 hat.

Achtung:

Die Utility **mdemo** unterstützt dieses Feature nicht.

void (*S_pa_mark)(obj *objp): r/w

In dieser Variablen kann die Adresse einer Funktion hinterlegt werden, die zur Erzeugung von Paintareas nach dem Darstellen eines Objektes von *FIX* aufgerufen wird. Weitere Hinweise finden sich im Abschnitt "Erzeugen von Paintareas" auf Seite 220.

int S_pa_max_tabs: r/w
int S_pa_tab_size: r/w

Über diese beiden Variablen kann der Cache für Paintareas eingestellt werden. S_pa_max_tabs bestimmt die maximale Anzahl an Einträgen. Jedes sichtbare Objekt benötigt einen Eintrag. S_pa_tab_size bestimmt die anfängliche Anzahl Paintareas pro Objekt. Diese Anzahl wird für jedes Objekt reserviert und bei Bedarf vergrößert. Der Default für beide Werte ist 20. Weitere Hinweise finden sich im Abschnitt "Konfiguration des Paintarea-Caches" auf Seite 228.

char * (*S_pa_ptrval_text)(char *ptrval, int pttype): r/w

In dieser Variablen kann die Adresse einer Funktion hinterlegt werden, die zu einem Zeiger in ptrval1 und ptrval2 einen Text liefert. Dieser Text wird zur Darstellung von Debuginformationen zu Paintareas verwendet. Weitere Hinweise finden sich im Abschnitt "Ausgabe von Informationen zu Debugzwecken" auf Seite 228.

void (*S_pa_translate_token)(paintarea *pa, char *transTxt): r/w

In dieser Variablen kann die Adresse einer Funktion hinterlegt werden, die zur Übersetzung von Texten in Paintareas verwendet wird. Weitere Hinweise finden sich im Abschnitt "Erzeugen von Paintareas" auf Seite 220.

int S_save_screen_row: r/w

Diese Variable dient zur Steuerung des aktuellen Satzes beim Wechsel einer Variante. Wenn die Variable S_save_screen_row den Wert 1 besitzt, dann wird beim Verlassen einer Variante vom Typ Tabelle, der erste dargestellte

Satz in der Maskendatenstruktur vermerkt. Liegt der aktuelle Satz beim Zurückschalten auf diese Variante zwischen dem gemerkten Satz und der Anzahl sichtbarer Sätze, dann wird der gemerkte Satz wieder zum ersten sichtbaren Satz. Liegt der aktuelle Satz, der sich in einer anderen Variante geändert haben kann, nicht mehr in diesem Bereich, dann macht *FIX* den aktuellen Satz mittels der alten Methode sichtbar. D.h. die Tabelle wird so lange nach oben geschoben, bis der Satz sichtbar wird. Wenn die Variable S_save_screen_row den Wert 0 besitzt (das ist der Default), dann wird dieses Verfahren immer angewendet.

int S_skip_sync_refresh: r/w

Diese Variable (Vorbelegung: 0) erlaubt es - in Verbindung mit der Ressource SkipSyncRefresh (vgl. Seite 448) -, die Aktualisierung des Bildschirms zu steuern. Mehr hierzu in “Konfigurierung” auf Seite 274.

int S_this_century: r/w

legt den “aktuellen 100-Jahre-Zeitraum” für die Konversion zweistelliger Jahresangaben fest. Die Variable darf nur Werte 0 ≤ S_this_century < 99 annehmen, Vorbelegung ist 69. Erfolgt die Prüfung eines Datumliterals oder dessen Konversion in einen Datums*wert* mittels *FIX*-Routinen (→ atodate()), so wird die Angabe *yy* als 19*yy* interpretiert, wenn *yy* ≥ S_this_century, sonst als 20*yy*.[1]

Der Entwickler kann mit dieser Variable die Interpretation zweistelliger Jahresangaben steuern. Eine Änderung des Wertes sollte ggf. vor dem Aufruf von fxinit() erfolgen. In der Anwendung sollten dann keine Literale mit zweistelliger Jahreskomponente an das Datenbanksystem oder andere Tools übergeben oder von dort übernommen werden!

char S_weekdays[7][4]: readonly

Diese Variable enthält die Abkürzungen für die Wochentage (Mo - So). Diese sind der Datei `$FXDIR/runtime/messages` (bzw. `$FXRUNTIMEDIR/messages`) entnommen.

int S_yes: r/w
int S_no: r/w

Die Variable S_yes enthält den Code des Zeichens, das *FIX* zur internen Repräsentation des Wahrheitswertes *Ja* verwenden soll. Bei dem Wert muss es sich um einen Buchstaben oder eine Ziffer aus dem Zeichensatz ISO 646:1083 handeln. Die Variable ist mit ‘j’ vorbelegt.

Die Variable S_no enthält den Code des Zeichens, das *FIX* zur (internen) Repräsentation des Wahrheitswertes *Nein* verwenden soll. Bei dem Wert muss es sich um einen Buchstaben oder eine Ziffer aus dem Zeichensatz ISO 646:1083 handeln, der - unabhängig von Groß-/Kleinschreibung - von dem Wert von S_yes verschieden ist. Die Variable ist mit ‘n’ vorbelegt.

Unabhängig von den für *Ja* und *Nein* hinterlegten Zeichencodes gilt: NULL < *Nein* < *Ja*.

Der Wert beider Variablen darf nach dem ersten Aufruf einer *FIX*-Funktion nicht mehr geändert werden.

Ist die Umgebungsvariable FXTRUTH nicht gesetzt (oder enthält sie einen nicht korrekten Wert), so werden die Zeichen S_yes und S_no auch zur Darstellung verwendet.

char *shellname: r/w

zeigt auf den Wert für SHELL, bzw. wenn nicht definiert, für shell. Ist keine der beiden Variablen definiert, hinterlegt *FIX* einen Zeiger auf die Stringkonstante “/bin/sh”.

1. Dies gilt nicht für die Funktionen fxrstrdate() und fxrdefmtdate(), die stattdessen die Umgebungsvariable FX_RDTCV_YEAR_THRESHOLD auswerten.

2 Ressourcen

2.1 Ressourcen für alle *FIX*-Programme

Die in den folgenden Abschnitten beschriebenen Ressourcen werden von allen *FIX*-Programmen ausgewertet.

AllowToggleFieldInsertMode

Werte

TRUE
FALSE

Beschreibung

Wird diese Ressource auf FALSE gesetzt, kann der Bedienungsmodus bei der Erfassung von FXCHARTYPE-Feldern(vgl. Seite 34) nicht mittels der Taste CI gewechselt werden.

EnclosedHeadlineEnabled

Werte

TRUE
FALSE

Beschreibung

Bei TRUE wird beim Anzeigen der Überschrift eines Objekt-Windows nur dann ein Leerzeichen voran- bzw. nachgestellt, wenn fxisprint() für ihr erstes bzw. letztes Zeichen TRUE liefert.

IoopOpt

Werte

TRUE
FALSE

Beschreibung

Wird die Ressource auf TRUE gesetzt, wird für die Kommunikation mit einem grafischen Frontend ein modifiziertes Protokoll verwendet, das für *FIX/Win* optimiert ist.

LoadBufferSize

Werte

Integerwerte > 65500

Beschreibung

Größe des Arbeitsbereichs zum Laden von Objekten.
Wird die Ressource nicht angegeben oder unterschreitet ihr Wert den Defaultwert (65500), werden 65500 Bytes bereitgestellt (was je nach Plattform für Masken mit etwa 200 Elementen ausreicht). Der Wert sollte nur erhöht werden, wenn mehr als 65500 / sizeof(field) Elemente unterstützt werden müssen, da eine Vergrößerung des Arbeitsbereichs automatisch ein proportionales Anwachsen einiger anderer Library-interner Strukturen bewirkt.

MsgWindowXPos, MsgWindowYPosOffset

Werte

MsgWindowXPos: 0-Bildschirmbreite
MsgWindowYPosOffset: Integerwert

Beschreibung

Diese Ressourcen definieren die Position des Meldungsfensters. MsgWindowXPos legt die Spalte fest. Überstehende Meldungen werden abgeschnitten. MsgWindowYPosOffset wird auf den Standardwert addiert. Negative Werte verschieben die Position nach oben, positive nach unten. Wird das Meldungsfenster außerhalb des Bildschirms platziert, so wird eine Meldung nicht mehr angezeigt. Dieses Verhalten ist problematisch, vor allem bei Meldungen die eine Eingabe fordern (z.B. jn_msg()). Deshalb sollte dieser Zustand vermieden werden.

OraDateQualifier

Werte

Integerwerte

Beschreibung

DATETIME-Qualifier; in Verbindung mit ESQL/C ohne Belang

OraRowidSize

Werte

≥ 18 (18)

Beschreibung

Länge der ROWID für Oracle; in Verbindung mit ESQL/C ohne Belang

SkipSyncRefresh

Werte

TRUE
FALSE

Beschreibung

Wird die Ressource auf TRUE gesetzt, unterstützt *FIX* das Abschalten von “SYNC-Refreshs”. Mehr hierzu in “Konfigurierung” auf Seite 274.

SwapOptions

Werte

1,2,3

Beschreibung

Wird das Bit 1 in der Ressource SwapOptions gesetzt, dann erlaubt *FIX* den Austausch von Formaten eines Feldes. Das Format wird beim Laden einer Variante nicht mehr auf Gleichheit geprüft. Austausch des Formates bedeutet:

- Beim Wechsel in eine andere Variante (Zeilen- oder Maskenvariante) wird das Format aus der neuen Variante (fmtstring, len, fmt, displen) übernommen.
- Es ist möglich, das Format mit Hilfe von f_AssignFormat(), f_RemoveFormat() und f_RestoreFormat() programmatisch zu verändern.

Beim Austausch eines Formates wird geprüft, ob das neue Format keine größere Feldlänge bedingt. Das in der mfo-Datei zur Variante 0 hinterlegte Format oder die dort hinterlegte Feldlänge definiert die maximal mögliche Länge für ein Feld.

Wird das Bit 2 in der Ressource SwapOptions gesetzt, dann kopiert *FIX* die anwendungsspezifischen Feld- und Maskeninformationen beim Wechsel von Varianten. Dies ist notwendig, wenn diese Informationen benötigt werden, um daraus Formatänderungen herzuleiten die in beiden Varianten unterschiedlich sind.

Die anwendungsspezifischen Informationen in ob_props->longval1 bis ob_props->longval8 und ob_props->ptrval1 bis ob_props->ptrval2 werden von *FIX* nur dann kopiert, wenn an der Adresse S_SwapPtrval keine Funktion hinterlegt ist. Andernfalls bleibt das Kopieren der Werte dieser Funktion überlassen.

UseFieldInsertMode

Werte

TRUE
FALSE

Beschreibung

Mit dieser Ressource kann der Bedienungsmodus für die Erfassung von FXCHARTYPE-Feldern (vgl. Seite 34) vorgegeben werden. Wird die Ressource auf TRUE gesetzt, wird anstelle des Overwrite-Modus der Insert-Modus verwendet.

Wie der Insert-Modus im Programm abgeschaltet werden kann, ist auf Seite 444 beschrieben.

fix_handled_signals

Werte

Durch Komma getrennte Liste von Integerwerten

Beschreibung

Über diese Ressource kann bestimmt werden, welche Signale von *FIX* behandelt werden. Die Ressource muss eine durch Komma getrennte Liste mit Signalnummern enthalten. Wird die Ressource nicht gesetzt, dann werden alle Signale von *FIX* behandelt.

selo_bottom_distance

Werte

Integerwerte von 2-6

Beschreibung

Distanz eines Selos oder eines Hilfetextes zum unteren Bildrand. Wenn die Ressource nicht angegeben wird, oder wenn ein falscher Wert angegeben wird, dann wird mit einem Wert von 4 gerechnet.

2.2 Ressourcen für bestimmte FIX-Programme

led und rled

Zu den Ressourcen des Layout-Editors siehe Seite 119.

fgmenu

SourceStyle

Werte

V3.1.0
V2.9.4
V2.9.3

Beschreibung

ohne Wirkung

TargetFileExtensions

Werte

fixed
variable

Beschreibung

plattformspezifische Dateiendungen; Makros O, A und E als Endung

fgmask

SourceStyle

Werte

V3.1.0
V2.9.4
V2.9.3

Beschreibung

Bestimmt den Stil der generierten Sourcen - V.2.9.3 meint Sourcen ähnlich wie Version 2.9.3, V.2.9.4 meint Sourcen ähnlich wie Version 2.9.4 bis 3.0.0.

BytesPerCharMethod

Werte

constant
macro

Beschreibung

Bestimmt, ob bei der Source-Generierung im Stil V3.1.0 die Größe von char- und fixchar-Hostvariablen mit einer numerischen Konstante oder dem Makro BPC vereinbart wird.

SQLDialect

Werte

INFORMIX
other

Beschreibung

z.Z. ohne Belang

SQLFileExtension

Werte

.ec
…

Beschreibung

Dateiendung für Embedded-SQL-Sourcen

SQLPrefix

Werte

EXEC SQL
exec sql
$

SQLIndicators

Werte

none - keine Indikatorvariablen
separate - eigene Datenstruktur für Indikatorvariablen
embedded - Indikatorvariablen in Datenbank-Hostvariablensatz integriert

Beschreibung

Bestimmt die Art der Indikatorvariablen

SQLIndSymbol

Werte

indicator
INDICATOR
Leerstring

Beschreibung

Trennsymbol zwischen Host- und Indikatorvariable

SQLIndSuffix

Werte

ind
kurze alphanumerische Zeichenkette

Beschreibung

Suffix zur Bildung des Indikatornamens

TargetFileExtensions

Werte

fixed
variable

Beschreibung

plattformspezifische Dateiendungen; Makros O, A und E als Endung

45 Generierung von Objektbeschreibungen

1 Generierung von Masken

FIX kann zu Datenbanktabellen Maskenbeschreibungsdateien erzeugen. **fxm** extrahiert mittels des Aufrufs

$FXDIR/cmd/do/ex-awmfo <dbtable> <mfopath> (-b | -t)

Information über die Spalten der Datenbanktabelle <dbtable> und generiert eine Maskenbeschreibungsdatei. <mfopath> muss ein Pfad relativ zum Anwendungsverzeichnis $FXHOME sein, dessen Dateianteil keine Großbuchstaben enthalten darf.

Die generierte Beschreibung wird als `$FXHOME/<mfopath>` abgelegt. Der Pfad für die Layoutdatei wird so bestimmt, dass dem Basisnamen, der sich ergibt, wenn man von <mfopath> den Verzeichnisanteil und ggf. die Endung .mfo abschneidet, bei Angabe des Schalters -b der Pfad `$FXHOME/mly/` vorangestellt und die Endung .mly angehängt bzw. bei Angabe des Schalters -t der Pfad `$FXHOME/pan/` vorangestellt und die Endung .pan angehängt wird. Objekt- und Layoutdatei dürfen noch nicht existieren.

Zu jeder Spalte der Datenbanktabelle wird ein Feld generiert. Das i-te Feld wird an der Position (i,25) der Maske, die Maske selbst an die Position (0,0) platziert. Der Feldtyp wird aus dem Spaltentyp abgeleitet, die Feldlänge entsprechend der folgenden Tabelle generiert:

Datentyp	Feldlänge
SQLCHAR	wie Spalte[1]
SQLNCHAR[2]	wie Spalte
SQLDATE	10
SQLSMINT	5
SQLINT	5
SQLSERIAL	5
SQLSMFLOAT	8 (Nachkommastellen nicht spezifiziert)
SQLFLOAT	8 (Nachkommastellen nicht spezifiziert)
SQLMONEY	wie Spalte
SQLDECIMAL	wie Spalte (auch Nachkommastellen)
SQLDTIME	wie Spalte (normkonform: Teilstring von “yyyy-mm-dd hh:nn:ss”)
SQLINTERVAL	wie Spalte (normkonform mit Vorzeichen)

Da die Größe einer Maske auf 22×80 beschränkt ist und die Maske standardmäßig mit einem Rahmen versehen wird, erhalten das 21-te und alle weiteren Felder die Eigenschaft NODISPL. (Sichtbare) Felder, die aufgrund ihrer Länge über die Spalte 78 hinausgehen, müssen später vom Entwickler manuell nachbehandelt werden. Als Titel wird “*MMM* Maske” verwendet. *MMM* steht für den Maskennamen, der dem Basisnamen der Datei, nach Großschrift gewandelt, entspricht.

Als Feldbeschriftung erscheint im ansonsten weißen Layout vor jedem Feld ab Spalte 1 “*MMM_FFF*”: *FFF* steht für den Spaltennamen in Großschrift. Eine Beschriftung, die mit den zur Verfügung stehenden Spalten 1 bis 23 nicht auskommt, wird abgeschnitten.

1. Hier wird von der Annahme ausgegangen, dass die Datenbank keine Multibyte-Kodierung verwendet.
2. SQLNCHAR wird von *FIX* als SQLCHAR behandelt.

46 Source-Generierung für Masken

FIX ist ein Applikations*generator*. Ausgehend von den Beschreibungsdateien von Objekten werden funktionsfähige C-Programme generiert, in denen der Programmierer anwendungsspezifische Ergänzungen vornehmen kann. Die Generierung erhebt hierbei nicht den Anspruch, eine gebrauchsfertige Anwendung zu liefern, sondern will den Software-Entwicklungsprozess vorrangig in zwei Schwerpunkten unterstützen:

- Sie stellt zum einen eine leistungsfähige Hilfe bei der Routinearbeit der Source-Erstellung dar. Die dabei anfallenden Fehler, z.B. die falsche Deklaration von Variablen, und der Pflegeaufwand bei Änderungen können durch eine leistungsfähige Generierung verringert werden.
- Zum anderen erreicht man durch den “Stil” des generierten Sourcecodes einen sehr flexiblen, die Innovationsmöglichkeiten des Einzelnen nicht hemmenden Rahmen, der bereits in der Vergangenheit vielen Anwendungen eine erprobte Grundstruktur gegeben hat.

Die mit Version 2.9.0 eingeführte Source-Generierung bietet als zusätzliches Feature die Nachgenerierung einzelner Programmteile wie Element-Bezeichner, Hostvariablen und mfn_bind-Arrays an. Dadurch ist es bei der nachträglichen Änderung von Masken möglich, die durch deren Form bedingten Teile des Sourcecodes neu zu erzeugen, ohne dass der vielleicht bereits vom Programmierer veränderte Ablaufteil davon berührt wird.

1 Struktur der Source

Die Generierung ist in der Lage, verschiedene durch den Entwickler wählbare Varianten der Generierung miteinander zu kombinieren.

Die erste Variationsmöglichkeit besteht in der Zahl der Hostvariablen-Sätze.

In der einfachen Variante ist nur ein Satz von Hostvariablen vorhanden, d.h. Datenbankschnittstelle und Maske benutzen die gleichen Variablen. Als Datenbank-C-Schnittstelle kommt in diesem Fall nur IBM Informix ESQL/C in Frage.

In der komplexeren Variante existieren drei Sätze von Hostvariablen: ein Satz *Feld-Hostvariablen* sowie ein Satz *Datenbank-Hostvariablen* und ein Satz von *Query-Hostvariablen*, in denen die Werte von Parametern für den Datenbankzugriff gehalten werden. Seit Version 2.9.4 ist zusätzlich die Generierung von *Indikatorvariablen* zu den beiden letztgenannten möglich (vgl. Seite 463). Der Werteaustausch zwischen den Variablensätzen erfolgt über eine ebenfalls generierte *Kopierfunktion*.

Die Unterscheidung zwischen Feld- und Datenbank-Hostvariablen erlaubt zum einen die unterschiedliche Repräsentation von Werten in Datenbank und Maske. So kann beispielsweise in der Datenbank in einer Spalte ein Zeitpunkt abgelegt sein, während in der Maske dessen Darstellung in zwei Feldern - Datum und Uhrzeit - erfolgt. Zum anderen kann bei der Trennung in drei Variablensätze der datenbankspezifische Teil auch in anderen Modulen Verwendung finden, z.B. können die datenbankspezifischen Teile eines Stammdaten-Moduls in ein Bewegungsdaten-Modul übernommen werden.

Der wesentlichste Aspekt der Trennung liegt jedoch in der größeren Abstraktion vom jeweiligen Datenhaltungssystem, was - insbesondere in Verbindung mit der unten beschriebenen Mehr-Dateien-Variante - die Portierbarkeit der Anwendung auf andere Datenhaltungssysteme ermöglicht.

Die zweite Variationsmöglichkeit betrifft die Anzahl der Dateien, aus denen das erzeugte Programm bestehen soll. Während in der einfachen Variante nur eine Source-Datei `source.ec` erzeugt wird, sind es in der komplexen Variante vier Dateien:

- eine Header-Datei *source*.h, die die Datenstrukturen für die (Feld-)Hostvariablen und die Vereinbarung der Element-Bezeichner enthält,
- eine Datei *source*_ms.c (bzw. .ec), die die Feld-Hostvariablen und mfn_bind-Arrays definiert,
- eine Datei *source*_mn.c (bzw. .ec), in der sich die Anwendungslogik befindet,
- eine Datei *source*_sq.ec, die die Schnittstelle zur Datenbank beinhaltet.

Die .h-Datei wird von den drei anderen mittels einer '#include'- oder, im Falle von Embedded-SQL-Sourcen, 'EXEC SQL include'-Anweisung eingeschlossen.

Die Ein-Dateien-Variante ist kompakter und in der Komplexität dem von früheren *FIX*-Versionen generierten Sourcecode vergleichbar.

Der Vorteil der Trennung in mehrere Dateien liegt darin, dass diejenigen Programmteile, die unmittelbar aus der Beschreibungsdatei abgeleitet sind, nachgeneriert werden können, ohne dass Datenbank- und Anwendungslogik davon berührt werden.

		Hostvariablen-Sätze	
		1	3
Anzahl Dateien	1	*source*.ec	*source*.ec
	4	*source*.h *source*_ms.ec *source*_sq.ec *source*_mn.ec	*source*.h *source*_ms.c *source*_sq.ec *source*_mn.c

Tabelle 2: Source-Dateien im Vergleich

Sourcecode-Dateien werden generell im Verzeichnis $FXHOME/c abgelegt.

1.1 Struktur der Source im einfachsten Fall (eine Datei, gemeinsam genutzte Variablen)

Die Source *source*.ec[1] enthält folgende Komponenten:

- die Definition der Konstanten FIXRELEASE, die anzeigt, mit welcher *FIX*-Version die Source erstellt wurde,
- Einschlüsse von C-Header-Dateien, Embedded-SQL-Header-Dateien und der *FIX*-Header-Datei fix/fix.h.
- einen deklarativen Teil, in dem u.a. die Hostvariablen (als C-struct innerhalb einer DECLARE SECTION) und die Element-Bezeichner (als Makros) vereinbart werden, sowie die mfn_bind-Arrays, mittels derer die Hostvariablen mit den Maskenfeldern verbunden werden,
- einen Datenbeschaffungsteil, der zum einen die direkte Schnittstelle zur Datenbank enthält: eine Funktion *maskenname*_Sql() pro beteiligter Maske. Zum anderen existiert in jedem Modul eine Funktion *maskenname*_IO(), die die Kommunikation zwischen Ablaufteil und _Sql-Funktion(en) steuert.
- pro Maske eine Anwendungslogik (*maskenname*_event_control()),
- pro Modul eine Modul-Prozedur (*maskenname*_proc()),

In der Source befindet sich keine main()-Funktion. Sie wird, gültig für alle Module, in einer separaten Datei main.c mitgeliefert und gemäß folgender Regel im Makefile mit eingebunden:

```
source:         $(BIN)/source
                @echo $@ done
```

1. Die Dateiendung kann durch die Ressource [fgmask].SQLFileExtension konfiguriert werden (vgl. Seite 451).

```
$(BIN)/source:      $(OBJS)/source.o
                    ( cd $(OBJS); \
                    $(CC) $(CFLAGS) \
                          -DPROC="maskenname_proc" [1] $(FXDIR)/src/main.c \
                          source.o $(LIBS) -o $@; \
                    rm -f main.o )

$(OBJS)/source.o:   source.ec
                    $(ESQLC) $(ESQLCFLAGS) source.ec [2]
                    ( cd $(OBJS); $(CC) -c $(CFLAGS) $(SRC)/source.c )
                    -rm source.c
```

Zum Aufbau des Makefile vgl. Seite 59. Die Dateiendungen von Objektdatei und Binärprogramm können mittels der Ressource [fgmask].TargetFileExtensions beeinflusst werden (vgl. Seite 452).

1.2 Struktur der Source im komplexesten Fall (vier Dateien, drei Variablensätze)

source.h:

source.h enthält die deklarativen Teile des Programms.

Hier wird die Datenstruktur für den Feld-Hostvariablensatz in Form einer Struktur vereinbart. Außerdem werden die Element-Bezeichner als Makros definiert.

Anmerkung:

Bei nur einem Hostvariablensatz ist die Vereinbarung in das Anweisungspaar

```
EXEC SQL BEGIN DECLARE SECTION ;
EXEC SQL END DECLARE SECTION ;
```

eingebettet.

*source*_ms.c:

Diese Datei definiert die Feld-Hostvariablen und die mfn_bind-Arrays.

*source*_sq.ec[3]:

Diese Datei beinhaltet die Schnittstelle zur Datenbank. In ihr werden der Datenbank-Hostvariablensatz und der Query-Hostvariablensatz, jeweils in Form einer Struktur, vereinbart (optional mit einer Indikatorvariablen zu jeder Komponente). Außerdem sind in ihr die _Sql-Funktionen und die Kopierfunktionen enthalten.

Wenn eine Trennung von Datenhaltung und Datenpräsentation gewünscht wird, gehört jeder Datenbankzugriff in diese Datei.

*source*_mn.c:

Hier finden sich der Ablaufteil des Programms, also die als Anwendungslogik benutzten Funktionen *maskenname*_event_control(), die Funktionen *maskenname*_ConsistencyControl() zur Konsistenzprüfung und die

1. Diese Vereinbarung bewirkt, dass als Modul-Prozedur *maskenname*_proc() aufgerufen wird (vgl. Seite 467).
2. Das Makro ESQLCFLAGS wurde erst bei *FIX* 3.1.0 eingeführt.
3. Die Dateiendung kann durch die Ressource [fgmask].SQLFileExtension konfiguriert werden (vgl. Seite 451).

Funktion *maskenname*_proc(), die das Anwendungsmodul darstellt. In dieser Datei werden die Eingriffe in den Programmablauf vorgenommen.

Auch in der Vier-Dateien-Variante wird die main()-Funktion aus `main.c` verwendet:

```
source:          $(BIN)/source ;
                 @echo $@ done

$(BIN)/source:   $(OBJS)/source_mn.o $(OBJS)/source_sq.o $(OBJS)/source_ms.o
                 ( cd $(OBJS); \
                 $(CC) $(CFLAGS) \
                        -DPROC="maskenname_proc" $(FXDIR)/src/main.c \
                        source_mn.o source_sq.o source_ms.o \
                        $(LIBS) -o $@; \
                 rm -f main.o )

$(OBJS)/source_sq.o:   source_sq.ec source.h
                       $(ESQLC) $(ESQLCFLAGS) source_sq.ec [1] [2]
                       ( cd $(OBJS); $(CC) -c $(CFLAGS) $(SRC)/source_sq.c )
                       -rm source_sq.c

$(OBJS)/source_ms.o:   source_ms.c source.h
                       ( cd $(OBJS); $(CC) -c $(CFLAGS) $(SRC)/source_ms.c )

$(OBJS)/source_mn.o:   source_mn.c source.h
                       ( cd $(OBJS); $(CC) -c $(CFLAGS) $(SRC)/source_mn.c )
```

Zum Aufbau des Makefile vgl. Seite 59. Die Dateiendungen von Objektdateien und Binärprogramm können mittels der Ressource [fgmask].TargetFileExtensions beeinflusst werden (vgl. Seite 452).

2 Elemente der Masken-Source

Im folgenden werden die einzelnen Teile der Source an Beispielen aus der mitausgelieferten Demo-Anwendung (Auftragsbearbeitung) erläutert. Sie enthält folgende Masken (hier vereinfacht dargestellt):

```
mask AUFTR …
22 77 1 1
field AUFTR_AUFNR          3 19 5      …   2        auftr.aufnr      …
field AUFTR_DATUM          3 39 10     …   514      auftr.datum      …
field AUFTR_BUDATUM        3 69 5      …   514      auftr.budatum    …
field AUFTR_BEARB          4 39 8      …   0        auftr.bearb      …
field AUFTR_VERTRETERNR 4 69 5         …   2:null   auftr.vertreternr …
field AUFTR_KUCODE         6 14 8      …   0        auftr.kucode     …
field AUFTR_KUNR           7 14 6      …   2        auftr.kunr       …
field AUFTR_AUFTEXT        7 24 76:51  …   0        auftr.auftext    …
```

1. Der IBM Informix ESQL/C-Präprozessor erlaubt es erst ab Version 6.x, einen Suchpfad für Header-Dateien anzugeben, und muss daher in dem Verzeichnis ablaufen, in dem die Quelltexte liegen.
2. Das Makro ESQLCFLAGS wurde erst bei *FIX* 3.1.0 eingeführt.

```
field AUFTR_SUMME       20 63 9.2   …   5     auftr.summe  …
table mfo/pos.mfo …
with aufnr=#AUFTR_AUFNR
END_OF_MFO

table POS…
11 77 10 1 1 1
field POS_AUFNR         0 0 5       …   2     pos.aufnr    \
                                              link #.AUFTR_AUFNR
field POS_ARTIKNR       3 4 7       …   2     pos.artiknr  …
field POS_BEZEI         3 16 12     …   0     pos.bezei    …
field POS_PREISEINZEL   3 36 8.2    …   5     pos.preiseinzel…
field POS_ANZ           3 51 5      …   1     pos.anz      …
field POS_PREISGES      3 64 8.     …   5     pos.preisges …
field POS_RAND1         3 2 1       …   0:null NULL        …
field POS_RAND2         3 73 1      …   0:null NULL        …
with aufnr=#AUFTR.AUFTR_AUFNR
END_OF_MFO
```

2.1 Der deklarative Teil

Der deklarative Teil, der am engsten vom Aufbau der Maske abhängig ist, befindet sich, wie oben erwähnt, entweder mit in der Datei *source*.ec oder, im Falle der Mehr-Dateien-Variante, in den Dateien *source*.h und *source*_ms.c.

2.1.1 Element-Bezeichner

Die Element-Bezeichner, die in den mfn_bind-Arrays und im Ablaufteil benutzt werden, sind als Makros realisiert. Die Makros sind gleich lautend mit dem Namen des Feldes oder der eingebetteten Maske. Um ihre Werte, positive Zahlen, möglichst über die gesamte Anwendung eindeutig zu machen und Verwechslungen mit Element-Positionen auszuschließen, beginnt die Vergabe mit der Nummer 1001 und wird bei jedem Element um 1 erhöht. Die Nummern von nachgelagerten Masken beginnen jeweils auf der nächsten Hunderter-Grenze erhöht um 1. Dabei werden bei jedem erneuten Aufruf der Generierung (weitere Module, Nachgenerierung von Modulen) die bereits verwendeten Nummern berücksichtigt, also nicht wieder verwendet.[1]

Beispiel:

```
/* Element-Bezeichner zur Maske AUFTR */
#define AUFTR_AUFNR          1001
#define AUFTR_DATUM          1002
#define AUFTR_BUDATUM        1003
#define AUFTR_VERTRETERNR    1004
#define AUFTR_BEARB          1005
#define AUFTR_KUCODE         1006
#define AUFTR_KUNR           1007
#define AUFTR_AUFTEXT        1008
#define AUFTR_SUMME          1009
#define POS                  1010      /* eingebettete Maske */
```

1. Die Generierung verwendet, wenn kein Startwert für die Element-Bezeichner beim Aufruf mitgegeben wird, den in der Datei c/MAX_PRG_ELEMNR hinterlegten Wert, der nach jedem Generierungslauf angepasst wird.

```
/* Element-Bezeichner zur eingebetteten Maske POS */
#define POS_AUFNR          1101
#define POS_ARTIKNR        1102
#define POS_BEZEI          1103
#define POS_PREISEINZEL    1104
#define POS_ANZ            1105
#define POS_PREISGES       1106
#define POS_RAND1          1107
#define POS_RAND2          1108
```

Bei der Vier-Dateien-Variante werden diese Makros in *source*.h vereinbart.

2.1.2 Feld-Hostvariablen

Zu den Masken wird jeweils eine Struktur (vom Typ struct m_*maskenname*_struct) mit Namen m_*maskenname* vereinbart, die zu jedem Feld, das in der Maskenbeschreibung mit einer Datenbankspalte assoziiert ist, eine Komponente enthält. Die Namen der Komponenten sind identisch mit denen der assoziierten Spalten (Spaltenname ohne Tabellenanteil), ihr Typ ist aus dem Feldtyp abgeleitet.

Beispiel:

```
/* Struktur des Hostvariablensatzes zur Maske AUFTR */
struct m_auftr_struct {
    long    aufnr;
    date_t  datum;
    date_t  budatum;
    long    vertreternr;
    char    bearb[8 + 1]; [1]
    char    kucode[8 + 1];
    long    kunr;
    char    auftext[76 + 1];
    dec_t   summe;
};

/* Struktur des Hostvariablensatzes zur eingebetteten Maske POS */
struct m_pos_struct {
    long    artiknr;
    char    bezei[12 + 1];
    dec_t   preiseinzel;
    short   anz;
    dec_t   preisges;
};

/* Hostvariablensatz zur Maske AUFTR */
struct m_auftr_struct m_auftr;

/* Hostvariablensatz zur eingebetteten Maske POS */
struct m_pos_struct   m_pos;
```

1. Diese Summenschreibweise - Anzahl Zeichen (× Faktor) + Platz für Nullbyte - wurde erst mit *FIX* 3.1.0 eingeführt. Vgl hierzu auch die Ressource BytesPerCharMethod auf Seite 450.

Bei der Mehr-Dateien-Variante werden die Datentypen in *source*.h vereinbart, die Hostvariablen selbst in *source*_ms.c.

Wird keine Trennung zwischen Feld- und Datenbank-Hostvariablen vorgenommen, so bildet der Feld-Hostvariablensatz auch gleichzeitig den Datenbank-Hostvariablensatz, d.h. die Datenbankzugriffe erfolgen ebenfalls über diese Struktur. Die Typen der Komponenten werden in diesem Falle wie im Beispiel auf Seite 462 vereinbart.

2.1.3 mfn_bind-Arrays

Die Zuordnung der Feld-Hostvariablen zu den jeweiligen Maskenfeldern erfolgt mit Hilfe von Arrays von mfn_bind-Strukturen. Die Felder werden immer mit dem Feld-Hostvariablensatz, d.h. den Komponenten der Struktur m_*maskenname*, verbunden. Wird keine Trennung zwischen Datenbank- und Feld-Hostvariablen vorgenommen, impliziert dies eine gleichzeitige Verbindung mit den Datenbank-Hostvariablen.

Die mfn_bind-Struktur enthält neben dem Namen des Elements, dem Zeiger auf die Hostvariable und dem Element-Bezeichner auch Größe und Typ der Variablen in kodierter Form. So kann bei Ausführung des Programms festgestellt werden, ob Typ und Größe, wie sie in der Beschreibungsdatei festgelegt sind, zu der Variablen passen.[1]

Bei Feld-Links findet sich in der mfn_bind-Struktur anstelle eines Zeigers auf die Variable ein Nullzeiger, da sich ein Feld-Link die Hostvariable mit seinem Root-Feld teilt. In einer eventuell vorhandenen Kopierfunktionen wird angenommen, dass bei einem Link auf das Feld *feld* der Maske *maske* dessen Feld-Hostvariable m_*maske*.*feld* heißt und zu demselben Modul gehört.

Für Felder, mit denen keine Datenbankspalte oder Variable in der Maskenbeschreibung assoziiert ist, wird kein Element im mfn_bind-Array erzeugt.

Beispiel:

```
/* mfn_bind-Array zur Maske AUFTR */
static mfn_bind AUFTR_bind[ ] = {
    { "AUFTR_AUFNR", (char *) &m_auftr.aufnr, AUFTR_AUFNR,
                    FXMFTYPE(sizeof(m_auftr.aufnr), FXCLONG) },
    { "AUFTR_DATUM", (char *) &m_auftr.datum, AUFTR_DATUM,
                    FXMFTYPE(sizeof(m_auftr.datum), FXCDATE) },
    { "AUFTR_BUDATUM", (char *) *m_auftr.budatum, AUFTR_BUDATUM,
                    FXMFTYPE(sizeof(m_auftr.budatum), FXCDATE) },
    { "AUFTR_VERTRETERNR", (char *) &m_auftr.vertreternr, AUFTR_VERTRETERNR,
                    FXMFTYPE(sizeof(m_auftr.vertreternr), FXCLONG) },
    { "AUFTR_BEARB", m_auftr.bearb, AUFTR_BEARB,
                    FXMFTYPE(sizeof( m_auftr.bearb), FXCCHAR) },
    { "AUFTR_KUCODE", m_auftr.kucode, AUFTR_KUCODE,
                     FXMFTYPE(sizeof(m_auftr.kucode), FXCCHAR) },
    { "AUFTR_KUNR", (char *) &m_auftr.kunr, AUFTR_KUNR,
                    FXMFTYPE(sizeof(m_auftr.kunr), FXCLONG) },
    { "AUFTR_AUFTEXT", m_auftr.auftext, AUFTR_AUFTEXT,
                    FXMFTYPE(sizeof(m_auftr.auftext), FXCCHAR) },
    { "AUFTR_SUMME", (char *) &m_auftr.summe, AUFTR_SUMME,
                    FXMFTYPE(sizeof(m_auftr.summe), FXCDECIMAL) },
    { "POS", (char *) 0, POS
                    /* Kodierung bei eingebetteter Maske nie ausgewertet */ },
    { (char *)0, (char *)0 }
};
```

1. Die Plausibilitätsprüfung hinsichtlich des Datentyp funktioniert nur, wenn bei einer Änderung des Datentyps der Variablen auch die Kodierung in der mfn_bind-Struktur angepasst wird (die Kodierung sollte sich selbst erklären).

```
/* mfn_bind-Array zur eingebetteten Maske POS */
static mfn_bind POS_bind[ ] = {
    { "POS_AUFNR", (char *)0
            /* Kodierung bei Feld-Link nie ausgewertet */ },
    { "POS_ARTIKNR", (char *) &m_pos.artiknr, POS_ARTIKNR,
            FXMFTYPE(sizeof(m_pos.artiknr), FXCLONG) },
    { "POS_BEZEI", m_pos.bezei, POS_BEZEI,
            FXMFTYPE(sizeof(m_pos.bezei), FXCCHAR) },
    { "POS_PREISEINZEL", (char *) &m_pos.preiseinzel, POS_PREISEINZEL,
            FXFXMFTYPE(sizeof(m_pos.preiseinzel), FXCDECIMAL) },
    { "POS_ANZ", (char *) &m_pos.anz, POS_ANZ,
            FXMFTYPE(sizeof(m_pos.anz), FXCSHORT) },
    { "POS_PREISGES", (char *) &m_pos.preisges, POS_PREISGES,
            FXMFTYPE(sizeof(m_pos.preisges), FXCDECIMAL) },
    { "POS_RAND1", (char *) 0, POS_RAND1
            /* Kodierung bei Feld ohne Bindung nie ausgewertet */ },
    { "POS_RAND2", (char *) 0, POS_RAND2
            /*Kodierung bei Feld ohne Bindung nie ausgewertet */ },
    { (char *)0, (char *)0 }
};
```

Bei der Mehr-Dateien-Variante werden die mfn_bind-Arrays in *source*_ms.c vereinbart.

2.1.4 Datenbank-Hostvariablen

Im Falle separater Datenbank-Hostvariablen erfolgt deren Vereinbarung ähnlich der der Feld-Hostvariablen in einer Struktur (vom Typ struct d_*tabelle*_struct) mit Namen d_*tabelle*. Hierbei steht *tabelle* für den Tabellenanteil der Datenbankspalte, die mit dem Fetch-Element der Maske assoziiert ist. Die Namen der Komponenten sind identisch mit denen der assoziierten Spalten (Spaltenname ohne Tabellenanteil), ihr Typ ist aus dem Feldtyp abgeleitet.

Zusätzlich wird pro Maske eine Struktur (vom Typ q_*maskenname*_struct) mit Namen q_*maskenname* generiert, die die Parameter für den Datenbankzugriff enthält (where-Bedingung der select-Anweisung).

Optional kann zu jedem Strukturelement eine entsprechende Indikatorvariable hinzukommen.

Beispiel:

```
/* Struktur des Datenbank-Hostvariablensatzes zur Maske AUFTR */
EXEC SQL BEGIN DECLARE SECTION ;
struct d_auftr_struct {
    long        aufnr;
    date[1]     datum;
    date        budatum,
    long        vertreternr,
    char        bearb[8 + 1];
    char        kucode[8 + 1];
    long        kunr;
    char        auftext[76 + 1];
    decimal     summe;
};
EXEC SQL END DECLARE SECTION ;
```

1. Die IBM Informix ESQL/C spezifischen Typbezeichnungen ‘fixchar[…]’, ‘date’, ‘decimal’, ‘datetime … to …’ und ‘interval … to …’ werden vom Präprozessor in C-Typen umgesetzt.

```
/* Struktur des Datenbank-Hostvariablensatzes zur eingebetteten Maske POS */
EXEC SQL BEGIN DECLARE SECTION ;
struct d_pos_struct {
    long        aufnr;
    long        artiknr;
    char        bezei[12 + 1];
    decimal     preiseinzel;
    short       anz;
    decimal     preisges;
};
EXEC SQL END DECLARE SECTION ;

/* Struktur des Query-Hostvariablensatzes zur Maske AUFTR */
EXEC SQL BEGIN DECLARE SECTION ;
struct q_auftr_struct {
    long        aufnr;
};
EXEC SQL END DECLARE SECTION ;

/* Struktur des Query-Hostvariablensatzes zur eingebetteten Maske POS */
EXEC SQL BEGIN DECLARE SECTION ;
struct q_pos_struct {
    long        aufnr;
};
EXEC SQL END DECLARE SECTION ;

/* Datenbank- und Query-Hostvariablensatz zur Maske AUFTR */
EXEC SQL BEGIN DECLARE SECTION ;
static struct d_auftr_struct    d_auftr;
static struct q_auftr_struct    q_auftr;
EXEC SQL END DECLARE SECTION ;

/* Datenbank- und Query-Hostvariablensatz zur eingebetteten Maske POS */
EXEC SQL BEGIN DECLARE SECTION ;
static struct d_pos_struct      d_pos;
static struct q_pos_struct      q_pos;
EXEC SQL END DECLARE SECTION ;
```

Bei der Mehr-Dateien-Variante erfolgen alle Vereinbarungen in *source*_sq.ec.

Indikatorvariablen

Die Methode von IBM Informix ESQL/C, den Wert NULL in die Hostvariable zu kodieren, ist bei anderen Datenbank-Entwicklungssystemen nicht üblich. Gewöhnlich wird hier ein Variablenpaar benötigt:

- die eigentliche Hostvariable, die entweder einen von NULL verschiedenen Wert besitzt oder deren Inhalt undefiniert ist, und
- eine ganzzahlige Indikatorvariable, die den Zustand der Hostvariable angibt:
  ≥ 0 bedeutet, dass der Inhalt der Hostvariablen gültig ist und den Wert beschreibt[1], < 0, dass der Wert NULL ist.
  Für Indikatorvariablen wird gewöhnlich der C-Datentyp short verwendet.

1. > 0 kennzeichnet i.d.R. einen Verlust an Genauigkeit.

Wo bei IBM Informix ESQL/C-Anweisungen allein die Hostvariable ausreicht

select … into …, :var_i, … from … where …, :par_j, …
insert into … values (…, :var_i, …)

etc., müssen dann jeweils zwei Variablen angegeben werden, sofern an der entsprechenden Position der Wert NULL beim Lesen auftreten kann bzw. beim Schreiben auftreten können soll:

…, :var_i indicator :ind_i, …

Das reservierte Wort “indicator” ist optional und dient zur leichteren Lesbarkeit.

FIX unterstützt zwei Methoden, Indikatorvariablen zu definieren:

- innerhalb der Datenbank- bzw. Query-Hostvariablen-Struktur, z.B.

```
EXEC SQL BEGIN DECLARE SECTION ;
struct d_auftr_struct {
    long        aufnr;
    short       aufnrind;
    date        datum;
    short       datumind;
    …
    };
EXEC SQL END DECLARE SECTION ;
```

- als zusätzliche Strukturen d_*tabelle*ind_struct und q_*maskenname*ind_struct neben den Hostvariablen-Sätzen, z.B.

```
EXEC SQL BEGIN DECLARE SECTION ;
…
struct d_auftrind_struct {
    short       aufnr;
    short       datum;
    …
    };
EXEC SQL END DECLARE SECTION ;
…
EXEC SQL BEGIN DECLARE SECTION ;
…
static struct d_auftrind_struct     d_auftrind;
EXEC SQL END DECLARE SECTION ;
```

Die erste Methode greift, wenn die Ressource [fgmask].SQLIndicators als Wert “embedded” besitzt, die zweite beim Wert “separate”. Voraussetzung für Indikatorvariablen sind getrennte Variablensätze für Felder und Datenbankschnittstelle.

Ob und welches Symbol zwischen Host- und Indikatorvariable generiert werden soll, kann ebenfalls durch eine Ressource gesteuert werden:

[fgmask.]SQLIndicatorSymbol : *Wert*

Der Suffix für die Namen der zusätzlichen Komponenten bzw. Strukturen ist mit “ind” vorbelegt, kann aber durch

[fgmask.]SQLIndicatorSuffix : *kurze alphanumerische Zeichenfolge*

frei konfiguriert werden.

2.1.5 Die Kopierfunktion *maskenname*_Copy()

Bei getrennten Datenbank- und Feld-Hostvariablen übernimmt die Kopierfunktion *maskenname*_Copy(), die in diesem Fall zu jeder Maske generiert wird, den dann erforderlichen Transport der Werte zwischen den Variablensätzen.

Je nach Modus, in dem die Funktion aufgerufen wird, überträgt sie die Daten aus den Datenbank- in die Feld-Hostvariablen (Modus DB_TO_MASK) bzw. umgekehrt (Modus MASK_TO_DB). Im Modus SELECT_PARAMS werden die Werte der als Parameter für den Lesezugriff benötigten Felder aus deren Feld-Hostvariablen in die Struktur q_*maskenname* kopiert.

Der Ablauf sieht wie folgt aus:

Vor dem Lesen werden die Werte der Felder *feld*, die an dem in der Maskenbeschreibung hinterlegten Schlüssel (with- und where-Angabe) beteiligt sind, in die Struktur q_*maskenname* kopiert. Dann erfolgt ein Zugriff auf die Datenbank nach dem Schema "select … where … = q_*maskenname*.*feld* …; fetch into d_*tabelle*". Wird ein Satz gefunden, werden die Werte in d_*tabelle* mit der Kopierfunktion nach m_*maskenname* übertragen.

Beim Schreiben (insert, update etc.) werden umgekehrt die Daten in der Struktur m_*maskenname* nach d_*tabelle* kopiert und von dort in die Datenbank geschrieben.

Beispiel:

```
/* _Copy-Funktion für die Hostvariablensätze zur Maske AUFTR */
static void AUFTR_Copy(int op) [1]
{
    switch (op) {
    case MASK_TO_DB:
        /* Feld-Hostvariablen -> Datenbank-Hostvariablen */
        d_auftr.aufnr := m_auftr.aufnr;
        d_auftr.datum := m_auftr.datum;
        d_auftr.budatum := m_auftr.budatum;
        d_auftr.vertreternr := m_auftr.vertreternr;
        d_auftr.bearb := m_auftr.bearb;
        d_auftr.kucode := m_auftr.kucode;
        d_auftr.kunr := m_auftr.kunr;
        d_auftr.auftext := m_auftr.auftext;
        d_auftr.summe := m_auftr.summe;
        break;

    case DB_TO_MASK:
        /* Datenbank-Hostvariablen -> Feld-Hostvariablen */
        m_auftr.aufnr := d_auftr.aufnr;
        m_auftr.datum := d_auftr.datum;
        m_auftr.budatum := d_auftr.budatum;
        m_auftr.vertreternr := d_auftr.vertreternr;
        m_auftr.bearb := d_auftr.bearb;
        m_auftr.kucode := d_auftr.kucode;
        m_auftr.kunr := d_auftr.kunr;
        m_auftr.auftext := d_auftr.auftext;
        m_auftr.summe := d_auftr.summe;
        break;

    case SELECT_PARAMS:
        /* Feld-Hostvariablen -> Parameter-Hostvariablen */
```

1. Tatsächlich werden alle Funktionsköpfe in der Form

   ```
   #ifdef __STDC__
   ANSI-C Notation
   #else
   K&R-C Notation
   #endif  /* __STDC__ */
   ```

   generiert.

```
            q_auftr.aufnr := m_auftr.aufnr;
            break;
        }
    }

    /*_Copy-Funktion für die Hostvariablensätze zur eingebetteten Maske POS */
    static void POS_Copy(int op)
    {
        switch (op) {
        case MASK_TO_DB:
            /* Feld-Hostvariablen -> Datenbank-Hostvariablen */
            d_pos.aufnr := m_auftr.aufnr;
            d_pos.artiknr := m_pos.artiknr;
            d_pos.bezei := m_pos.bezei;
            d_pos.preiseinzel := m_pos.preiseinzel;
            d_pos.anz := m_pos.anz;
            d_pos.preisges := m_pos.preisges;
            break;

        case DB_TO_MASK:
            /* Datenbank-Hostvariablen -> Feld-Hostvariablen */
            m_pos.artiknr := d_pos.artiknr;
            m_pos.bezei := d_pos.bezei;
            m_pos.preiseinzel := d_pos.preiseinzel;
            m_pos.anz := d_pos.anz;
            m_pos.preisges := d_pos.preisges;
            break;

        case SELECT_PARAMS:
            /* Feld-Hostvariablen -> Parameter-Hostvariablen */
            q_pos.aufnr := m_auftr.aufnr;
            break;
        }
    }
```

Im einfachsten Fall erfolgen die obigen “Zuweisungen” durch in `fix/fix.h` definierte Makros der Form FX*type*_TO_DB*type*(d,m) bzw. DB*type*_TO_FX*type*(m,d), die die Variableninhalte einfach kopieren. Sie können aber auch durch Funktionsaufrufe ersetzt werden, die beispielsweise eine Typumwandlung vornehmen.

Bei der Mehr-Dateien-Variante werden diese Funktionen in *`source`*`_sq.ec` vereinbart.

Indikatorvariablen

Bei Sourcen mit Indikatorvariablen generiert *FIX* in die Source mit der Datenbankschnittstelle zusätzlich die C-Präprozessor-Anweisung

```
#include <fix/fxdbval.h>
```

und generiert - anstelle der Makros aus `fix/fix.h` - Aufrufe der in `fxdbval.h` bekanntgemachten Funktionen

- zum Setzen von Datenbank-Hostvariable und Indikatorvariable gemäß dem Wert der Feld-Hostvariablen:

```
int fxLoadChar(char *fxvar, int fxlen, char *dbvar, int dblen, short *dbind)
int fxLoadGraphics(char *fxvar, int fxlen, char *dbvar, int dblen, short *dbind)
int fxLoadTruth(char *fxvar, char *dbvar, short *dbind)
int fxLoadShort(short *fxvar, short *dbvar, short *dbind)
```

int fxLoadLong(long *fxvar*, long *dbvar*, short *dbind*)
int fxLoadFloat(float *fxvar*, float *dbvar*, short *dbind*)
int fxLoadDouble(double *fxvar*, double *dbvar*, short *dbind*)
int fxLoadDec(dec_t *fxvar*, dec_t *dbvar*, short *dbind*)
int fxLoadDate(date_t *fxvar*, long *dbvar*, short *dbind*)
int fxLoadDtime(dtime_t *fxvar*, dtime_t *dbvar*, short *dbind*)
int fxLoadIntrvl(intrvl_t *fxvar*, intrvl_t *dbvar*, short *dbind*)

- zum Setzen des Wertes der Feld-Hostvariablen gemäß den Werten von Datenbank-Hostvariable und Indikatorvariable:

int fxStoreChar(char *fxvar*, int *fxlen*, char *dbvar*, int *dblen*, short *dbind*)
int fxStoreGraphics(char *fxvar*, int *fxlen*, char *dbvar*, int *dblen*, short *dbind*)
int fxStoreTruth(char *fxvar*, char *dbvar*, short *dbind*)
int fxStoreShort(short *fxvar*, short *dbvar*, short *dbind*)
int fxStoreLong(long *fxvar*, long *dbvar*, short *dbind*)
int fxStoreFloat(float *fxvar*, float *dbvar*, short *dbind*)
int fxStoreDouble(double *fxvar*, double *dbvar*, short *dbind*)
int fxStoreDec(dec_t *fxvar*, dec_t *dbvar*, short *dbind*)
int fxStoreDate(date_t *fxvar*, long *dbvar*, short *dbind*)
int fxStoreDtime(dtime_t *fxvar*, dtime_t *dbvar*, short *dbind*)
int fxStoreIntrvl(intrvl_t *fxvar*, intrvl_t *dbvar*, short *dbind*)

2.2 Die Funktion main()

In der von *FIX* generierten Source selbst befindet sich keine Funktion main(). Stattdessen wird eine separate Datei `main.c` mit einer solchen Funktion mitgeliefert, die für alle generierten Module verwendet werden kann.

In der darin enthaltenen Funktion main() findet die Initialisierung von *FIX* sowie das Öffnen der Datenbank statt, des Weiteren der Aufruf der Modul-Prozedur und eine kontrollierte Beendigung des Programms. Der Name der Modul-Prozedur, i. Allg. *maskenname*_PROC(), wird im Makro PROC erwartet, das bei der Übersetzung so definiert werden muss, dass es zum Namen der jeweiligen Modul-Prozedur expandiert wird (vgl. Makefile-Eintrag auf Seite 456).[1]

```
int  PROC(void);

int main (int argc, char *argv[])
{
    long ret;

    arginspect(&argc, argv);
    fxinit(0, (char **)0);
    if ((ret = fx_database_connect((char *)0)) != 0) {
        fxquit();
        exit(ret);
    }
    (void)PROC();
    ret = fx_database_disconnect();
    fxquit();
```

1. Diese Verfahrensweise erlaubt keine Verwendung von Parametern in der Modul-Prozedur. Hier muss ein Eingriff von Hand erfolgen.

```
    exit(ret);
}
```

2.3 Die Modul-Prozedur *maskenname*_proc()

Die Funktion *maskenname*_proc() stellt das eigentliche Modul dar. Sie befindet sich in der Datei *source*.ec bzw. *source*_mn.c.

Beispiel:

```
int AUFTR_proc(void)
{
    static int loaded = FALSE;
    int rc;

    if (loaded == FALSE) {
        /* Masken laden */
        AUFTR_mskp = loadmask("mfo/auftr.mfo");
        /* Felder der Maske AUFTR an Feld-Hostvariablen binden */
        mfn_varbind(AUFTR_bind, (mask *)AUFTR_mskp);
        /* der Maske AUFTR Anwendungslogik zuordnen */
        o_install_ev_control((obj *)AUFTR_mskp, AUFTR_event_control);
        POS_mskp = (table *)fx_mfo_f(AUFTR_mskp, POS);
        /* Felder der Maske POS an Feld-Hostvariablen binden */
        mfn_varbind(POS_bind, (mask *)POS_mskp);
        /* der Maske POS Anwendungslogik zuordnen */
        o_install_ev_control((obj *)POS_mskp, POS_event_control);
        /* merken, dass die Masken bereits geladen sind, so dass sie bei einem erneuten
           Aufruf nicht wieder geladen werden */
        loaded = TRUE;
    }

    /* Masken bearbeiten */
    rc = (int)perform((obj *)AUFTR_mskp, NO_EVENT_CONTROL);

#ifdef DELOBJ
    /* Masken-Datenstrukturen freigeben */
    o_free((obj *)AUFTR_mskp);
    /* dann muessen die Masken bei einem erneuten Aufruf des Moduls wieder geladen
       werden */
    loaded = FALSE;
#endif      /* DELOBJ */

    return(rc);
}
```

Da jeder Maske ihre eigene Anwendungslogik zugeordnet ist, erfolgt der Aufruf von perform() ohne Übergabe einer Anwendungslogik.

Wird beim Übersetzen das Symbol DELOBJ definiert, erfolgt beim Verlassen der Modul-Prozedur eine Freigabe des von den Masken belegten Speicherplatzes. Durch Zurücksetzen der Variablen loaded wird sichergestellt, dass beim nächsten Aufruf der Modul-Prozedur die Objekte wieder neu geladen werden. Dies kann - je nach Anwendung - beträchtlich Speicherplatz sparen, führt aber zu etwas größeren Ladezeiten.

2.4 Die Funktion *maskenname*_event_control()

Zu jeder Maske wird eine Funktion *maskenname*_event_control() als Anwendungslogik generiert. Außerdem werden zu jedem Modul in einer Steuerstruktur *maskenname*_modul Informationen über den Zustand, in dem sich das Modul zurzeit befindet, und die auszuführenden Aktionen abgelegt. Diese Struktur umfasst drei Komponenten:

- mode: Modulzustand
- exec: Schalter für die Ausführung von Funktionen (z.B. Datenbankzugriff)
- restart: Schalter für die Reinitialisierung der Maske(n) und die Transaktionsbehandlung

In der Strukturkomponente mode der Steuerstruktur ist der momentane Zustand des Moduls festgehalten. Ein Modul kann sich in folgenden Zuständen befinden (d.h. mode kann folgende Werte annehmen):

NOTHING	neutraler Zustand
FX_READ	Lesen
FX_INSERT	Neuaufnahme
FX_UPDATE	Korrektur
FX_DELETE	Löschen

Die Werte von exec und restart sind nur relevant, solange ein Aufruf der Anwendungslogik andauert: sie werden von der Anwendungslogik besetzt und innerhalb des gleichen Aufrufs ausgewertet.

exec regelt die tatsächliche Ausführung der zu dem Zustand, in dem sich das Modul zurzeit befindet, gehörenden Aktion, und kann die Werte TRUE oder FALSE annehmen. Wenn ein Modul sich beispielsweise im Zustand FX_READ befindet, wird der Datenbankzugriff nur ausgeführt, wenn exec den Wert TRUE hat.

restart regelt das Zurücksetzen des Moduls und kann folgende Werte annehmen:

NOTHING	kein Zurücksetzen
ROLLBACK	Initialisieren der Maske(n), Zurücksetzen der Transaktion
NO_ROLLBACK	Initialisieren der Maske(n), kein Zurücksetzen der Transaktion

Die Anwendungslogik selbst ist dreigeteilt.

Der erste Teil (*Event-Teil*) behandelt die Events. Die Besonderheit besteht darin, dass die Aktionen nicht unmittelbar an der Stelle des Erkennens einer Situation erfolgen, sondern dies zentral zwei nachgeschaltete Teile am Ende der Anwendungslogik übernehmen, gesteuert von *maskenname*_modul. Beim Erkennen eines Events, das eine Aktion auslösen soll (wie z.B. L_NXTFIELD im Fetch-Element), wird hierzu der Zustand des Moduls neu gesetzt und die Ausführung veranlasst, indem exec den Wert TRUE erhält.

Im nachfolgenden zweiten Teil (*exec-Teil*) wird der Wert von exec überprüft. Ist er TRUE, wird die dem momentanen Zustand des Moduls entsprechende Aktion ausgeführt.

Anschließend wird der Wert von restart untersucht (*restart-Teil*). Ist restart ungleich NOTHING, werden sämtliche Masken geleert und initialisiert, das Modul wird zurückgesetzt und das Startelement besucht. Je nach dem Wert von restart wird außerdem noch die (virtuelle) Transaktion zurückgesetzt.

Beispiel:

```
static struct modul_struct     AUFTR_modul;

static Event AUFTR_event_control(obj *objp, Event event)
{
    int mfo_elemnr = CURRENTFIELD((mask *)objp);
                                        /* Position des akt. Elements oder -1 */
    int prg_elemnr = fx_prg_nr((mask *)objp, mfo_elemnr);
                                        /*Bezeichner des akt. Elements oder -1 */
    Event rc = event;                   /* Ergebnis des Aufrufs */

    /* Event-Teil */
    switch (event) {
```

```
        /* Hier findet die Event-Behandlung statt und der Rückkehrwert rc wird bestimmt.
           Beim Erkennen gewisser Situationen wird der Modulzustand neu bestimmt und
           die Ausfuehrung einer Aktion gefordert */
        …
    }

    /* exec-Teil */
    if (AUFTR_modul.exec == TRUE) {
        /* eine Aktion ist auszuführen */
        AUFTR_modul.exec = FALSE;
        /* welche Aktion auszufuehren ist, bestimmt der Modulzustand */
        switch (AUFTR_modul.mode) {
        case FX_READ:
            /* Lesen */
            …
            break;
        case FX_INSERT:
            /* Neuaufnahme */
            …
            break;
        case FX_UPDATE:
            /* Korrektur */
            …
            break;
        case FX_DELETE:
            /* Loeschen */
            …
            break;
        }
    }
    /* restart-Teil */
    if (AUFTR_modul.restart != NOTHING) {
        /* Modul soll zurueckgesetzt werden */
        …
        AUFTR_modul.restart = NOTHING;
    }

    return(rc);
}
```

Die Dreiteilung der Anwendungslogik bedingt, dass keiner ihrer Teile die Funktion vorzeitig durch eine return-Anweisung verlassen darf, da jeder Teil durch Setzen von exec und/oder restart die Steuerung verändern kann. Das Verlassen der Anwendungslogik erfolgt erst durch die return-Anweisung am Ende der Funktion.

Einige Beispiele sollen den Ablauf verdeutlichen.

Lesen eines Datensatzes

Wird im Event-Teil der Anwendungslogik festgestellt, dass das Fetch-Element der Maske vorwärts verlassen wurde (Event L_NXTFIELD), so erfolgt nach *FIX*-Logik eine Suche in der Datenbank und, bei erfolgreicher Suche, ein Einlesen des Satzes und Verzweigen in den Korrektur-Modus, andernfalls ein Verzweigen in den Neuaufnahme-Modus bzw. im Fehlerfall ein Zurücksetzen des Moduls. Die Komponente mode erhält also den Wert FX_READ (Lesen) und exec wird auf TRUE gesetzt (Lesen soll tatsächlich ausgeführt werden).

Beispiel:

```
case L_NXTFIELD:
    if (mfo_elemnr == FETCHFIELD((mask *)objp)) {
        AUFTR_modul.mode = FX_READ;
        AUFTR_modul.exec = TRUE;
    }
    …
```

Danach wird der Event-Teil mittels “break” verlassen, und es beginnt der exec-Teil.

Ist exec gleich TRUE (im Beispiel gegeben), wird mode ausgewertet und in eine Funktion verzweigt, die die Datenbank-Kommunikation (hier: Satz lesen) übernimmt. Wird ein Satz gefunden, werden die Daten in die Maskenfelder übernommen und mode erhält den Wert FX_UPDATE. Wird kein Satz gefunden, wird mode FX_INSERT zugewiesen. Tritt ein Fehler auf und die Bearbeitung muss abgebrochen werden, wird restart geeignet gesetzt (hier auf ROLLBACK, da zuvor eine Transaktion eröffnet wurde, die wieder beendet werden muss).

Beispiel:

```
    case FX_READ:
        set_obj_state(objp, READ);
        fx_begin_transaction();
        /* Daten lesen und Steuerinformation neu bestimmen */
        switch (AUFTR_modul.mode = AUFTR_IO(FX_READ)) {
        case IOERROR:
            AUFTR_modul.restart = ROLLBACK;
            rc = L_FIRSTFIELD;
            break;
        case FX_INSERT:
            set_obj_state(objp, ADD);
            break;
        case FX_UPDATE:
            m_display_data((mask *)objp);
            m_display_data((mask *)POS_mskp);
            set_obj_state(objp, UPDATE);
            break;
        }
        break;
```

Es folgt der restart-Teil. Hier werden die Masken geleert und initialisiert und das Startfeld wird zum aktuellen Element (dieses Verhalten erfolgt z.B. nach erfolgreichem Abspeichern). Enthält restart den Wert ROLLBACK (weil z.B. ein Fehler aufgetreten ist), wird zusätzlich durch fx_rollback_transaction() die Transaktion zurückgesetzt.

Beispiel:

```
if (AUFTR_modul.restart != NOTHING) {
    if (AUFTR_modul.restart == ROLLBACK)
        /* Transaktion zuruecksetzen */
        fx_rollback_transaction();
    /* Masken reinitialisieren */
    sm_empty((submask *)POS_mskp);
    restart(objp);
    set_obj_state(objp, NOTHING);
    AUFTR_modul.restart = NOTHING;
}
```

Speichern

Beim Einlesen eines Satzes wurde mode auf FX_INSERT oder FX_UPDATE gesetzt. Bei der Behandlung des Events L_NXTFIELD wird exec bei Verlassen des letzten Elements auf TRUE gesetzt.

Beispiel:

```
case L_NXTFIELD:
    …
    else if (mfo_elemnr == LASTFIELD((mask *)objp)) {
        AUFTR_modul.exec = TRUE;
    }
    …
```

Im exec-Teil wird nun mittels Aufruf der _IO-Funktion der Abgleich mit der Datenbank vorgenommen. Tritt ein Fehler auf, wird restart auf ROLLBACK gesetzt, damit die Maske(n) und die Daten wieder in den ursprünglichen Zustand zurückversetzt werden. War die Datenbankoperation hingegen erfolgreich, ist die Bearbeitung des Satzes beendet, die Transaktion wird mittels fx_commit_work() gültig und restart erhält den Wert NO_ROLLBACK, so dass nur die Maske initialisiert wird.

Beispiel:

```
case FX_INSERT:
case FX_UPDATE:
    set_obj_state(objp, WRITE);
    /* Daten speichern und Steuerinformation neu bestimmen */
    if (AUFTR_IO(AUFTR_modul.mode) == IOERROR) {
        AUFTR_modul.restart = ROLLBACK;
    }
    else {
        fx_commit_transaction();
        AUFTR_modul.restart = NO_ROLLBACK;
    }
    rc = L_FIRSTFIELD;
    break;
```

Im restart-Teil schließlich werden die Maske(n) initialisiert und - je nach Wert von restart - auch die Transaktion zurückgesetzt.

Löschen

Im Event-Teil nimmt bei der Behandlung von L_DELETE das Modul den Zustand FX_DELETE an, sofern dieser zuvor nicht FX_INSERT war (bei einer Neuaufnahme ist Löschen nicht sinnvoll). exec erhält den Wert TRUE, damit die Löschoperation ausgeführt wird, und restart den Wert NO_ROLLBACK, da nach dem Löschen des Satzes die Bearbeitung beendet ist.

Beispiel:

```
case L_DELETE:
    if (AUFTR_modul.mode == FX_INSERT
        || mfo_elemnr <= FETCHFIELD((mask*)objp))
    {
        oflash(objp, ACHTUNG, fxsystxt(M_no_delete));
        rc = L_STAY;
    }
    else {
```

```
        AUFTR_modul.mode = FX_DELETE;
        AUFTR_modul.exec = TRUE;
        AUFTR_modul.restart = NO_ROLLBACK;
    }
    break;
```

Der exec-Teil läuft analog zum Speichern ab. Mittels Aufruf der _IO-Funktion wird versucht, den Satz aus der Datenbank zu löschen. Bei Erfolg wird die Transaktion mit fx_commit_transaction() bestätigt und restart behält den Wert NO_ROLLBACK. Im Fehlerfall wird restart auf ROLLBACK gesetzt.

Beispiel:

```
case FX_DELETE:
    set_obj_state(objp, DELETE);
    if (AUFTR_IO(FX_DELETE) == IOERROR) {
        AUFTR_modul.restart = ROLLBACK;
    }
    else {
        fx_commit_transaction();
        AUFTR_modul.restart = NO_ROLLBACK;
    }
    rc = L_FIRSTFIELD;
    break;
```

Im restart-Teil wird dann die Maske geleert und die Transaktion beendet.

Abbruch der Bearbeitung durch L_PRVFIELD

Tritt das Event L_PRVFIELD im auf das Fetch-Element folgenden Element auf, erfolgt im Event-Teil eine Umsetzung von restart auf ROLLBACK. exec wird nicht gesetzt, da keine Datenbankaktion auszuführen ist.

Beispiel:

```
case L_PRVFIELD:
    if (mfo_elemnr == FETCHFIELD((mask *)objp) + 1) {
        AUFTR_modul.restart = ROLLBACK;
        rc = L_FIRSTFIELD;
    }
    else if (mfo_elemnr == STARTFIELD((mask *)objp)) {
        oflash(objp, ACHTUNG, fxsystxt(M_how_to_end));
        rc = L_STAY;
    }
    break;
```

Im folgenden restart-Teil wird die Maske initialisiert und die Transaktion mit fx_rollback_transaction() beendet, d.h. die Daten werden in ihren ursprünglichen Zustand zurückversetzt.

Achtung:

Bei geschachtelten Programm- oder Modulaufrufen kann dieses Verhalten unerwünschte Folgen haben. Wird innerhalb eines Moduls bei der Bearbeitung eines Satzes in ein anderes Modul verzweigt und dort die Bearbeitung abgebrochen, so erfolgt auch im aufrufenden Modul ein Zurücksetzen der Transaktion (vgl. Abbildung auf Seite 136).

Anmerkung zum Event-Teil

Bei der Behandlung des Events L_NXTFIELD wird von der Generierung eine Abfrage auf die Eigenschaft TOUCHED (ISTOUCHED()) vorgegeben, so dass, wenn das Feld verändert wurde, vom Programmierer reagiert werden kann. Der Aufruf von UNTOUCH(), der dem Feld die Eigenschaft TOUCHED wieder entzieht, wird mitgeneriert.

2.5 Die Funktion *maskenname*_ConsistencyControl()

Zu jeder Maske wird eine Funktion *maskenname*_ConsistencyControl() generiert, in der Felder und Daten auf Plausibilität, Integrität und Konsistenz überprüft werden können. Ein Aufruf dieser Funktion bleibt dem Entwickler überlassen.

Die Funktion erhält als Argument den Element-Bezeichner des zu überprüfenden Feldes oder das Makro ALL_FIELDS, wenn alle Felder überprüft werden sollen. In diesem Fall werden für alle Felder die in der Funktion vom Programmierer angegebenen Überprüfungen vorgenommen. Dabei kann die Reihenfolge vom Entwickler verändert werden. Zurückgeben sollte die Funktion den Element-Bezeichner eines beanstandeten Feldes oder bei erfolgreich verlaufener Konsistenzprüfung einen negativen Wert.

Die Funktion ist besonders dienlich, wenn z.B. ein Feld nach dem Verlassen auf jeden Fall (nicht nur bei L_NXTFIELD) überprüft werden muss oder wenn beim Verlassen einer Maske (L_LEAVE_OBJECT) oder eines Satzes (L_LEAVE_RECORD) alle Felder überprüft werden sollen.

Beispiel:

```
static int AUFTR_ConsistencyControl(const int prg_elemnr)
{
    switch (prg_elemnr) {
    case ALL_FIELDS:
        /* NOBREAK */
    case AUFTR_AUFNR:
        /* Code zur Kontrolle des Feldes AUFTR_AUFNR */
        if (prg_elemnr != ALL_FIELDS)
            break;
        /* NOBREAK */

    case AUFTR_DATUM:
        /* Code zur Kontrolle des Feldes AUFTR_DATUM */
        if (prg_elemnr != ALL_FIELDS)
            break;
        /* NOBREAK */

    …

    case AUFTR_AUFTEXT:
        /* Code zur Kontrolle des Feldes AUFTR_AUFTEXT */
        if (prg_elemnr != ALL_FIELDS)
            break;
        /* NOBREAK */

    case AUFTR_SUMME:
        /* Code zur Kontrolle des Feldes AUFTR_SUMME */
        if (prg_elemnr != ALL_FIELDS)
            break;

    }
    return(-1);
```

```
}
```

Da das switch-Statement ohne Entwickler-Eingriff keinen Sinn macht, wird es als Kommentar generiert.

Bei der Vier-Dateien-Variante wird diese Funktion in `source_mn.ec` vereinbart.

2.6 Datenmanipulation

Seit Version 2.9.0 wird die Datenschnittstelle in einer Form generiert, die einen hohen Abstraktionsgrad bezüglich der Datenbankzugriffe schafft. Dadurch soll die Portierbarkeit auf andere Datenbanksysteme und die Entwicklung Client-Server-orientierter Anwendungen gefördert werden.

In der Anwendungslogik der Masken wird eine “logische Datenextraktions- bzw. Modifikationsroutine” aufgerufen, die Funktion *maskenname*_IO(). Sie benutzt die *FIX*-Funktion fx_io(), um eine logische Aufgabe, beispielsweise das Lesen eines Auftrages, durchzuführen. fx_io() wiederum bedient sich hierzu generierter anwendungsspezifischer Funktionen *maskenname*_Sql(), um die notwendigen Datenbankoperationen durchzuführen.

2.6.1 Die Funktion *maskenname*_Sql()

Zu jeder Maske wird eine Funktion *maskenname*_Sql() generiert, die die grundlegenden Datenbankoperationen realisiert. Als Argument erhält diese Funktion den Bezeichner für eine Operation. Die Bezeichner sind als Makros in `fix/fx_io.h` vereinbart (im folgenden Beispiel sind alle Operationsbezeichner enthalten). Zurückgegeben wird 0 bei Erfolg bzw. ein “normierter” Fehlercode (→ fx_SC()).

Beispiel (vereinfacht):

```
int AUFTR_Sql(int op)
int op;
{
    static long locking_read_cursor_declared_at_connect_cnt = -1L;
    static long update_cursor_declared_at_connect_cnt = -1L;
                                                /* cursorspezifische Zaehler */
    int rc;

    switch (op) {
    case DECLARE_FOR_UPDATE_CURSOR:
    case DECLARE_LOCKING_READ_CURSOR:
        /* Cursor deklarieren */
        if (locking_read_cursor_declared_at_connect_cnt == fx_connect_cnt())
            return(SUCCESS);
EXEC SQL    declare AUFTR_cursor cursor for select
            aufnr, …, summe
            from auftr
            where aufnr=:q_auftr.aufnr [1]
            for update;
        update_cursor_declared_at_connect_cnt =
                    locking_read_cursor_declared_at_connect_cnt = fx_connect_cnt();
        break;

    case OPEN_FOR_UPDATE_CURSOR:
    case OPEN_LOCKING_READ_CURSOR:
```

1. Der Bedingungsteil des Statements ergibt sich aus den **with**- and **where**-Angaben in der Maskenbeschreibungsdatei.

```
            /* Cursor oeffnen */
EXEC SQL    open AUFTR_cursor;
            break;

        case FETCH_FOR_UPDATE_CURSOR:
        case FETCH_LOCKING_READ_CURSOR:
            /* naechsten Satz lesen */
EXEC SQL    fetch AUFTR_cursor into :d_auftr.aufnr, …, :d_auftr.summe;
            break;

        case SELECT_SINGLE_ROW:
            /* einzelnen Satz lesen */
EXEC SQL    select aufnr, …, summe
                into :d_auftr.aufnr, …, :d_auftr.summe
                from auftr
                where aufnr=:q_auftr.aufnr;
            break;

        case PROBE_FOR_UPDATE_CURSOR:
            /* nächsten Satz lesen, um ihn anschliessend ersetzen zu können */
EXEC SQL    fetch AUFTR_cursor;
            break;

        case UPDATE_FOR_UPDATE_CURSOR:
            /* aktuellen Satz korrigieren */
EXEC SQL    update auftr
                set aufnr = :d_auftr.aufnr, …, summe = :d_auftr.summe
                where current of AUFTR_cursor;
            break;

        case INSERT_BY_SINGLE_STATEMENT:
            /* Satz einfuegen */
EXEC SQL    insert into auftr (aufnr, …, summe)
                values (:d_auftr.aufnr, …, :d_auftr.summe);
            break;

        case DELETE_ALL_BY_SINGLE_STATEMENT:
            /* Satz bzw. Saetze loeschen */
EXEC SQL    delete from auftr where aufnr = :q_auftr.aufnr;
            break;

        case DELETE_FOR_UPDATE_CURSOR:
            /* aktuellen Satz löschen */
EXEC SQL    delete from auftr where current of AUFTR_cursor;
            break;

        case CLOSE_FOR_UPDATE_CURSOR:
        case CLOSE_LOCKING_READ_CURSOR:
            /* Cursor schliessen */
EXEC SQL    close AUFTR_cursor;
            break;
/*
        die folgenden Anweisungen stehen in Kommentarklammern, da das Freigeben eines
        Cursors von IBM Informix ESQL/C erst ab Version 4.0 unterstuetzt wird und der
```

```
        IBM Informix ESQL/C-Praeprozessor Direktiven fuer den C-Praeprozessor nicht
        beruecksichtigt
    #ifdef FREE_CURSOR
        case FREE_FOR_UPDATE_CURSOR:
        case FREE_LOCKING_READ_CURSOR:
    EXEC SQL    free AUFTR_cursor;
            locking_read_cursor_declared_at_connect_cnt = -1L;
            update_cursor_declared_at_connect_cnt = -1L;
            break;
    #endif
    */
        default:
            return(FCT_NOT_IMPLEMENTED);
        } /* switch */

        /* es wurde eine Datenbankanweisung ausgefuehrt; anhand des Wertes in
           SC wird geprueft, ob ein (schwerwiegender) Fehler aufgetreten ist */
        if ((rc = fx_SC(SC, op)) == IOERROR)
            sql_error(SC, "in AUFTR_Sql()");

        return(rc);
    }
```

Neben den obigen Cursorn enthalten die _Sql-Funktionen auch Code für einen READONLY-Cursor: für diesen sind die Operationen

```
DECLARE_READONLY_CURSOR
OPEN_READONLY_CURSOR
FETCH_READONLY_CURSOR
CLOSE_READONLY_CURSOR
FREE_READONLY_CURSOR      (ebenfalls auskommentiert)
```

vorgesehen.[1]

Bei der Vier-Dateien-Variante sind diese Funktionen in `source_sq.ec` enthalten.

2.6.2 Die IO-Funktion *maskenname*_IO()

Dieser Funktion liegt der Gedanke zugrunde, dass man bei der Programmierung ein Datenobjekt, etwa einen Auftrag, eher als Einheit sieht, anstatt eine starke inhaltliche Trennung in dessen Bestandteile vorzunehmen. Dies kommt auch einem objektorientierten Programmierstil näher.

Kennzeichen der Funktion *maskenname*_IO() ist, dass von ihr *alle* Datenbankaktionen im Zusammenhang etwa mit dem Lesen des Auftrags vorgenommen werden, d.h. es werden die Daten für die *Haupt*maske und die darin eingebetteten Masken beschafft. Das Gleiche gilt für das Zurückschreiben.

Pro Modul wird eine _IO-Funktion generiert. Die _IO-Funktion wird jeweils im exec-Teil der Anwendungslogik der Hauptmaske aufgerufen. Sie erhält als Argument eine Zugriffsart entsprechend dem augenblicklichen Zustand des Moduls (FX_READ, FX_INSERT, FX_UPDATE, FX_DELETE) und liefert bei Erfolg als Resultat den Zustand zurück, in den das Modul versetzt werden soll (z.B. nach erfolgreichem Einlesen FX_INSERT oder FX_UPDATE), ansonsten einen Fehler (IOERROR).

1. Benutzt werden diese Operationen von der Methode METH_READONLY_CURSOR; vgl. hierzu Abschnitt 2.6.2 auf Seite 477.

Zur Durchführung der Datenbankzugriffe verwendet *maskenname*_IO() die *FIX*-Funktion fx_io(), die zu jeder der Zugriffsarten Lesen, Einfügen, Korrigieren und Löschen verschiedene Vorgehensweisen - *Methoden* - kennt. Generiert wird immer die Vorgehensweise STANDARD.

fx_io() ist eine universelle Funktion, die mit Hilfe der vom Entwickler bereitgestellten _Sql-Funktion und Kopierfunktion gewisse Methoden der Datenextraktion oder -manipulation realisiert. fx_io() hat folgende Form:

```
int fx_io(int mode, int method, obj *obj,
        int (*sql_fct)(int), void (*copy_fct)(int), char *usr_data)
```

wobei

mode die Zugriffsart angibt (FX_READ, FX_INSERT, FX_UPDATE, FX_DELETE),

method die Methode angibt, nach der der Zugriff vorgenommen werden soll (generiert wird die Methode STANDARD),

obj ein Zeiger auf eine Einzelsatz-Maske oder Mehrsatz-Maske ist, die die Daten enthält,

sql_fct ein Zeiger auf die Funktion ist, die die Datenbankoperationen realisiert,

copy_fct ein Zeiger auf die Funktion ist, die die Daten in die entsprechenden Datenbank- bzw. Feld-Hostvariablen (und umgekehrt) transportiert (das Makro NO_COPY teilt fx_io() mit, dass kein Transport erfolgen muss),

usr_data ein Zeiger auf Daten ist, die eventuell von einer von STANDARD abweichenden Methode *method* benötigt werden (NO_USR_DATA zeigt fx_io() an, dass solche nicht erforderlich sind).

Die folgende Übersicht zeigt, welche Operationen die Funktion *sql_fct* für die Zugriffsart *mode* nach der Vorgehensweise *method* mindestens unterstützen muss:

Zugriffsart	Methode	
FX_READ	METH_LOCKING_READ_CURSOR (STANDARD)	
	eigenständige Einzelsatz-Maske	Mehrsatz-Maske
	DECLARE_FOR_UPDATE_CURSOR	DECLARE_LOCKING_READ_CURSOR
	OPEN_FOR_UPDATE_CURSOR	OPEN_LOCKING_READ_CURSOR
	FETCH_FOR_UPDATE_CURSOR	FETCH_LOCKING_READ_CURSOR
	CLOSE_FOR_UPDATE_CURSOR	CLOSE_LOCKING_READ_CURSOR
	FREE_FOR_UPDATE_CURSOR	FREE_LOCKING_READ_CURSOR
	vgl. Abschnitt 3, Abb. 37	vgl. Abschnitt 3, Abb. 39
	eingebettete Einzelsatz-Maske	
	SELECT_SINGLE_ROW	
	vgl. Abschnitt 3, Abb. 37	
FX_INSERT	METH_SINGLE_STATEMENT (STANDARD)	
	Einzelsatz-Maske	Mehrsatz-Maske
	INSERT_BY_SINGLE_STATEMENT	INSERT_BY_SINGLE_STATEMENT
	vgl. Abschnitt 3, Abb. 40	vgl. Abschnitt 3, Abb. 41
FX_UPDATE	METH_LOCKING_READ_CURSOR (STANDARD)	
	eigenständige Einzelsatz-Maske	Mehrsatz-Maske
		DECLARE_FOR_UPDATE_CURSOR
		OPEN_FOR_UPDATE_CURSOR
		PROBE_FOR_UPDATE_CURSOR
	UPDATE_FOR_UPDATE_CURSOR	UPDATE_FOR_UPDATE_CURSOR
		DELETE_FOR_UPDATE_CURSOR
	CLOSE_FOR_UPDATE_CURSOR	CLOSE_FOR_UPDATE_CURSOR
	FREE_FOR_UPDATE_CURSOR	FREE_FOR_UPDATE_CURSOR
		INSERT_BY_SINGLE_STATEMENT
	vgl. Abschnitt 3, Abb. 42	vgl. Abschnitt 3, Abb. 44

eingebettete Einzelsatz-Maske
wie Mehrsatz-Maske
vgl. Abschnitt 3, Abb. 39

FX_UPDATE METH_UNIQUE_KEY

Einzelsatz-Maske	Mehrsatz-Maske
nicht unterstützt	DECLARE_FOR_UPDATE_CURSOR OPEN_FOR_UPDATE_CURSOR FETCH_FOR_UPDATE_CURSOR DELETE_FOR_UPDATE_CURSOR CLOSE_FOR_UPDATE_CURSOR FREE_FOR_UPDATE_CURSOR INSERT_BY_SINGLE_STATEMENT vgl. Abschnitt 3, Abb. 45

FX_UPDATE METH_DELETE :

Einzelsatz-Maske	Mehrsatz-Maske
nicht unterstützt	DELETE_ALL_BY_SINGLE_STATEMENT INSERT_BY_SINGLE_STATEMENT vgl. Abschnitt 3, Abb. 46

FX_DELETE METH_LOCKING_READ_CURSOR (STANDARD)

eigenständige Einzelsatz-Maske	Mehrsatz-Maske
	DECLARE_FOR_UPDATE_CURSOR OPEN_FOR_UPDATE_CURSOR PROBE_FOR_UPDATE_CURSOR
DELETE_FOR_UPDATE_CURSOR	DELETE_FOR_UPDATE_CURSOR
CLOSE_FOR_UPDATE_CURSOR	CLOSE_FOR_UPDATE_CURSOR
FREE_FOR_UPDATE_CURSOR	FREE_FOR_UPDATE_CURSOR
vgl. Abschnitt 3, Abb. 47	vgl. Abschnitt 3, Abb. 49
eingebettete Einzelsatz-Maske	
wie Mehrsatz-Maske	
vgl. Abschnitt 3, Abb. 39	

Methoden dienen dazu, eine einheitliche, erweiterbare Schnittstelle für den Programmierer zu schaffen, mit der - neben den mitgelieferten Standard-Methoden von *FIX* - anwendungsspezifische Datenzugriffsmethoden definiert und verwendet werden können.

Weiterhin möchten wir uns bei der Weiterentwicklung von *FIX* die Möglichkeit offen halten, schon absehbare und künftige Performance-Verbesserungen beim Datenbanksystem (Trigger, kaskadierendes Löschen etc.) mit minimalem Pflegeaufwand am Sourcecode zu realisieren. In naher Zukunft sollen die dynamischen Datenbankzugriffe über eine dem Sourcecode gleichende Schnittstelle ansprechbar sein und es so dem Entwickler erlauben, die Zugriffsfunktion von Fall zu Fall auszutauschen.

Die Funktion *maskenname*_IO() hat folgende Form:

Beispiel:

```
static int AUFTR_IO(const int mode)
{
    switch (mode) {
    case FX_READ:
        /* Zugriffsart Lesen */
        switch (fx_io(mode, STANDARD, (obj *)AUFTR_mskp,
                                        AUFTR_Sql, AUFTR_Copy, NO_USR_DATA))
        {
        case SUCCESS:
            /* Datensatz in Tabelle AUFTR gefunden */
```

```
                if (fx_io(mode, STANDARD, (obj *)POS_mskp, POS_Sql,
                                                    POS_Copy, NO_USR_DATA) < 0)
                    return(IOERROR);
                return(FX_UPDATE);
            case FX_SQLNOTFOUND:
                /* kein Datensatz in Tabelle AUFTR gefunden */
                return(FX_INSERT);
            case FX_SQLLOCKED:
                /* Datensatz in Tabelle AUFTR gesperrt */
                rt_msg(ACHTUNG, fxsystxt(M_locked));
                /* NOBREAK */
            default:
                /* Fehler beim Zugriff auf Tabelle AUFTR */
                return(IOERROR);
            }
            break;

        case FX_INSERT:
            /* Zugriffsart Einfuegen */
        case FX_UPDATE:
            /* Zugriffsart Korrigieren */
        case FX_DELETE:
            /* Zugriffsart Loeschen */
            if (fx_io(mode, STANDARD, (obj *)AUFTR_mskp,
                                    AUFTR_Sql, AUFTR_Copy, NO_USR_DATA) < 0)
                return(IOERROR);
            if (fx_io(mode, STANDARD, (obj *)POS_mskp, POS_Sql,
                                                    POS_Copy, NO_USR_DATA) < 0)
                return(IOERROR);
            return(SUCCESS);
        }
        return(IOERROR);
    }
```

Bei der Vier-Dateien-Variante ist diese Funktion in *source*_mn.c enthalten.

Redeklaration und Freigabe von Cursorn

Durch Definition des Symbols FREE_CURSOR wird eine Möglichkeit geschaffen, Cursor freizugeben und ggf. zu redeklarieren.[1] Durch diese Methode ist außerdem ein Datenbankwechsel zur Laufzeit einfacher möglich. Hierzu existieren zwei Zähler: ein globaler Zähler, der für die gesamte Laufzeit des Prozesses gilt, und ein cursorspezifischer Zähler.

Der globale Zähler (Datenbank-Zähler) wird bei jedem Verbindungsaufbau zu einer Datenbank (→ fx_database_connect()) um 1 hochgezählt. Bei der Deklaration oder dem Öffnen eines Cursors wird der cursorspezifische Zähler mit dem Datenbank-Zähler verglichen. Sind beide verschieden - wie z.B. beim ersten Öffnen des Cursors, wenn der Datenbank-Zähler bereits um 1 erhöht wurde, der cursorspezifische Zähler jedoch noch nicht - erfolgt eine Deklaration des Cursors und der Wert des Datenbank-Zählers wird in dem cursorspezifischen Zähler gespeichert. Damit gilt dieser Cursor als bekannt, und beim nächsten OPEN erfolgt kein erneutes Deklarieren des Cursors, da Datenbank- und cursorspezifischer Zähler identisch sind.

Wird der Cursor freigegeben, wird der cursorspezifische Zähler auf -1 gesetzt. Beim nächsten Öffnen (bzw. einer erneuten Deklaration) werden die beiden Zähler wiederum miteinander verglichen, und da nun die Werte nicht mehr identisch sind, erfolgt eine Deklaration.

1. IBM Informix ESQL/C 2.1 kennt kein Freigeben von Cursorn; deshalb ist das Setzen von FREE_CURSOR hier unzulässig.

Beim Datenbankwechsel erhöht sich der Wert des Datenbank-Zählers um 1. Damit sind beim nächsten Öffnen eines Cursors Datenbank-Zähler und cursorspezifischer Zähler unterschiedlich, und es erfolgt eine Deklaration, nun für die neue Datenbank.

Diese Vorgehensweise wurde gewählt, da bei einem Datenbankwechsel alle deklarierten Cursor ungültig werden (können).

Schematisches Beispiel:

```
…
/* Datenbank 1 öffnen, vor Aufruf von fx_database_connect() sind alle Zähler = -1 */
fx_database_connect("Datenbank_1");
/* Datenbank-Zähler = 0, cursorspezifische Zähler = -1 */
…
AUFTR_IO(FX_READ);
/* Vor Aufruf: Datenbank-Zähler = 0, cursorspezifische Zähler (für AUFTR_cursor und POS_cursor) = -1.
Dadurch automatische Deklaration der betroffenen Cursor.
Nach Aufruf: Datenbank-Zähler = 0, cursorspezifische Zähler (AUFTR_cursor und POS_cursor) = 0 */
…
/* Freigeben der Cursor "von Hand" */
EXEC SQL free AUFTR_cursor; Zähler_für_AUFTR_cursor = -1;
/* Datenbank-Zähler = 0, cursorspezifischer Zähler (nur für AUFTR_cursor) = -1 */
…
AUFTR_IO(FX_READ);
/* Vor Aufruf: Datenbank-Zähler = 0, cursorspezifischer Zähler für AUFTR_cursor = -1, für POS_cursor = 0.
Automatische Deklaration für AUFTR_cursor, keine Deklaration für POS_cursor, da dieser noch bekannt ist.
Nach dem Aufruf: Datenbank-Zähler = 0, cursorspezifische Zähler (für AUFTR_cursor und POS_cursor) = 0 */
…
/* Wechsel zu Datenbank 2 */
fx_database_connect("Datenbank_2");
/* Datenbank-Zähler = 1, cursorspezifische Zähler = 0 */
…
AUFTR_IO(FX_READ);
/* Vor dem Aufruf: Datenbank-Zähler = 1, cursorspezifische Zähler (AUFTR_cursor und POS_cursor) = 0.
Automatische Deklaration für beide Cursor und damit Bekanntmachen der Cursor für die neue Datenbank. Nach
dem Aufruf: Datenbank-Zähler = 1, cursorspezifische Zähler = 1 */
…
/* Zurück zu Datenbank 1 */
fx_database_connect("Datenbank_1");
/* Datenbank-Zähler = 2, cursorspezifische Zähler = 1 */
…
AUFTR_IO(FX_READ);
/* Vor dem Aufruf: Datenbank-Zähler = 2, cursorspezifische Zähler (AUFTR_cursor und POS_cursor) = 1.
Automatische Deklaration für beide Cursor und damit Bekanntmachen der Cursor für Datenbank 1. Nach dem
Aufruf: Datenbank-Zähler = 2, cursorspezifische Zähler = 2 */
```

3 Beschreibung der Methoden

Die folgenden Abbildungen legen dar, in welcher Weise ein Aufruf der Form

fx_io(*mode*, *method*, *obj*, *sql_fct*, *copy_fct*, *usr_data*)

die von *sql_fct* zur Verfügung gestellten Operationen benutzt, um die Zugriffsart *mode* in der Vorgehensweise *method* für das Objekt *obj* zu realisieren.

Abb. 37 FX_READ - METH_LOCKING_READ_CURSOR - eigenständige Einzelsatz-Maske

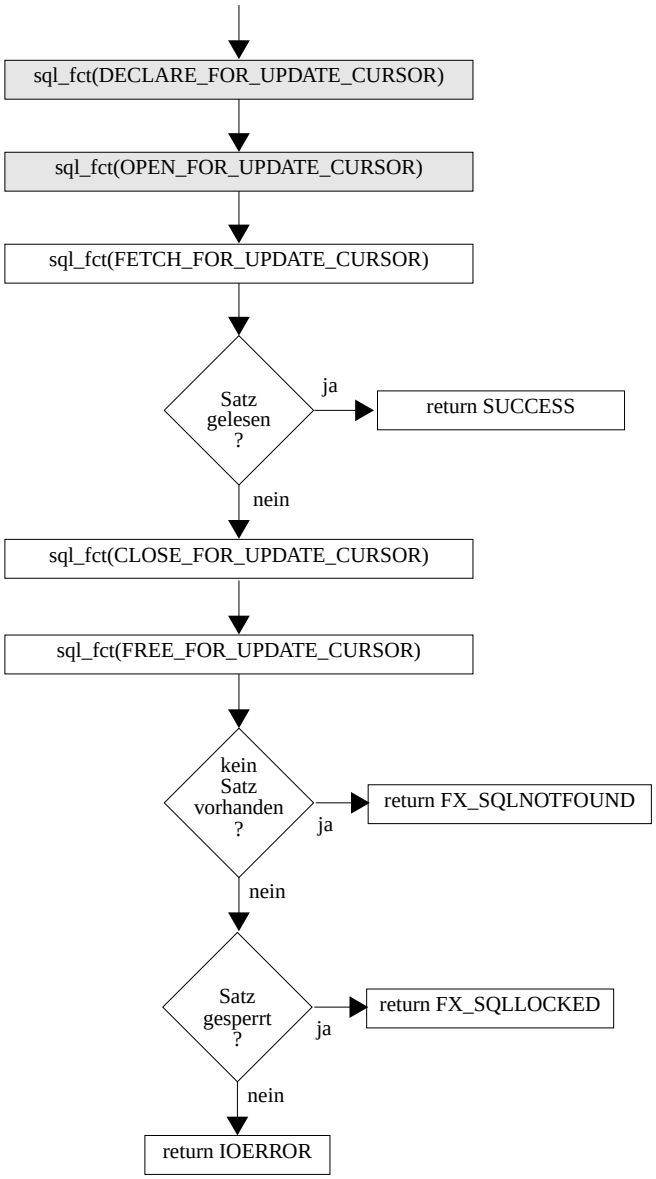


Abb. 38 FX_READ - METH_LOCKING_READ_CURSOR - eingebettete Einzelsatz-Maske

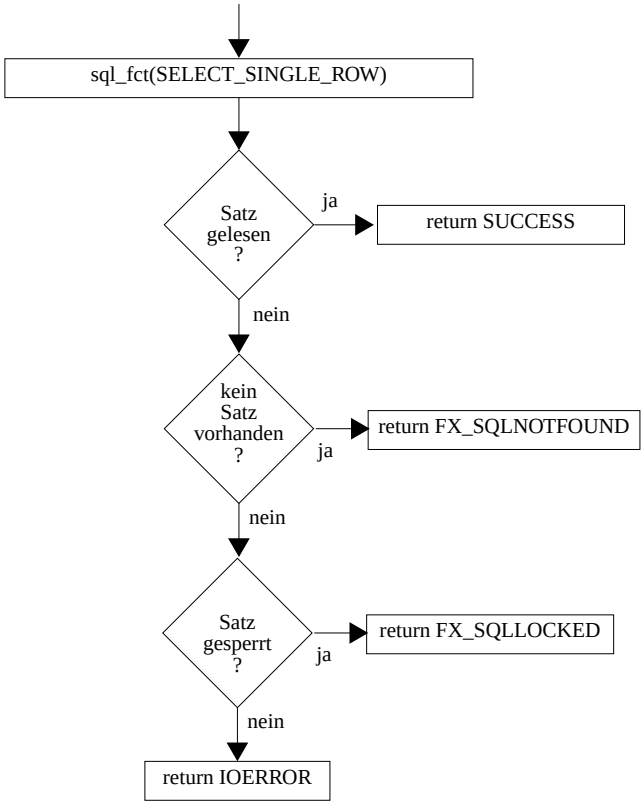


Abb. 39 FX_READ - METH_LOCKING_READ_CURSOR - Mehrsatz-Maske

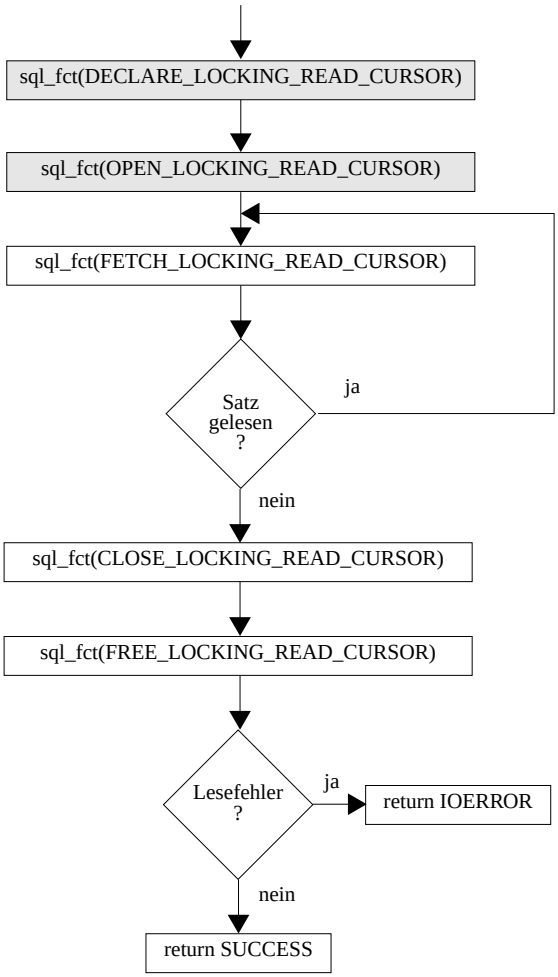


Abb. 40 FX_INSERT - METH_SINGLE_STATEMENT - Einzelsatz-Maske

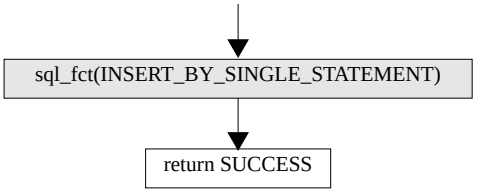


Abb. 41 FX_INSERT - METH_SINGLE_STATEMENT - Mehrsatz-Maske

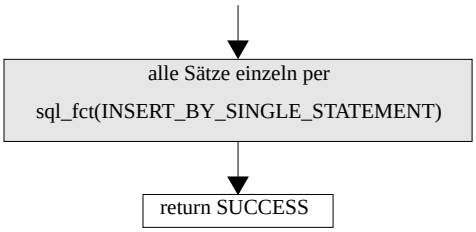


Abb. 42 FX_UPDATE - METH_LOCKING_READ_CURSOR - eigenständige Einzelsatz-Maske

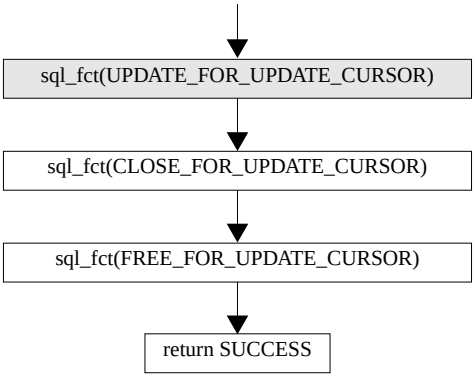


Abb. 43 FX_UPDATE - METH_LOCKING_READ_CURSOR - eingebettete Einzelsatz-Maske

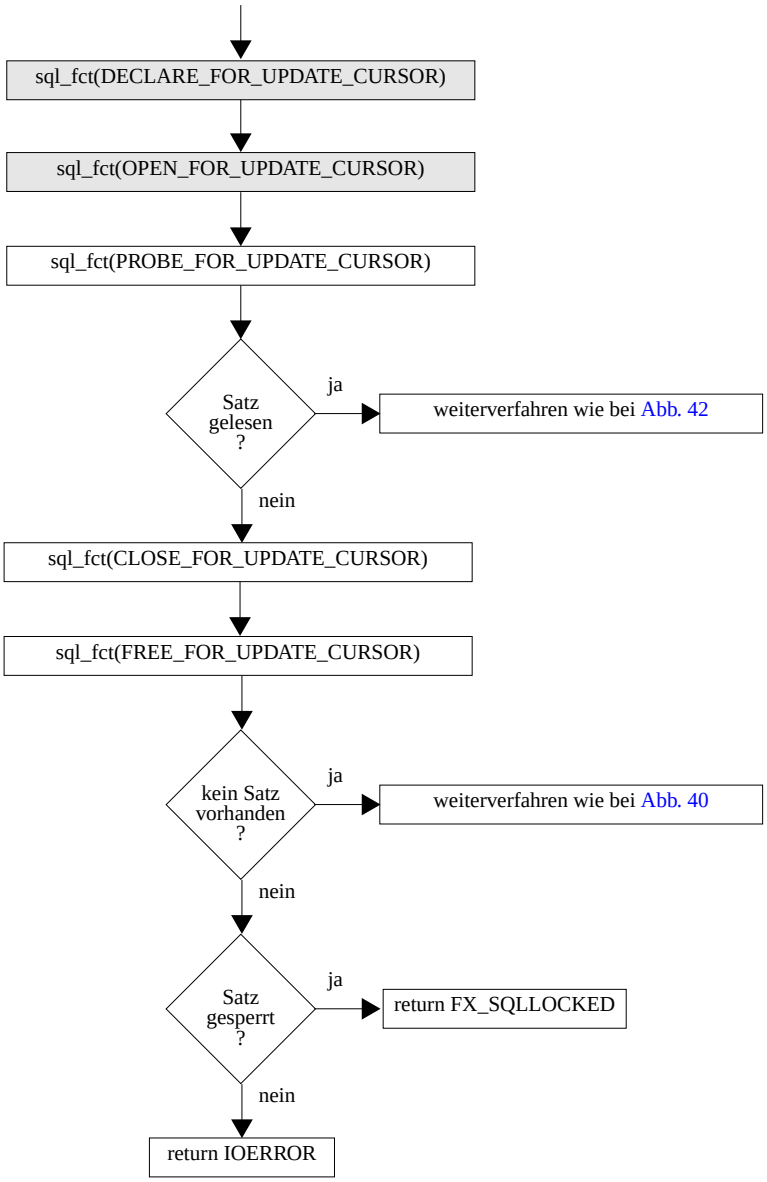


Abb. 44 FX_UPDATE - METH_LOCKING_READ_CURSOR - Mehrsatz-Maske

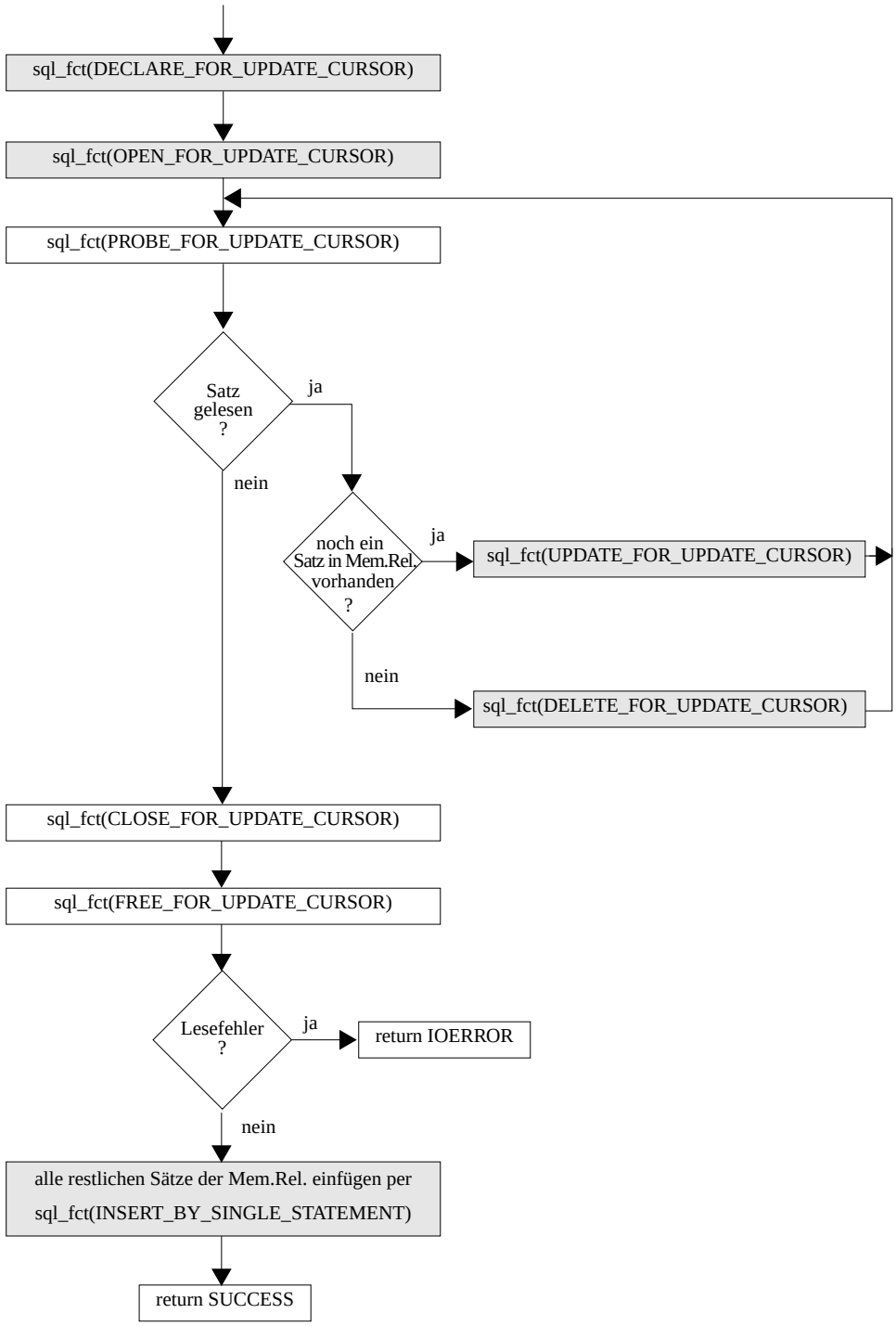


Abb. 45 FX_UPDATE - METH_UNIQUE_KEY - Mehrsatz-Maske

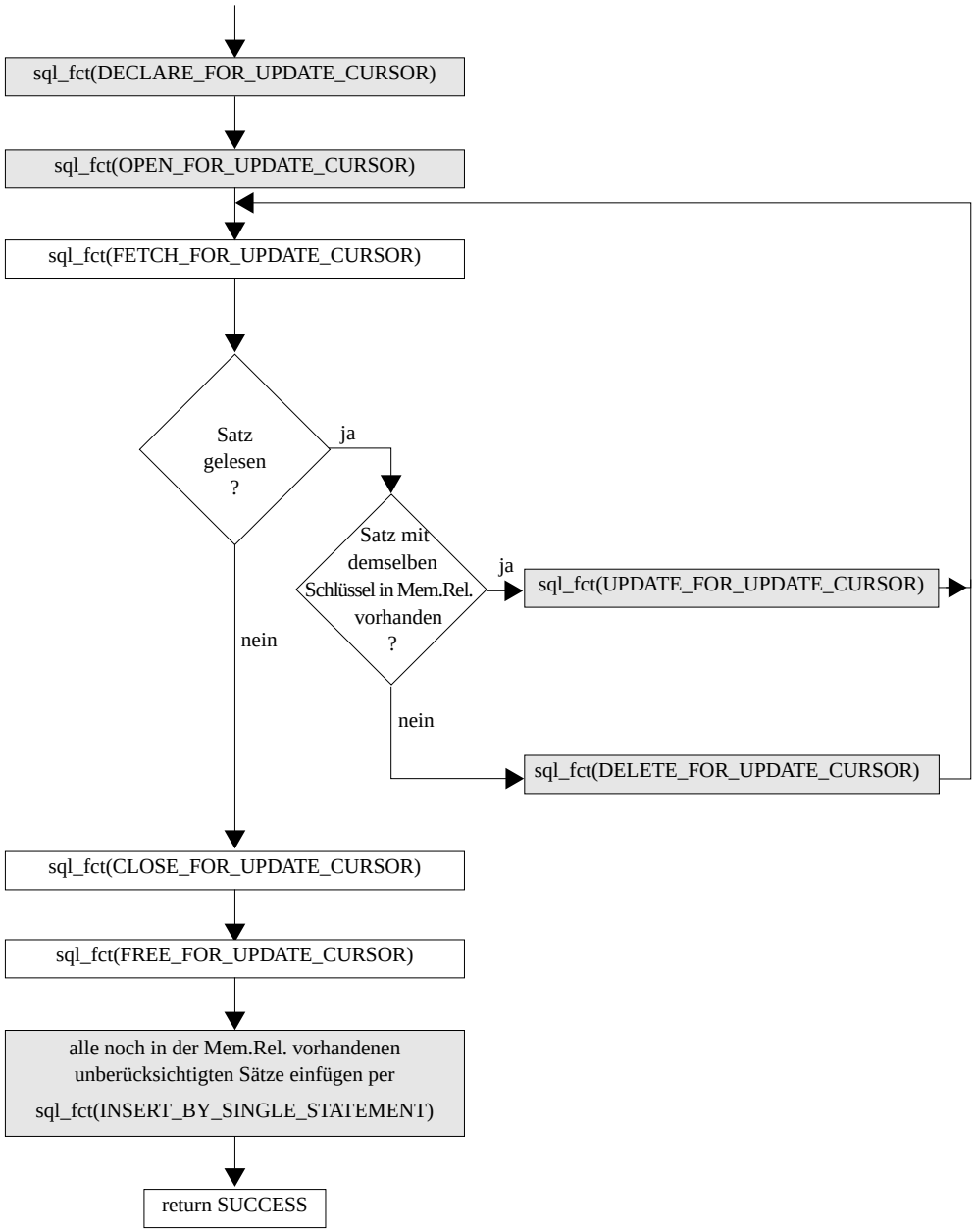


Abb. 46 FX_UPDATE - METH_DELETE - Mehrsatz-Maske

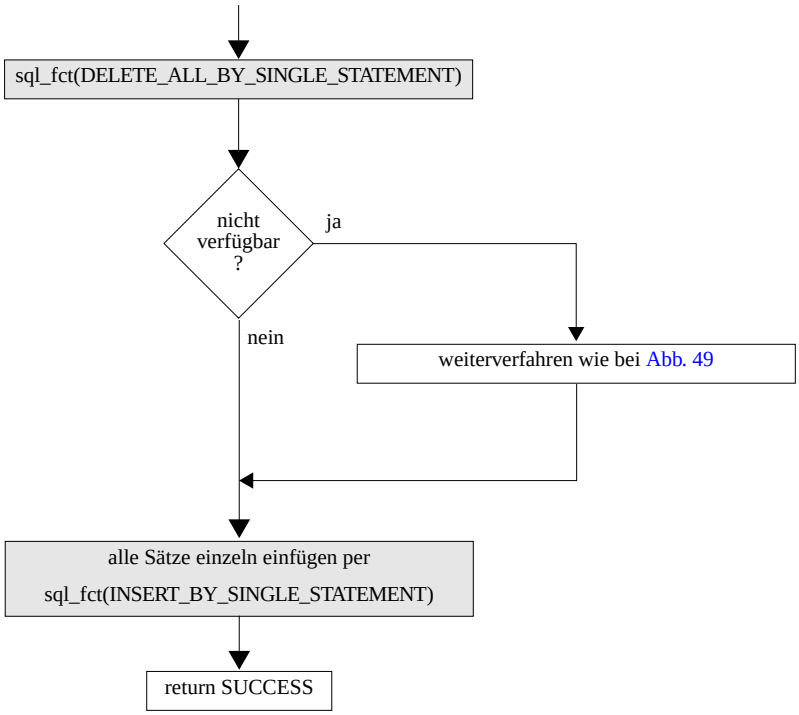


Abb. 47 FX_DELETE - METH_LOCKING_READ_CURSOR - eigenständige Einzelsatz-Maske

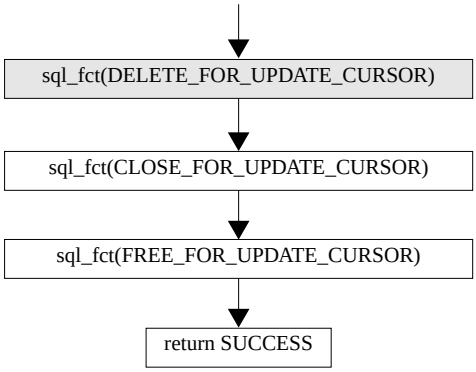


Abb. 48 FX_DELETE - METH_LOCKING_READ_CURSOR - eingebettete Einzelsatz-Maske

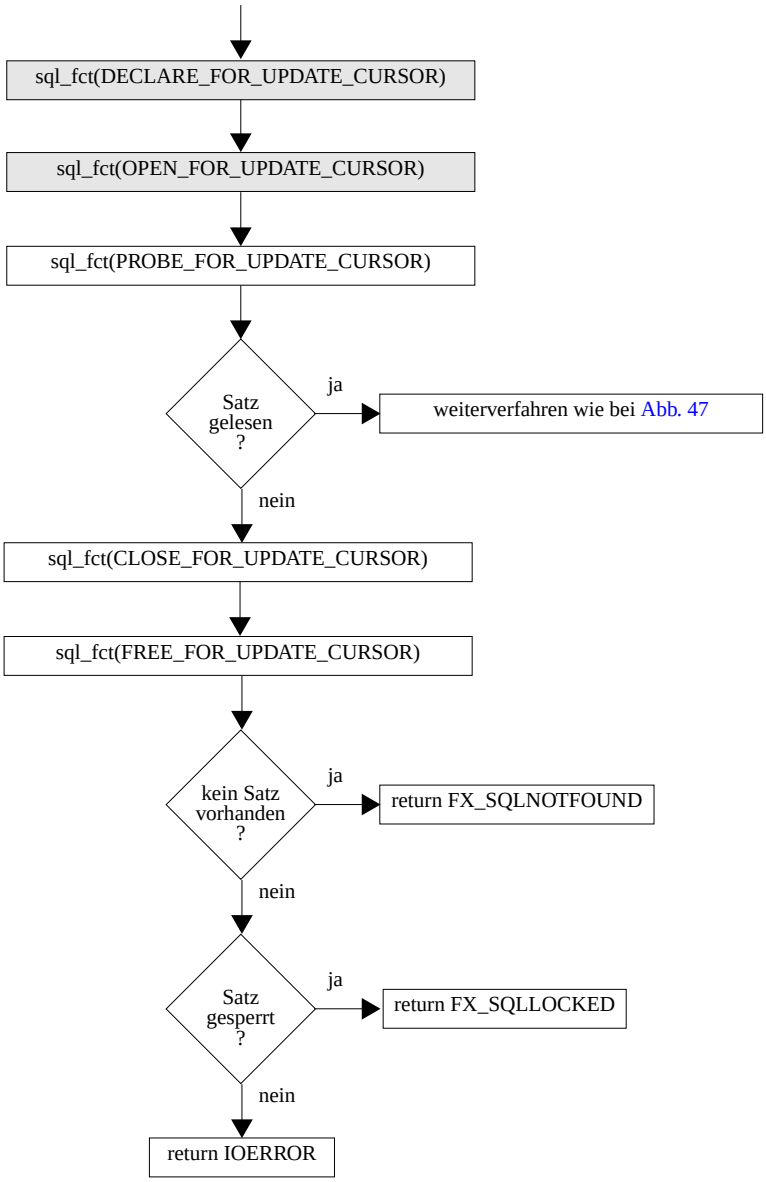


Abb. 49 FX_DELETE - METH_LOCKING_READ_CURSOR - Mehrsatz-Maske

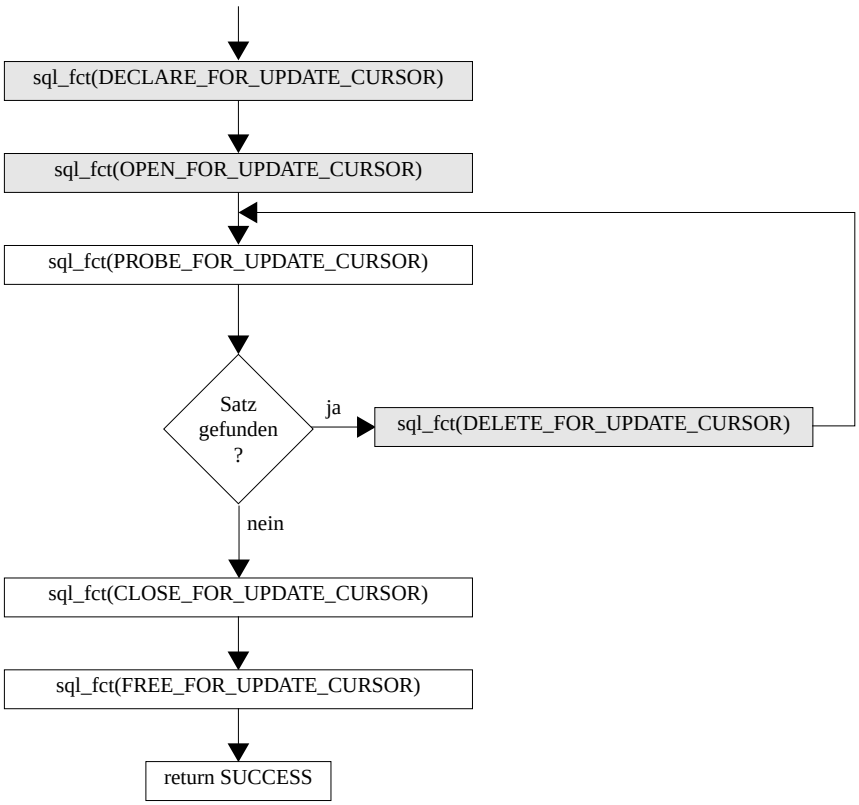


Abb. 50 FX_READ - METH_READONLY_CURSOR - eigenständige Einzelsatz-Maske

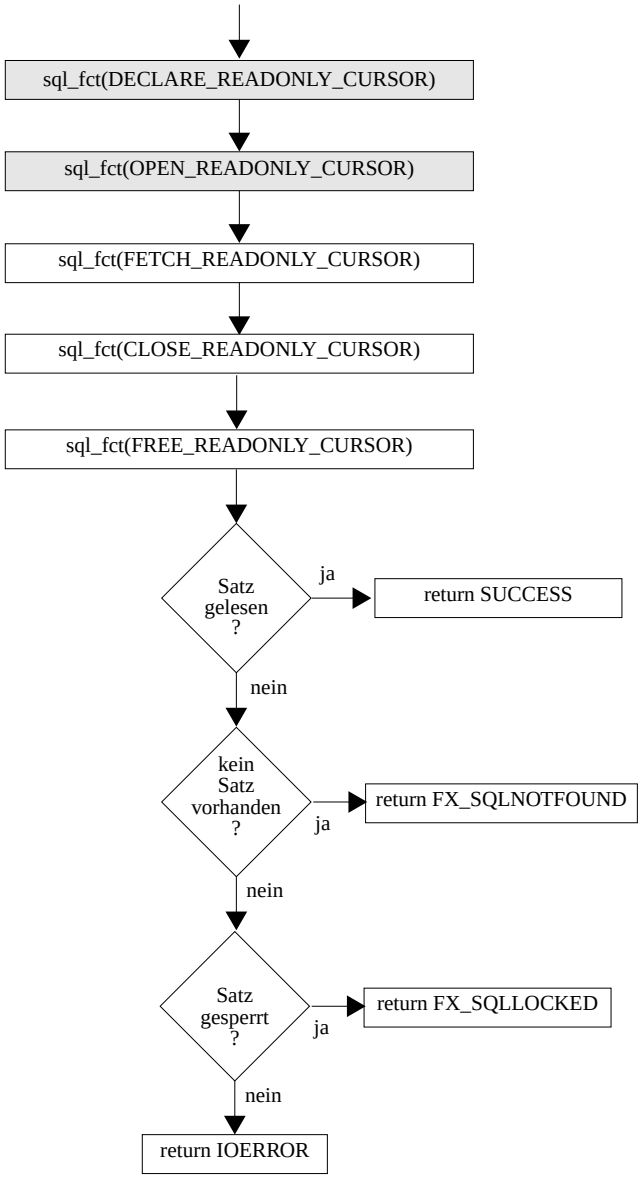


Abb. 51 FX_READ - METH_READONLY_CURSOR - eingebettete Einzelsatz-Maske

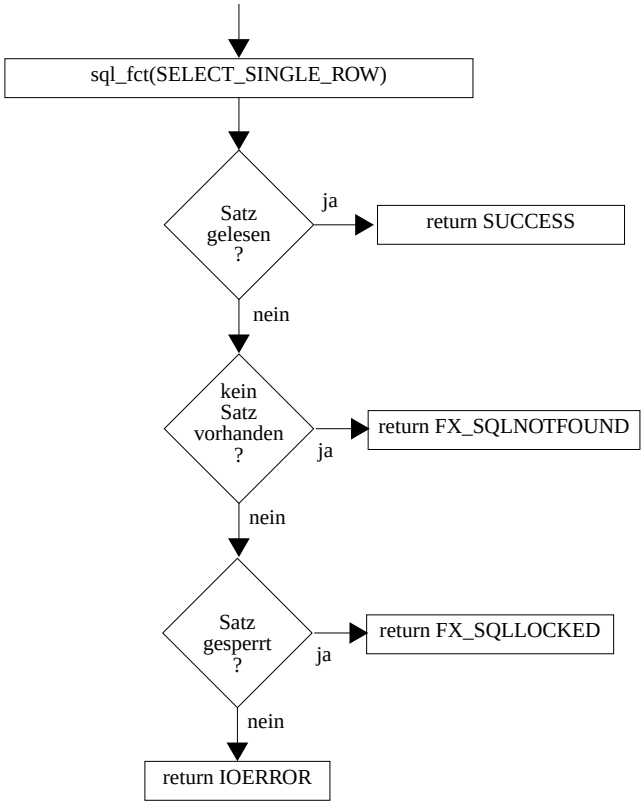
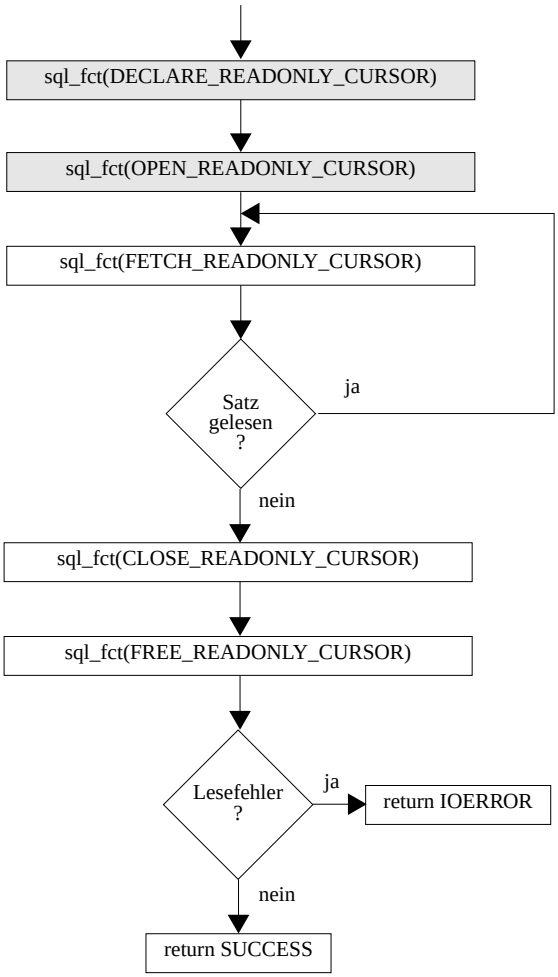


Abb. 52 FX_READ - METH_READONLY_CURSOR - Mehrsatz-Maske



Liefert einer der markierten Aufrufe von *sql_fct* einen Fehler (Ergebnis ungleich SUCCESS), bricht fx_io() mit dem Ergebnis IOERROR ab. Bei den nicht markierten Aufrufen wird dagegen mit der Bearbeitung fortgefahren.

4 Aufruf der Generierung aus der Shell

Die Generierung wird wohl meist aus dem Entwicklermenü **fxm**, kann aber auch von der Shell aus aufgerufen werden.

Beim Aufruf muss unterschieden werden, ob ein Programm erzeugt werden soll, einzelne Programm-Komponenten oder einzelne Source-Dateien nachgeneriert werden sollen oder nur ein Eintrag im Makefile erzeugt werden soll.

Das Generierungsprogramm **fgmask** bekommt durch den ersten Schalter mitgeteilt, was generiert werden soll, und erhält als Argumente die Maskenbeschreibungsdatei und wahlweise den Basisnamen für die Zieldatei(en):

- Zur Erzeugung von Programmen ist als erstes Argument -p anzugeben:

  fgmask -p [-1, -o, -n<num>, -m] <beschreibungsdatei> [<basisname>] [-]

  Bei Angabe von -1 wird das Programm in einer Datei abgelegt. Bei -o wird zwischen Feld- und Datenbank-Hostvariablen unterschieden. Mit -n<num> kann ein Anfangswert für die Elementbezeichner vorgegeben werden. Bei Angabe von -m werden geeignete Regeln nach `makefile` geschrieben.

- Zur Generierung von Komponenten ist als erstes Argument -k anzugeben:

  fgmask -k [-1, -o, -n<num> , -<flags>] <beschreibungsdatei> [<basisname>] [-]

  Bei -1 wird von einer Programmdatei ausgegangen. Die Schalter -o und -n<num> haben die gleiche Bedeutung wie oben. Die Elemente von <flags> bedeuten:

  d Element-Bezeichner generieren
  b Datenstrukturen d_*tabelle* generieren
  a Datenstrukturen m_*maskenname* generieren
  q Datenstrukturen q_*maskenname* generieren
  v mfn_bind-Arrays generieren
  c Copy-Funktionen generieren
  i IO-Funktion generieren
  s Sql-Funktionen generieren
  e Anwendungslogik-Funktionen generieren
  m Regeln im Makefile generieren
  p Modul-Prozedur generieren
  sowie
  R Rekursiv für alle eingebetteten Masken

- Zum Nachgenerieren von Dateien ist als erstes Argument -i anzugeben:

  fgmask -ih [-o, -n<num>] <beschreibungsdatei> [<basisname>] [-] /* .h */
  fgmask -im [-o] <beschreibungsdatei> [<basisname>] [-] /*_ms .c */
  fgmask -is [-o] <beschreibungsdatei> [<basisname>] [-] /* _sq.ec */
  fgmask -in [-o] <beschreibungsdatei> [<basisname>] [-] /* _mn.c */

  Die Schalter -o und -n<num> haben die gleiche Bedeutung wie oben.

- Zum Erzeugen eines Eintrags im Makefile ist als erstes Argument -m anzugeben:

  fgmask -m [-1] <beschreibungsdatei> [<basisname>] [-]

  Der Schalter 1 hat die gleiche Bedeutung wie oben.

Wird als letztes Argument '-' angegeben, erfolgt die Ausgabe nach stdout. Wird kein Basisname angegeben, wird der Maskenname als Basisname übernommen.

5 Die Funktion perf()

perf() stellt eine bequeme Alternative zur Generierung einer Masken-Source dar, wenn die in der Anwendung benötigte Logik nicht allzu sehr von der generierten Logik abweicht:

```
Event perf(char *maskendatei,
        Event (*usr_control)(obj *, Event ),
        void (*usr_startup)(obj *))
```

perf() bietet das in der Beschreibungsdatei *maskendatei* hinterlegte Objekt, bei dem es sich um eine Maske handeln muss, zur Bearbeitung an. Ist die entsprechende Maske bereits geladen, so wird das geladene Exemplar benutzt[1], anderenfalls die Maske aus der Datei *maskendatei* zunächst geladen[2] und die Funktion *usr_startup* mit dem Maskenzeiger als Argument gerufen.

Die Maske wird anschließend mittels perform(…, *usr_control*) zur Bearbeitung angeboten. Handelt es sich um eine Einzelsatz-Maske, wird hierbei eine "erweiterte Standardlogik" benutzt, die vorab eine Anwendungslogik emuliert, wie sie *FIX* für die Maske generieren würde.[3] Hierzu stellt *FIX* eine temporäre _modul-Struktur (vgl. Seite 469) und eine eingebaute allgemeine _IO-Funktion (siehe Seite 501) bereit, bei der die Datenbankzugriffe mittels der in Abschnitt 5.2 beschriebenen Funktion perf_sql() erfolgen.

Die für die Aufrufe von perf_sql() benötigten Kontexte (vgl. Seite 497) werden von perf() vorab bestimmt und in dynamisch allokierten Datenstrukturen hinterlegt, die vor der Rückkehr aus der Funktion perf() wieder freigegeben werden. So ist sichergestellt, dass fx_io() bereits einen Kontext vorfindet und diesen nicht erst berechnen muss (Letzterer bliebe bis zum Programmende erhalten, da ja jederzeit weitere Aufrufe folgen könnten).

Ergebnis von perf() ist der Wert, den perform() zurückgegeben hat.

5.1 Erweiterte Standardlogik

Ein vor der ersten Phase der Standardlogik (vgl. "Schritt 3a: Event-Behandlung bzgl. des Satzes" auf Seite 200) eingeschobener Zwischenschritt prüft - analog zu einer generierten Anwendungslogik - die aktuelle Situation (in der unten angegebenen Reihenfolge) und stößt ggf. die nachfolgend aufgeführte Aktion an:

- L_NXTFIELD im Fetch-Element (vgl. Seite 471):
  Das Modul wird in den Zustand FX_READ versetzt, exec erhält den Wert TRUE.
- L_NXTFIELD im letzten Element (vgl. Seite 472):
  exec erhält den Wert TRUE.
- DELETE, während sich das Modul im Zustand FX_INSERT befindet (vgl. Seite 472):
  Hier erfolgt eine Meldung und die Standardlogik wird veranlasst, statt L_DELETE das Event L_STAY zu behandeln.
- DELETE, während sich das Modul nicht im Zustand FX_INSERT befindet (vgl. Seite 472):
  Das Modul wird in den Zustand FX_DELETE versetzt, exec erhält den Wert TRUE, restart den Wert NO_ROLLBACK.
- L_END in einem Element "hinter" dem Fetch-Element:
  Hier erfolgt eine Meldung, und die Standardlogik wird veranlasst, statt L_END das Event L_STAY zu behandeln.
- L_PRVFIELD in dem auf das Fetch-Element folgenden Element (vgl. Seite 473):
  restart erhält den Wert ROLLBACK[4], und die Standardlogik wird veranlasst, statt L_PRVFIELD das Event L_FIRSTFIELD zu behandeln.
- L_PRVFIELD im Startelement (vgl. Seite 473):
  Hier erfolgt eine Meldung, und die Standardlogik wird veranlasst, statt L_PRVFIELD das Event L_STAY zu behandeln.
- L_FIRSTFIELD im Startelement:
  Hier erfolgt eine Meldung, und die Standardlogik wird veranlasst, statt L_FIRSTFIELD das Event L_STAY zu behandeln.
- L_FIRSTFIELD in einem Element "hinter" dem Fetch-Element:

1. In diesem Fall darf es sich nicht um eine mehrfach geladene Maske handeln, und die Maske darf weder eingebettet noch aktiv sein.
2. Von perf() geladene Masken bleiben über den Aufruf hinweg geladen.
3. Die Maske sollte keine eigene Anwendungslogik besitzen und ihre Elemente sollten selbst keine Masken einbetten; in diesen Fällen erfolgt eine Warnung.
4. vgl. aber globale Variable S_backwards_rollback.

restart erhält den Wert ROLLBACK[1].

- L_ABORT in dem auf das Fetch-Element folgenden Element:

  restart erhält den Wert ROLLBACK, und die Standardlogik wird veranlasst, statt L_ABORT das Event L_FIRSTFIELD zu behandeln.

Vor Ausführung der Standardlogik wird nun exec ausgewertet, wobei entsprechend dem exec-Teil einer generierten Anwendungslogik (vgl. Seite 470) verfahren wird, allerdings statt einer generierten _IO-Funktion eine *FIX*-eigene Funktion benutzt wird.

Ebenfalls analog zum generierten restart-Teil (vgl. Seite 470) wird anschließend restart ausgewertet. Ggf. wird die Transaktion zurückgesetzt und die Maske sowie alle unmittelbar eingebetteten Mehrsatz-Masken geleert und reinitialisiert.

Erst danach wird gemäß der Standardlogik fortgefahren.

Behandlung des Events BT_LEFT

Die erweiterte Standardlogik von perf() bildet im Falle klicksensitiver Masken BT_LEFT auf L_STAY ab und verhindert so einen Wechsel des Feldes auf Satzebene. Der Grund hierfür ist, dass *FIX* die Beschaffung bzw. das Verwerfen der Daten an das "Überschreiten" des Fetch-Elements koppelt, bei einem wahlfreien Springen aber regelmäßig nicht entscheiden kann, ob alle zur Datensuche notwendigen Felder belegt wurden.

Allerdings ist die Anwendungslogik nicht daran gehindert, - entweder bei klicksensitiven Masken auf konventionelle Weise durch o_RetrieveItemClickedOn() oder allgemeiner mittels des in "Empfang von Maus-Events" auf Seite 159 beschriebenen Verfahrens - das angeklickte Element abzufragen und, wenn sinnvoll, dorthin zu positionieren (→ _fnr). In diesem Fall sollte sie anschließend L_STAY zurückgeben.

Um die Datenbeschaffung und Transaktionsverwaltung einer mit perf() bearbeiteten Maske zu steuern, ermöglicht *FIX* der Anwendung durch perfGetModule() den Zugriff auf die perf()-eigene _modul-Struktur.

Hinweis:

Für Entwickler, die sich näher mit den hier und in "Empfang von Maus-Events" auf Seite 159 beschriebenen Möglichkeiten beschäftigen wollen, steht im Verzeichnis `$FXDIR/src` eine Demo-Source `perfdemo.c` bereit, die in die *FIX*-Demo-Anwendung integriert werden kann.

5.2 Die allgemeine _Sql-Funktion perf_sql()

FIX beinhaltet eine Funktion, die in gewissen Fällen die generierte _Sql-Funktion ersetzen kann. Wird fx_io() mit dem Argument PERF_SQL (Makro aus `fix/fx_io.h`) anstelle von *maskenname*_Sql aufgerufen, verwendet *FIX* eine eingebaute _Sql-Funktion perf_sql(), die die von den Methoden geforderten Operationen mittels dynamischem SQL realisiert. Dynamisches SQL verwendet anstelle von Datenbank-Hostvariablen zur Übertragung von Spalten- und Parameterwerten so genannte *Descriptor Areas* (struct sqlda), die die Anzahl der bereitgestellten bzw. erwarteten Spalten und u.a. jeweils den Typ und die Adresse einer Speicherfläche für den Wert beinhalten müssen.

Der Einsatz von perf_sql() setzt voraus, dass

1. höchstens eine Tabelle zu behandeln ist, und zwar die aus der ersten Spaltenreferenz in der Beschreibungsdatei[2],
2. genau diejenigen Spalten dieser Tabelle behandelt werden sollen, zu denen Spaltenreferenzen in der Beschreibungsdatei existieren, und
3. keine Spalte mehr als einmal referenziert wird.

1. vgl. Fußnote auf Seite 496.

2. Ist keines der Felder an eine Datenbankspalte gebunden, kehren alle Nicht-Lese-Operationen sofort mit SUCCESS, Lese-Operationen mit FX_SQLNOTFOUND zurück.

Zur Ermittlung der Spaltenliste wird die Maske nach Feldern durchsucht, bei denen Bindungen aufgeführt sind. Die Tabellenkomponente der ersten Bindung legt die Tabelle für die Datenbankoperationen fest; nur Spalten dieser Tabelle werden in die Spaltenliste übernommen.

Beispiel:

field f1 … NULL
field f2 … tab1.col1
field f3 … var1
field f4 … tab2.col
field f5 … tab1.col2
field f6 … var2

→ Tabelle: tab1
Spaltenliste: col1, col2

perf_sql() hat konzeptionell etwa folgende Form:

Hinweis:
…*_STMT(Maske)* bzw. *x_CURSORNAME(Maske)* steht für generierte eindeutige Bezeichner, *COLUMNS_DESCRIPTOR(Maske)* für eine der Spaltenliste, *PARAMS_DESCRIPTOR(Maske)* für eine den Parametern im Bedingungsteil entsprechende Descriptor Area und *x* für eine jede der Cursor-Formen LOCKING_READ, FOR_UPDATE etc.

```
int perf_sql(int op)
{
    struct sqlda   *columns_area,
                   *params_area;   /* Zeiger auf Descriptor Areas für Spalten bzw.
                                      Parameter */
    int rc = 0;

    /*
     * da die Funktion für beliebige Masken funktionieren muss, aber von einer Maske
     * nichts weiß, übernimmt fx_io() die Aufgabe, einen Kontext bereitzustellen, auf
     * den perf_sql() zugreift
     */
    Kontext lesen: Maske, Tabelle, Cursorname etc.

    /*
     * bei der folgenden Darstellung ist die Fehlerbehandlung ausgeklammert; natuerlich
     * wird nach jeder SQL-Anweisung rc gesetzt und die switch-Anweisung verlassen
     */
    switch (op) {
    case SELECT_SINGLE_ROW:
        if (Maske besitzt Tabellenreferenz tabelle) {
            Anweisungstext = "select … spalte_i, … from tabelle [ where suchkriterium[1] ]
                                    into descriptor columns_area
                                    using descriptor params_area";
            PARAMS_DESCRIPTOR(Maske) und COLUMNS_DESCRIPTOR(Maske)
                aufbauen;
            columns_area = &COLUMNS_DESCRIPTOR(Maske);
            params_area = &PARAMS_DESCRIPTOR(Maske);
        Anweisung ausfuehren;
        }
```

1. Das Suchkriterium ergibt sich aus den **with**- and **where**-Angaben der Maskenbeschreibungsdatei.

```
    break;

case DECLARE_x_CURSOR:
    if (Maske besitzt Tabellenreferenz tabelle ) {
        Anweisungstext = "select … spalte_i, … from tabelle [ where suchkriterium ]
                                    [ for update ]";
EXEC SQL    prepare x_READSTMT(Maske) from Anweisungstext;
EXEC SQL    declare x_CURSORNAME(Maske) cursor for x_READSTMT(Maske);
    }
    break;

case OPEN_x_CURSOR:
    if (Maske besitzt Tabellenreferenz tabelle) {
        params_area = &PARAMS_DESCRIPTOR(Maske);
EXEC SQL    open x_CURSORNAME(Maske) using descriptor params_area;
    }
    break;

case FETCH_x_CURSOR:
    if (Maske besitzt Tabellenreferenz tabelle) {
        /* nächsten Satz lesen und Werte ablegen */
        COLUMNS_DESCRIPTOR(Maske) aufbauen;
        columns_area = &COLUMNS_DESCRIPTOR(Maske);
EXEC SQL    fetch x_CURSORNAME(Maske) using descriptor columns_area;
    }
    else
        rc = Einzelsatz-Maske? SUCCESS : FX_SQLNOTFOUND;
    break;

case PROBE_FOR_UPDATE_CURSOR:
    if (Maske besitzt Tabellenreferenz tabelle) {
        /* nächsten Satz lesen, ohne Werte abzulegen */
EXEC SQL    fetch x_CURSORNAME(Maske);
    }
    break;

case UPDATE_FOR_UPDATE_CURSOR:
    if (Maske besitzt Tabellenreferenz tabelle) {
        Anweisungstext = "update tabelle set … spalte_i = ? …
                            where current of FOR_UPDATE_CURSORNAME(Maske)";
EXEC SQL    prepare UPDATE_POSITIONED_STMT(Maske) from Anweisungstext;
        COLUMNS_DESCRIPTOR(Maske) aufbauen;
        columns_area = &COLUMNS_DESCRIPTOR(Maske);
EXEC SQL    execute UPDATE_POSITIONED_STMT(Maske)
            using descriptor columns_area;
EXEC SQL    free UPDATE_POSITIONED_STMT(Maske);
    }
    else
        rc = FX_SQLNOTFOUND;
    break;

case INSERT_BY_SINGLE_STATEMENT:
    if (Maske besitzt Tabellenreferenz tabelle) {
        Anweisungstext = "insert into tabelle ( … spalte_i, …) values (…, ?, …)";
```

```
    EXEC SQL    prepare INSERT_STMT(Maske) from Anweisungstext;
                COLUMNS_DESCRIPTOR(Maske) aufbauen;
                columns_area = &COLUMNS_DESCRIPTOR(Maske);
    EXEC SQL    execute INSERT_STMT(Maske) using descriptor columns_area;
    EXEC SQL    free INSERT_STMT(Maske);
        }
        break;

    case DELETE_ALL_BY_SINGLE_STATEMENT:
        if (Maske besitzt Tabellenreferenz tabelle) {
            Anweisungstext = "delete from tabelle [ where suchkriterium ]";[1]
    EXEC SQL    prepare DELETE_SEARCHED_STMT(Maske) from Anweisungstext;
                PARAMS_DESCRIPTOR(Maske) aufbauen;
                params_area = &PARAMS_DESCRIPTOR(Maske);
    EXEC SQL    execute DELETE_SEARCHED_TMT(Maske)
                        using descriptor params_area;
    EXEC SQL    free DELETE_SEARCHED_STMT(Maske);
        }
        break;

    case DELETE_FOR_UPDATE_CURSOR:
        if (Maske besitzt Tabellenreferenz tabelle) {
            Anweisungstext = "delete from tabelle
                                        where current of x_CURSORNAME(Maske)";
    EXEC SQL    prepare DELETE_POSITIONED_STMT(Maske) from Anweisungstext;
    EXEC SQL    execute DELETE_POSITIONED_STMT(Maske);
    EXEC SQL    free DELETE_POSITIONED_STMT(Maske);
        }
        break;

    case CLOSE_x_CURSOR:
        if (Maske besitzt Tabellenreferenz tabelle) {
    EXEC SQL    close x_CURSORNAME(Maske);
        }
        break;

    case FREE_x_CURSOR:
        if (Maske besitzt Tabellenreferenz tabelle) {
    EXEC SQL    free x_READSTMT(Maske);
        }
        break;

    default:
        return(FCT_NOT_IMPLEMENTED);
    } /* switch */

    if (rc == IOERROR && SC < 0)
        sql_error(SC, "in perf_sql()");

    return(rc);
}
```

1. vgl. Fußnote auf Seite 498.

Diese naive Form ist allerdings, was den Ressourcenverbrauch (Anzahl der prepare-ten Statements und der prepare-Operationen) anbelangt, kaum brauchbar. Betrachtet man allerdings die Art, in der fx_io() die einzelnen Operationen anstößt, näher, so macht man folgende Beobachtung:

1. Zu jeder Maske existiert zu einem Zeitpunkt nur ein Cursor (READONLY, LOCKING_READ, FOR_UPDATE).

   Daher verwendet *FIX* pro Maske nur einen Statement-Bezeichner und einen Cursornamen für alle Lese-Operationen. Folglich können nur solche cursorbezogenen Operationen angestoßen werden, die zu dem momentanen Cursor passen, und vor der Deklaration eines Cursors muss der zuletzt benutzte Cursor durch eine zu ihm konforme FREE_*x*_CURSOR-Operation freigegeben werden.

2. Beim Schreiben läuft meist eine Folge von Lese&Ersetze-Operationen gefolgt von entweder einer Folge von Einfüge-Operationen oder einer Folge von Lese&Lösche-Operationen ab.

   Anstatt also das ausgeführte Statement sofort wieder freizugeben, ist es sinnvoll, es “aufzubewahren”, da es häufig für die nächste schreibende Operation wieder verwendet werden kann (also kein erneuter Aufbau des Anweisungstextes und keine prepare-Anweisung notwendig sind). Andererseits kann man mit einem Statement-Bezeichner für alle schreibenden Operationen auskommen, da bei einem Wechsel der Operation das letzte Statement “eine Weile lang” nicht mehr gebraucht wird. perf_sql() gibt folglich nur bei einem Wechsel der schreibenden Operation (oder der Maske) das zuletzt benutzte Statement frei und bereitet das neue Statement auf.

3. Die Deskriptoren zu einer Maske - sie beinhalten Spaltenzahl sowie zu jeder Spalte Typ, Länge und Adresse (= wrkp des Maskenfeldes) - sind in der Regel konstant.

   Aus Effizienzgründen berechnet *FIX* die Descriptor Areas für jede Maske nur einmal; der Text des Suchkriteriums und der Aufbau der Descriptor Areas werden dauerhaft im Kontext gespeichert und bei Bedarf wieder verwendet.[1]

   FIX ermittelt Typ und Länge der Spalten, wenn zum ersten Mal eine Operation angestoßen wird, die den Deskriptor benötigt, d.h. die *Output Descriptor Area* für die Spaltenliste (COLUMNS_DESCRIPTOR) wird bei der ersten DECLARE_*x*_CURSOR-, UPDATE_FOR_UPDATE_CURSOR- oder INSERT_BY_SINGLE_STATEMENT-Operation ermittelt, der Text des Suchkriteriums und die *Input Descriptor Area* für dessen Parameter (PARAMS_DESCRIPTOR) bei der ersten DECLARE_*x*_CURSOR- oder DELETE_ALL_BY_SINGLE_STATEMENT-Operation.

   Ein INFORMIX-spezifisches Problem stellen SERIAL-Spalten dar: ihnen darf bei **update** und **insert** kein Wert explizit zugewiesen werden. Da aus der Maske der exakte Spaltentyp nicht hervorgeht, wird die Output Descriptor Area nicht unmittelbar aus der Felddefinition, sondern mittels **describe** der SQL-Anweisung erzeugt und erst im Nachhinein der Typ in der Descriptor Area an den der Feld-Hostvariablen, die als Speicherfläche fungiert, angepasst. So kann eine SERIAL-Spalte erkannt und gesondert behandelt werden.[2]

   Bei den Operationen UPDATE_FOR_UPDATE_CURSOR und INSERT_BY_SINGLE_STATEMENT wird eine SERIAL-Spalte, wenn die Output Descriptor Area bereits bekannt ist, erst gar nicht in die Anweisung übernommen bzw. sie wird nachträglich eliminiert, wenn die SERIAL-Spalte erst beim **describe** der Anweisung erkannt wird. Aus der Output Descriptor Area wird die SERIAL-Spalte zur Ausführung der Anweisung ausgeblendet.

   Zu beachten ist allerdings, dass im Gegensatz zu einem erfolgreichen Einfügen, bei dem perf_sql() den von INFORMIX selbständig vergebenen neuen SERIAL-Wert - dieser ist in der SQL Communication Area zu finden - in die Feld-Hostvariable übernimmt, bei einer Update-Operation nicht gewährleistet ist, dass anschließend die Werte in der Hostvariable und im Datenbanksatz übereinstimmen.

5.3 Die perf()-eigene _IO-Funktion

Sie entspricht konzeptionell einer generierten _IO-Funktion (vgl. Seite 477). Während Letzterer jedoch eine feste, zum Generierungszeitpunkt bekannte Einbettungsstruktur zugrunde liegt, wird die perf()-eigene _IO-Funktion mit beliebig

1. Da auch die Adresse gepuffert wird, darf zwischen zwei Aufrufen der perf_sql() für die gleiche Maske kein Austausch der Feld-Hostvariablen erfolgen (→ mfn_varbind()). Dies gilt nicht für mehrmalige Aufrufe von perf(): perf() verwirft am Ende alle Kontexte, die aufgebaut wurden, um die Maske mit der “erweiterten Standardlogik” zu bearbeiten.

2. Hierbei gilt allerdings, dass höchstens eine Spalte den Typ SERIAL besitzen darf, was jedoch angesichts der Tatsache, dass alle Spalten der gleichen Tabelle entstammen und eine Tabelle maximal eine SERIAL-Spalte besitzen kann, in der Praxis nur dann eine Einschränkung darstellt, wenn mehrere Felder die gleiche Bindung aufweisen.

strukturierten Masken konfrontiert, behandelt allerdings nur die Maske selbst und die *unmittelbar* in sie eingebetteten Masken.

```
switch (mode) {
case FX_READ :
    /* Zugriffsart Lesen */
    switch (fx_io(mode, STANDARD, maske, PERF_SQL, NO_COPY, NO_USR_DATA)) {
    case SUCCESS :
        ret = FX_UPDATE;
        for alle in maske eingebetteten Mehrsatz-Masken elem while ret == FX_UPDATE
            if (fx_io(mode, STANDARD, elem, PERF_SQL, NO_COPY,
                                                        NO_USR_DATA) < 0)
                ret = IOERROR;
        break;
    case FX_SQLNOTFOUND :
        ret = FX_INSERT;
        break;
    case FX_SQLLOCKED :
        rt_msg(ACHTUNG, fxlsystxt(M_locked));
        ret = IOERROR;
        break;
    default :
        ret = IOERROR;
        break;
    }
    break;
case FX_INSERT :
    /* Zugriffsart Einfuegen */
case FX_UPDATE :
    /* Zugriffsart Korrigieren */
case FX_DELETE :
    /* Zugriffsart Loeschen */
    ret = 0;
    if (fx_io(mode, STANDARD, maske, PERF_SQL, NO_COPY, NO_USR_DATA) < 0)
        ret = IOERROR;
    for alle in maske eingebetteten Mehrsatz-Masken elem while ret == 0
        if (fx_io(mode, STANDARD, elem, PERF_SQL, NO_COPY, NO_USR_DATA) < 0)
            ret = IOERROR;
    break;
}
return(ret);
```

Hinweis:

Wird ein *FIX*-Programm mit dem Schalter -test 2 gestartet, so werden

- der Aufbau und Abbau der Kontexte,
- die Aufrufe von perf_sql(),
- die abgesetzten SQL-Anweisungen und der danach in SQLCODE vorgefundene Wert, und
- die angelegten Descriptor Areas

nach stderr protokolliert.

47 Source-Generierung für Menüs

Nicht nur zu Masken, sondern auch zu Menüs kann Sourcecode generiert werden. Ausgangspunkt ist eine Menübeschreibungsdatei. Bei der Generierung berücksichtigt *FIX* auch alle in **perform**-Aktionen angesprochenen Untermenüs beliebiger Stufe.

Beispiel:

```
menue AUFTRMEN …
function AUFTR_control 17 30 5 44 …
AUFTR_1 6 3 0 "Einfuehrung"          call entry_help
AUFTR_2 8 3 0 "Auftragsprogramm"     call AUFTb_proc
AUFTR_3 10 3 0 "wie generiert"       call AUFTR_proc
AUFTR_4 12 3 0 "Sourcen dazu"        perform men/sources.men
AUFTR_5 14 3 0 "die DB-Tabellen"     shclear "less sql/*"
END_OF_MFO
```

1 Struktur der Source

Generiert wird eine Datei *source*`.c`, wobei *source* durch den Entwickler festgelegt wird.

Die Datei enthält folgende Komponenten:

- die Definition der Konstanten FIXRELEASE, die anzeigt, mit welcher *FIX*-Version die Source erstellt wurde,
- Einschlüsse von C-Header-Dateien und der *FIX*-Header-Datei `fix/fix.h`,
- einen Deklarationsteil, der extern- und static-Vereinbarungen von Funktionen beinhaltet,
- eine Funktion dcl_usr_fct(),
- Funktionen, die als Vorlage für die Anwendungslogik dienen (nur, wenn das Menü oder eines seiner Untermenüs eine **function**-Angabe enthält),
- eine Modul-Prozedur *menüname*_proc().

1.1 Der Deklarationsteil

Für jeden der in einer **call**-Aktion auftauchenden Funktionsbezeichner wird die extern-Vereinbarung einer Funktion gleichen Namens generiert. Dabei nimmt *FIX* an, dass die Funktion ein Resultat vom Typ int hat und keine Parameter besitzt.[1] Wenn die Menübeschreibung eine **function**-Angabe enthält, wird dafür die static-Vereinbarung einer gleich lautenden Funktion erzeugt.

1. Wenn die Funktion kein Resultat vom Typ int hat, muss die Vereinbarung im Nachhinein angepasst werden. Im Beispiel ist dies bei der *FIX*-Funktion entry_help() der Fall.

Beispiel:

```
static Event    AUFTR_control(obj *, Event);
extern int      entry_help(void); [1]
extern int      AUFTb_proc(void);
extern int      AUFTR_proc(void);
```

1.2 Die Funktion dcl_usr_fct()

Die im Deklarationsteil vereinbarten Funktionen müssen, damit *FIX* sie aufrufen kann, im Programm durch Aufrufe von addfct() mit den in den Beschreibungsdateien verwendeten Bezeichnern assoziiert werden. Diese Aufgabe übernimmt die Funktion dcl_usr_fct(), die in der Modulprozedur aufgerufen wird.

Beispiel:

```
static void dcl_usr_fct(void)
{
        addfct("AUFTR_control", (int (*)())AUFTR_control);
        addfct("entry_help", (int (*)())entry_help);
        addfct("AUFTb_proc", (int (*)())AUFTb_proc);
        addfct("AUFTR_proc", (int (*)())AUFTR_proc);
}
```

1.3 Anwendungslogik

Zu der ersten im Verlauf der Generierung vorgefundenen **function**-Angabe wird eine Muster-Anwendungslogik erzeugt, die zwischen allen wesentlichen Events differenziert.

Beispiel:

```
static Event AUFTR_control(obj *objp, Event event)
{
    menue *menp = (menue *)objp;
    int item_nr;
    Event rc;

    item_nr = CURRENTMENITEM(menp);
    rc = event;

    switch (event) {
    case L_LOAD_OBJECT :
        break;
    case L_FREE_OBJECT :
        break;

    case L_ENTER_OBJECT :
        break;
    case L_QUIT_OBJECT :
        break;
    case L_LEAVE_OBJECT :
        break;
    case L_ENTER_ENTRY :
        break;
```

```
        case L_LEAVE_ENTRY :
            break;
        case L_BEFORE_EXECUTION :
            break;
        case L_AFTER_EXECUTION :
            break;

        case K_kr :
            break;
        case K_kl :
            break;
        case K_ku :
            break;
        case K_kd :
            break;
        case K_kh :
            break;
        case K_PD :
            break;
        case K_PU :
            break;
        case K_HP :
            break;
        case K_ST :
            break;
        }
        return(rc);
    }
```

Für **function**-Angaben weiterer Untermenüs würde eine kompaktere Form erzeugt:

```
    static Event bezeichner(obj *objp, Event event)
    {
        menue *menp = (menue *)objp;
        int item_nr;
        Event rc;

        item_nr = CURRENTMENITEM(menp);
        rc = event;
        /*
         *  Verarbeitungslogik
         */
        return(rc);
    }
```

1.4 Die Modul-Prozedur *menüname*_proc()

Als letzte Komponente wird eine Funktion *menüname*_proc() generiert, in der das Menü geladen und ausgeführt wird.

Beispiel:

```
    int AUFTRMEN_proc(void)
    {
        static char *objname = "men/auftr.men";
```

```
    static obj *objp = (obj *)0;
    static BOOLEAN init_done = FALSE;
    int rc = 0;

    if (! init_done) {
        /* beim ersten Aufruf mache FIX die im Menue benannten Funktionen bekannt */
        dcl_usr_fct();
         init_done = TRUE;
    }

    if (objp != 0 || (objp = (obj *)loadmenue(objname)) != 0) {
        rc = (int)perform(objp, NO_EVENT_CONTROL);
#ifdef DELOBJ
        o_free(objp);
        objp = (obj *)0;
#endif /* DELOBJ */
    }
    else
        rt_msg(FEHLER, fxsystxt(M_objnotfound), objname);
    return(rc);
}
```

1.5 Das Programm

Auch hier wird die Funktion main() aus `main.c` mit eingebunden (vgl. Seite 467). Im Makefile müssen ggf. weitere Abhängigkeiten und zusätzlich einzubindende Dateien von Hand ergänzt werden:

```
source:             $(BIN)/source
                    @echo $@ done

$(BIN)/source:      $(OBJS)/source.o
                    ( cd $(OBJS); \
                    $(CC) $(CFLAGS) -DPROC="menüname_proc" \
                            $(FXDIR)/src/main.c source.o $(LIBS) -o $@; \
                    rm -f main.o )

$(OBJS)/source.o:   source.c
                    ( cd $(OBJS); $(CC) -c $(CFLAGS) $(SRC)/source.c )
```

Die Dateiendungen von Objektdatei und Binärprogramm können mittels der Ressource [fgmenu].TargetFileExtensions beeinflusst werden (vgl. Seite 450).

2 Aufruf der Generierung aus der Shell

Zur Generierung von Menü-Sourcecode dient das Programm **fgmenu**, das als Argumente die Menübeschreibungsdatei und wahlweise den Basisnamen für die Zieldatei erhält:

fgmenu [-m] <beschreibungsdatei> [<basisname>] [-]

Der Schalter bedeutet:

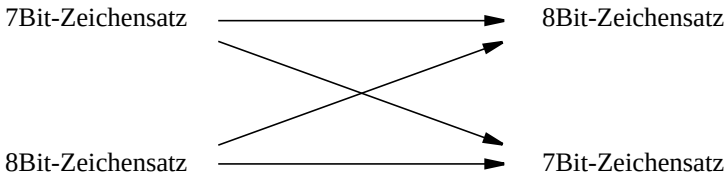
m zusätzlich Regel im Makefile generieren

Wird als letztes Argument ‘-’ angegeben, erfolgt die Ausgabe nach stdout. Wird kein Basisname angegeben, wird der Menüname als Basisname übernommen.

Relative Dateipfade werden bzgl. `$FXHOME` aufgelöst.

48 Unterstützung von 8Bit-Zeichen in Dateien

Im Gegensatz zu kundenspezifischen Anwendungsprogrammen, die sich an dem für die jeweilige Plattform typischen Zeichensatz orientieren, muss ein Entwicklungswerkzeug wie *FIX* die Übertragbarkeit der mit ihm erstellten Applikationen auf andere Plattformen gewährleisten. Folgende Konstellationen sind möglich:



Damit auch auf solchen Plattformen, die nicht über 8Bit-fähige Utilities (z.B. Texteditoren) verfügen, *FIX*-Anwendungen erstellt werden können, erlaubt *FIX* beim Lesen und Schreiben von

- Objektbeschreibungsdateien (.men, .mfo, .sel, .cho),
- Datendateien für Choices (.dat),
- Hilfetexten
- der Rohfassung der Meldungsdatei (`fx_texte`)

die Angabe von Zeichen auch in oktaler bzw. hexadezimaler *Ersatzdarstellung*:

\ooo (ein bis drei Oktalziffern mit einem Dezimalwert zwischen 0 und 255) bzw.
\xhh (ein oder zwei Hexadezimalziffern mit einem Dezimalwert zwischen 0 und 255)

Bei Plattformen, die 8Bit-fähig sind, kann selbstverständlich auch das Zeichen selbst benutzt werden.

Für *darstellbare* Zeichen mit Codes kleiner-gleich 127 (Makro isprint() aus `ctype.h`) sollte die Ersatzdarstellung allerdings vermieden werden, insbesondere in solchen Zeichensequenzen, die Symbole trennen (Leerzeichen, Tabulatoren, Zeilenvorschübe), begrenzen (Anführungszeichen) oder die Funktion eines Schlüsselwortes haben (Umschaltecodes für Videoattribute, Seitentrenner in Hilfetexten usw.). Unzulässig ist die Ersatzdarstellung bei Verwendung des Zeichens als Fluchtsymbol (‘\’, ‘^’).

Auf Plattformen, die nicht 8Bit-fähig sind, führen nicht in Ersatzdarstellung angegebene Zeichen oberhalb von 127 beim Lesen zu einer Warnung bzw. werden beim Schreiben (**led**, Hardcopy-Datei, Kommandoausführung) in Ersatzdarstellung ausgegeben. Beides kann unterdrückt werden durch Angabe des Programmschalters -8. Umgekehrt kann durch Angabe des Schalter -7 die Verwendung der Ersatzdarstellung beim Schreiben von Dateien erzwungen werden.[1]

Nicht darstellbare Zeichen zwischen 0 und 127 (Makro iscontrol() aus `ctype.h`) werden in jedem Fall in oktaler Ersatzdarstellung ausgegeben.

1. **fxm** vererbt diese Schalter an von ihm gestartete *FIX*-Programme.

49 Terminal-Anpassung

1 Bildschirm

Zur Bildschirmansteuerung greift *FIX* auf Beschreibungen im Verzeichnis `$FXDIR/etc/fxtermcap` zurück. Dort wird eine Datei im *termcap*-Format (siehe UNIX Manual) erwartet, deren Namen mit dem Wert der Umgebungsvariablen FXTERM (ersatzweise TERM) übereinstimmt.[1,2]

Bei der Auslieferung von *FIX* werden Anpassungen an gängige Terminals mitgeliefert. Sollte Ihr Terminal nicht dabei sein, schauen Sie zunächst, ob vielleicht eine Anpassung an ein dem Ihren ähnliches Terminal enthalten ist. Sollte das nicht der Fall sein, erfahren Sie unten, wie Sie eine eigene Beschreibungsdatei erstellen können.

Analog zur Umgebungsvariablen TERMCAP kann für *FIX* die Umgebungsvariable FXTERMCAP gesetzt werden.

Die Bildschirmbeschreibung für *FIX* enthält zum einen Einträge, die unmittelbar aus `/etc/termcap` übernommen (oder aus terminfo abgeleitet) werden können; dies sind:

li	Anzahl Zeilen (ohne Statuszeile !)	
co	Anzahl Spalten	
bs	^H ist backspace	
cl	Bildschirm löschen	(clear screen and home cursor)
ce	Löschen bis Zeilenende	(clear to end of line)
cm	Cursor absolut positionieren	(cursor motion)
nd	Cursor nach rechts	(non destructive space)
up	Cursor eine Zeile nach oben	(upline)
so	Invers ein	(start standout mode)
us	Unterstreichen ein	(start underscore mode)
se	alle Attribute aus	
eA	alternativen Zeichensatz laden	

“se” wird nicht exakt in der üblichen Bedeutung verwendet. Übernehmen Sie ggf. die Sequenz für “me” (turn off attributes) aus `/etc/termcap` oder kombinieren Sie die dort angegebenen Sequenzen für “so” (end standout mode) und “ue” (end underscore mode).

Daneben gibt es eine Reihe speziell für *FIX* verwendeter *Capabilities*, für die geeignete Steuersequenzen angegeben werden müssen. Aus Kompatibilitätsgründen mit früheren *FIX*-Versionen stimmen einige Namen leider nicht mit den üblichen Capability-Namen von termcap überein (sofern vorhanden, sind Letztere in [ ] angegeben). Mit Ausnahme von GS und GE sind alle Einträge optional, d.h. brauchen nur angegeben zu werden, wenn sie vom Default abweichen:

is	Sequenz zur Terminal-Initialisierung	[ti]	Default: ""
ls	Halbhell ein	[mh]	Default: ""
bl	Blinken (+invers) ein	[mb]	Default: wie “so”)

1. Die Benutzung TERM-bezogener Dateien anstelle einer Gesamtdatei dient ausschließlich dazu, die Distribution zu vereinfachen. Das Dateiformat entspricht exakt dem von `/etc/termcap`.

2. Ist die Umgebungsvariable FXTERMPATH gesetzt, wird `fxtermcap/$FXTERM` statt in `$FXDIR/etc` in einem der in $FXTERMPATH aufgeführten Verzeichnisse erwartet.

GS	Semigrafik-Zeichensatz ein	[as]	
GE	Semigrafik-Zeichensatz aus	[ae]	
xa	Semigrafik beschädigt Attribut [1]		Default: nicht gesetzt
cf	Cursor unsichtbar	[vi]	Default: ""
cn	Cursor in Normaldarstellung	[ve]	Default: ""
ep	Sequenz zum Betreten der Statuszeile	[ts]	Default: ""
lp	Sequenz zum Verlassen der Statuszeile	[fs]	Default: ""
sl	Sequenz zum Positionieren in der Statuszeile		
	gewöhnlich identisch mit cm	[ts]	Default: "" (nicht möglich)
3D	siehe Seite 512		

Capabilities können mit **tcout** und **tgoto** getestet werden.

Semigrafik

Die gerätespezifische Realisierung der Semigrafik (vgl. Kapitel *41*) wird ebenfalls in der Terminalbeschreibung hinterlegt. Der Eintrag

SG	Übersetzungstabelle für *FIX*-Grafikzeichen

bestimmt, wie ein Grafik-Code ausgegeben wird, z.B. bedeutet "SG=…91s…[0[…", dass für die obere rechte Ecke (9) das Zeichen mit Code 's' aus dem (mit GS/GE umgeschalteten) Semigrafik-Zeichensatz (1) und für den linken Feldbegrenzer ([) das Zeichen '[' aus dem Standard-Zeichensatz (0) auszugeben ist. Ist der GS-Eintrag leer oder fehlt der SG-Eintrag oder ist leer, werden die Defaultwerte verwendet (siehe Seite 271).

Für nicht in SG definierte Codes gibt *FIX* üblicherweise das entsprechende *geräteabhängige* Semigrafikzeichen aus; wenn Sie stattdessen ein Leerzeichen bevorzugen, so beginnen Sie die Definition von SG mit der Zeichenfolge \033.

Zu beachten ist auch, dass *FIX* aus Gründen der Kompatibilität bei Terminals

- für die äußeren Linienelemente i. Allg. die Definition für das entsprechende innere Linienelement übernimmt. Durch Setzen der booleschen Capability *3D* kann die Identifizierung der äußeren mit den inneren Zeichen jedoch unterdrückt werden: die Zeichen werden dann gemäß der Definition in *SG* bzw. geräteabhängig ausgegeben. Die Angabe von 3D kann zweckmäßig sein bei Terminals, die mehr als einen Linientyp unterstützen.
- für die Scroll-Begrenzer '<' und '>' i. Allg. die Definition für die Standard-Begrenzer '[' und ']' übernimmt. Durch Setzen der booleschen Capability *3D* kann die Identifizierung der Scroll- mit den Standard-Feldbegrenzern jedoch unterdrückt werden: die Zeichen '<' und '>' werden dann gemäß der Definition in *SG* bzw. geräteabhängig ausgegeben.

Prinzipiell kann mittels SG jedem Code zwischen ' ' und '~' ein alternatives Zeichen zugewiesen werden, also auch solchen, für die *FIX* keine Bedeutung vorgibt. Da der Umfang der Semigrafik jedoch von Terminal zu Terminal stark differieren kann, ist eine mit viel Semigrafik versehene Anwendung schlecht zu portieren.

Das Kommando **edtc** ruft $EDITOR mit der gültigen Bildschirmbeschreibung als Argument auf.

1. Ist die boolesche Capability xa definiert, wird nach jeder Umschaltung zwischen normalem und Semigrafikzeichensatz die Steuersequenz zum Setzen des aktuellen Videoattributs wiederholt. Diese Funktionalität wird benötigt bei Bildschirmen, bei denen GS und GE zugleich das mittels so, us, bl oder ue eingestellte Videoattribut zurücksetzen.

2 Tastatur

2.1 Tastenbeschreibung

FIX-Programme erwarten im Verzeichnis `$FXDIR/etc/fxkeycap` eine Datei mit Namen `$FXKEYBOARD` (ersatzweise `$TERM`)[1], die eine Abbildung der Sondertasten von *FIX* auf Tasten(folgen) enthält, die an Ihrem Terminal diesen Tasten entsprechen, sowie Tastenbeschriftungen, die zur gerätespezifischen Wiedergabe von Tasten in Texten dienen.

2.2 Defaultwerte für Tasten

Wenn die Tastenbeschreibungsdatei keinen oder einen leeren Eintrag für eine Taste enthält, wird ihr Defaultwert aktiv. Mit Ausnahme einiger weniger Tasten, die unten aufgeführt sind, folgen Defaultwerte folgender Regel:

> Defaultwert für die Taste Xy ist ESC X y

Ist in der Tastenbeschreibungsdatei die Pseudotaste EC spezifiziert, wird statt ESC diese Taste genommen.

Die Ausnahmen sind:

EN	^D
RT	^M
DL	^H
TB	^I
sh	INTERRUPT

Keinen Defaultwert besitzen die Tasten LF, h1, h2, h3, h4, h5, h6, h7, h8, h9, h0.

2.3 Die Utility keyinit

Zur Erstellung von Tastenbeschreibungen für nicht unterstützte Terminals enthält das *FIX*-Runtime-System die Utility **keyinit**. Dieses Programm erzeugt eine Tastenbeschreibung `$FXTERM` (ersatzweise `$TERM`) im Verzeichnis `$FXDIR/etc/fxkeycap`. Ehe Sie **keyinit** aufrufen, sollten Sie sich mit den *FIX*-Tasten vertraut gemacht und sich überlegt haben, welche Taste(nfolge) Sie welcher “Funktion” zuordnen wollen. Die von der Tastenfolge produzierte Zeichenkette darf nicht mit einem darstellbaren Zeichen beginnen und keine Taste(nfolge) darf Beginn einer anderen sein (also nicht ‘Esc’ ‘f’ und ‘Esc’ ‘f’ ‘1’).

Wenn **keyinit** nach einer Taste fragt, drücken Sie einfach die gewünschte Taste(nfolge) und anschließend ‘Return’ (^M) oder aber ‘-’, wenn die Taste nicht verfügbar sein soll. Wenn Sie nur ‘Return’ drücken, wird der Defaultwert benutzt.

Wollen Sie einer *FIX*-Taste mehrere Tasten(folgen) zuordnen, schließen Sie die Tasteneingabe statt mit ‘Return’ mit ‘LineFeed’ (^J) ab; **keyinit** fragt dann nach einer alternativen Taste(nfolge).

Da ^M und ^J eine Tasteneingabe abschließen, müssen Sie eine Taste(nfolge), die diese Zeichen enthält, - leider senden die Funktionstasten einiger Terminals solche Sequenzen - bei der Eingabe vollständig in Begrenzer einschließen. Als Begrenzer können Sie jedes Zeichen zwischen ‘ ’ und ‘~’ verwenden, das in der Taste(nfolge) nicht vorkommt.

1. Ist die Umgebungsvariable FXTERMPATH gesetzt, wird `fxkeycap/$FXKEYBOARD` statt in `$FXDIR/etc` in einem der in $FXTERMPATH aufgeführten Verzeichnisse erwartet.

Beispiel:

Die mit F7 markierte Taste sendet ‘E[m^M’; Sie geben ein: / F7 / Return

Nicht verwenden dürfen Sie die INTERRUPT-Taste (DEL, ^C), da Sie **keyinit** damit abbrechen.

Nach Eingabe der Taste(nfolge) können Sie eine Beschriftung für die Taste angeben, einen bis zu 5 Zeichen langen Text, der die Taste beschreibt (z.B. “ShF1” für “Shift Taste + mit F1 markierte Funktionstaste”). Innerhalb des Textes können Sie die üblichen Umschaltcodes für Videoattribut und Zeichensatz verwenden; diese gehen nicht in die Textlänge ein.

2.4 Format der Tastenbeschreibungsdatei

Jede Zeile definiert eine Taste. Folgt unmittelbar auf die Tastendefinition ein ‘#’, wird der Rest der Zeile als Kommentar angesehen.

Beispiel:

hc=:	Defaultwert (ESC h c) übernehmen
h9=none:	Taste nicht verfügbar
DL=^H:"Bcksp"	Taste mit entsprechender Beschriftung
PL=E9:E]:"^Nlya^O"# Page-Left	alternative Taste, Semigrafik-Label

Das Kommando **edkc** ruft $EDITOR mit der gültigen Tastenbeschreibung als Argument auf.

Unbenannte Events

Links vom Gleichheitszeichen können neben den vordefinierten Tastenbezeichnungen wie HP, RT auch Eventcodes verwendet werden:

```
711=\Exyz
```

Erkennt *FIX* die rechts vom Gleichheitszeichen angegebene Zeichenfolge, wird hierfür das *unbenannte Event* (Event)711 generiert.

Um Konflikte mit künftigen *FIX*-Erweiterungen zu vermeiden, sollten sich Entwickler, die dieses Feature für ihre Anwendung nutzen wollen, auf Codes von 700 bis 999 beschränken.

Obwohl technisch möglich, ist die Angabe einer Tastenbeschriftung unsinnig, da auf Grund der fehlenden Bezeichnung eine Einbettung in Hilfetexte oder Meldungen analog zu #HP nicht in Frage kommt.

3 Die Umsetzung von Zeichen bei der Ein- und Ausgabe

3.1 Ausgabe

Der Code, den ein Zeichen innerhalb des verwendeten Zeichensatzes belegt, muss nicht zwangsläufig mit dem übereinstimmen, der die Darstellung des Zeichens am Bildschirm bewirkt. Unter Umständen erfordert die Wiedergabe eines Zeichens aus dem Zeichensatz sogar das Senden einer Codesequenz.

FIX erlaubt daher, zu den Zeichen des Zeichensatzes - getrennt nach Geräten - Codesequenzen anzugeben, die zur Ausgabe des Zeichens am Bildschirm verwendet werden sollen. Hierzu dienen die Dateien `$FXTERM` im Unterverzeichnis `fxcharmap` des Zeichensatz-Verzeichnisses `$FXCHARSET`.

Existiert keine entsprechende Datei[1], findet keine Umsetzung statt; bei einer korrupten Datei erfolgt ein Programmabbruch.

Die Datei `fxcharmap/$FXTERM` muss aus Zeilen der Form

<zeichenbenennung> = <codesequenz>

bestehen, wobei

<zeichenbenennung>	::=	Zeichenbenennung aus der Datei `charset` im Verzeichnis `$FXCHARSET`
<codesequenz>	::=	maximal 31 8Bit-Zeichen (für Ersatzdarstellungen gelten die gleichen Regeln wie bei termcap-Dateien)

Daran anschließen kann sich ein mit # eingeleiteter Kommentar, der sich dann bis zum Ende der Zeile erstreckt (auch reine Kommentarzeilen sind möglich).

Zeichen aus den Klassen CONTROL und SPACE bzw. BLANK sowie die Zeichen '^', '\', '#' müssen in Ersatzdarstellung angegeben werden (termcap-Format).

Beispiel:

Beim Terminal 97801 von Fujitsu Siemens wird mit der Sequenz "\E(K" der deutsche, mit "\E)K" der US-Zeichensatz geladen. Enthält `charset` z.B. die Definition

Ae = 0x5B, … # A dieresis

so würde folgender Eintrag in der Umsetzungsdatei die Ausgabe von 'Ä' bewirken:

Ae=\E(K[\E)K

3.2 Eingabe

Der Code, der von der mit einem bestimmten Zeichen beschrifteten Taste der Tastatur gesendet wird, muss nicht zwangsläufig derselbe sein wie der, den das Zeichen im Zeichensatz der Anwendung besitzt. Unter Umständen kann ein Zeichen aus dem Zeichensatz sogar nur durch Kombination mehrerer Tasten produziert werden.

FIX erlaubt daher, zu den Zeichen des Zeichensatzes - getrennt nach Geräten - Codesequenzen anzugeben, die bei Lesen von einem Gerät in dieses Zeichen übersetzt werden sollen. Hierzu dienen die Dateien `$FXTERM` im Unterverzeichnis `fxkeymap` des Zeichensatz-Verzeichnisses `$FXCHARSET`.

Existiert keine entsprechende Datei[2], findet keine Umsetzung statt; bei einer korrupten Datei erfolgt ein Programmabbruch.

1. Wird das *FIX*-Programm mit dem Schalter -test 16384 gestartet, erfolgt in diesem Fall eine Warnung.
2. Wird das *FIX*-Programm mit dem Schalter -test 16384 gestartet, erfolgt in diesem Fall eine Warnung.

Die Datei `fxkeymap/$FXTERM` muss aus Zeilen der Form

<zeichenbenennung>=<codesequenz>

bestehen, wobei

<zeichenbenennung>	::=	Zeichenbenennung aus der Datei `charset` im Verzeichnis `$FXCHARSET`
<codesequenz>	::=	maximal 31 8Bit-Zeichen (für Ersatzdarstellungen gelten die gleichen Regeln wie bei termcap-Dateien)

Daran anschließen kann sich ein mit # eingeleiteter Kommentar, der sich dann bis zum Ende der Zeile erstreckt (auch reine Kommentarzeilen sind möglich).

Die Zeichenbenennungen müssen paarweise verschieden sein. Umfasst <codesequenz> mehrere Zeichen, muss der Code des ersten Zeichens aus der Klasse CONTROL sein. Zeichen aus den Klassen CONTROL und SPACE bzw. BLANK sowie die Zeichen '^', '\', '#' müssen in Ersatzdarstellung angegeben werden (termcap-Format).

Beispiel:

Enthält `charset` die Definition

```
Ae = 0x8E, … # A dieresis
```

und soll die Taste { das Zeichen 'Ä' produzieren, so würde dies durch folgenden Eintrag in der Umsetzungsdatei bewirkt:

```
Ae={
```

Durch die Angabe

```
@{=^V{
```

wäre die Eingabe des Zeichens '{' durch die Tastenfolge "Control+V {" möglich.

50 Lizenzkonzept

1 Lizenzprüfung durch *FIX*

Das Lizenzkonzept beruht darauf, dass beim Start eines *FIX*-Programms ein Server-Prozess - der *Lizenz-Server* **liserver** - angesprochen wird und auch laufende *FIX*-Programme diesem Server während des Ablaufs sporadisch ihre Existenz melden. Der Server-Prozess wird, sofern er nicht bereits läuft, automatisch vom *FIX*-Programm gestartet.

Der Lizenz-Server prüft zunächst, ob der Entwickler/Anwender autorisiert ist, *FIX*-Programme zu starten. Diese Prüfung geschieht anhand der Datei, die in der Umgebungsvariable FXINSTFILE hinterlegt ist, sonst anhand der Datei `$FXDIRSYS/spec/fxinstfile`. Die Daten in dieser Datei werden bei der Registrierung (vgl. Abschnitt 5) erstellt.

Sollte die Datei nicht konsistent oder, bei einer befristeten Lizenz, das darin spezifizierte Zeitlimit überschritten sein, so wird dennoch versucht, den Lizenz-Server zu starten. Hierzu werden die Seriennummer und der Aktivierungsschlüssel aus der Datei `$FXDIRSYS/spec/fxserial` ausgelesen. Passen beide zueinander, wird der Lizenz-Server gestartet. In diesem Falle sollte *FIX* jedoch möglichst bald (erneut) registriert werden.

Der Lizenz-Server prüft weiter, ob die Anzahl der angemeldeten Terminals (oder PCs mit *FIX/Win*) die zulässige Höchstgrenze überschreitet (vgl. aber “Prozessbezogene Lizenzierung” auf Seite 518). Ist dies der Fall, kontrolliert der Server, ob angemeldete Programme zwischenzeitlich terminiert haben und somit die Anzahl der angemeldeten Arbeitsplätze doch unterhalb der Höchstgrenze liegt. Pro Terminal (oder PC mit *FIX/Win*) können maximal sechs *FIX*-Prozesse zur gleichen Zeit genutzt werden (vgl. aber *Beschränkung der Prozesszahl pro Arbeitsplatz*).

Ein laufender Server-Prozess kann mit der Utility **likill** beendet werden. Da bei einem von ihm lizenzierten Programm dann allerdings nach einiger Zeit ein automatischer Abbruch erfolgt, sollten zum Zeitpunkt des Aufrufs keine Anwendungen aktiv sein.

Die Art der Identifizierung eines Systems ist abhängig vom Betriebssystem.

2 Dämon-Lizenzen

Der Lizenz-Server kann so konfiguriert werden, dass er bei der Lizenzzuteilung zwischen interaktiven und Hintergrundprozessen unterscheidet.

FIX sieht einen Prozess als Dämon an, wenn stdin zum Zeitpunkt des Aufrufs von fxinit() kein gültiger Filedeskriptor ist und der Prozess nicht mit dem Schalter -service gestartet worden ist. Bei entsprechender Konfiguration des Lizenz-Servers erhält ein solcher Prozess eine besondere *Dämon-Lizenz*. Sämtliche Prozesse mit einer Dämon-Lizenz werden als *ein* Arbeitsplatz (User) gezählt, der beliebig viele Prozesse (also mehr als die üblichen 6 pro Arbeitsplatz) gleichzeitig nutzen kann.

Achtung

> Ein Prozess, dem vom Lizenz-Server eine Dämon-Lizenz zugeteilt worden ist, initialisiert weder die Window-Verwaltung noch die Datenstrukturen zur Tastenerkennung, Bildschirmansteuerung oder Signalbehandlung, so dass die Verwendung darauf beruhender Funktionen im Allgemeinen zu einem Programmabbruch mit Core Dump führt.

Zur Konfiguration muss auf der zehnten oder einer folgenden Zeile der Lizenzdatei folgende Angabe stehen:

DaemonLicenses : *on*

Fehlt eine solche Angabe oder steht dort statt “on” das Symbol “off”, so wird nicht zwischen interaktiven und Hintergrundprozessen unterschieden.

Zur Zeit kann der Wert nicht mittels **licensefix** verändert werden; eine vorhandene Angabe wird bei der Registrierung jedoch bewahrt.

3 Beschränkung der Prozesszahl pro Arbeitsplatz

Um die Prozesszahl pro Arbeitsplatz (User) herabzusetzen, muss die zehnte oder eine der folgenden Zeilen der Lizenzdatei folgende Angabe enthalten:

ProcessesPerLicense: *n*

Fehlt eine solche Angabe oder ist der Wert *n* größer als 6, so lässt der Lizenz-Server höchstens 6 (interaktive) Prozesse pro Arbeitsplatz zu. Unabhängig von *n* wird pro Arbeitsplatz mindestens ein Prozess ermöglicht.

Die Zählung von Dämon-Prozessen erfolgt wie in Abschnitt 2 beschrieben.

Zur Zeit kann der Wert nicht mittels **licensefix** verändert werden; eine vorhandene Angabe wird bei der Registrierung jedoch bewahrt.

Hinweis:

Wird Prozess-Lizenzierung (vgl. Abschnitt 4.1 und Abschnitt 4.2) benutzt, so ist die Angabe ohne Belang.

4 Alternative Lizenzierungsverfahren

Der Lizenz-Server kann so konfiguriert werden, dass er bei der Lizenzzuteilung nicht Arbeitsplätze (User), sondern Prozesse zählt.

Bei Prozess(gruppen)bezogener Lizenzierung sind die Angabe

ProcessesPerLicense : *n*

bzw. die eingebaute Begrenzung der Prozesse pro Arbeitsplatz ohne Bedeutung. Jeder Arbeitsplatz kann alle noch verfügbaren Lizenzen vereinnahmen.

Die Zählung von Dämon-Prozessen erfolgt jedoch auch dann wie in Abschnitt 2 beschrieben.

4.1 Prozessbezogene Lizenzierung

Auf der zehnten oder einer folgenden Zeile der Lizenzdatei muss dazu folgende Angabe stehen:

PerProcessLicensing : *on*

Fehlt eine solche Angabe oder steht dort statt “on” das Symbol “off”, so werden wie einführend beschrieben Arbeitsplätze (mit bis zu maximal 6 Prozessen pro Arbeitsplatz) gezählt.

Zur Zeit kann der Wert nicht mittels **licensefix** verändert werden; eine vorhandene Angabe wird bei der Registrierung jedoch bewahrt.

4.2 Prozessgruppenbezogene Lizenzierung

Der Lizenz-Server kann so konfiguriert werden, dass bei der Lizenzzuteilung *an Prozesse* Prozesse mit gleicher Prozessgruppe nur eine Lizenz verbrauchen.

Prozess-Lizenzierung (vgl. Abschnitt 4.1) muss aktiviert sein. Zusätzlich muss auf der zehnten oder einer folgenden Zeile der Lizenzdatei folgende Angabe stehen:

MergeProcessGroup: *on*

Fehlt eine solche Angabe oder steht dort statt “on” das Symbol “off”, so bleibt die Prozessgruppe des anfordernden Prozesses unberücksichtigt.

Zur Zeit kann der Wert nicht mittels **licensefix** verändert werden; eine vorhandene Angabe wird bei der Registrierung jedoch bewahrt.

5 Registrierung

Mit dem Script **licensefix** kann der Erwerber von *FIX* die Angaben, die zur Registrierung von *FIX* notwendig sind, selbst erfassen und nach Erhalt des zugehörigen Schlüssels durch den Lieferanten die endgültigen Lizenzdaten speichern. Die Registrierung sollte möglichst bald nach der Installation vorgenommen werden.

6 Anzeigen der vergebenen Lizenzen

Ein Aufruf der Utility

fxlishow

zeigt die zu diesem Zeitpunkt in Gebrauch befindlichen Lizenzen:

1 of 10 licenses in use.
Arbeitsplatz PID
…

wobei *Arbeitsplatz* ein TTY oder eine IP-Adresse (*FIX/Win*, *FIX/Web*) ist.

fxlishow verwendet zum Datenaustausch mit dem Lizenz-Server **liserver** temporär die Datei `/tmp/lishowdata`. Durch Angabe eines absoluten Pfades als Argument kann die Dateieinstellung übersteuert werden.

Hinweis:

fxlishow sollte mit s-Bit unter der gleichen UID wie der Lizenzserver laufen, da der Lizenzserver die temporäre Datei nur dann beschreibt, wenn sie (a) leer ist und (b) entweder seine effektive UID oder seine effektive GID mit dem Eigentümer bzw. der Gruppe der Datei übereinstimmt.

Anhang A Internationalisierung

FIX erlaubt die Berücksichtigung kultureller Gepflogenheiten bei der Darstellung von Zahlen, Datum- und Wahrheitswerten.

1 Zahlen

Das zur Darstellung des Dezimalpunktes verwendete Zeichen kann durch Setzen der Umgebungsvariablen FXRADIXCHAR gesteuert werden. Zulässige Werte sind "." und ",". Das zur Darstellung des Tausender-Trennzeichen verwendete Zeichen (',' oder '.') ergibt sich hieraus entsprechend.

Ist die Umgebungsvariable FXRADIXCHAR nicht gesetzt, verhält sich *FIX*, als besäße sie den Wert "."; genügt ihr Wert nicht obigen Anforderungen, erfolgt ein Programmabbruch.

Bei numerischen Feldern ohne Formatangabe ist der Wert von FXRADIXCHAR unerheblich; hier wird der Dezimalpunkt stets als '.' dargestellt.

2 Datum (FXDATETYPE)

Die Reihenfolgen der Komponenten Tag, Monat und Jahr sowie das Trennzeichen zwischen diesen kann durch Setzen der Umgebungsvariablen FXDATE gesteuert werden.

Die ersten vier Zeichen von FXDATE müssen aus den Zeichenfolgen "D", "M" und "Y4" gebildet sein, von denen jede genau einmal auftreten muss.

Optional kann ein fünftes Zeichen folgen, bei dem es sich um ein Satzzeichen (Klasse *punctuation*) des Zeichensatzes ISO 646:1983 handeln muss (d.h. die Zeichen '@', '[', ']', '_', '{', '}', '~' sind unzulässig). Anstelle von '.' wird dann dieses Zeichen als Komponenten-Trennzeichen verwendet.

Durch ein sechstes Zeichen, das den gleichen Bedingungen wie das fünfte genügen muss, kann ein alternatives Trennzeichen für solche Darstellungen vorgegeben werden, die die Komponente Tag nicht beinhalten. Anderenfalls wird das erstrangige Trennzeichen verwendet.

Ist die Umgebungsvariable FXDATE nicht gesetzt, verhält sich *FIX*, als besäße sie den Wert "DMY4./"; genügt ihr Wert nicht obigen Anforderungen, erfolgt ein Programmabbruch.

Länge und Steuerwert bestimmen zusammen die Darstellung:

Beispiel:

FXDATE	Default	"MDY4/"
len		
5	mm/yy	mm/dd
6	ddmmyy	mmddyy
7	mm/yyyy	mm/yyyy
8	dd.mm.yy	mm/dd/yy
10	dd.mm.yyyy	mm/dd/yyyy

Bei der Eingabe in ein Datum-Feld wird jedes Zeichen aus der Klasse *Punctuation* (vgl. Kapitel *40*) in das passende Trennzeichen umgesetzt.

Damit bei der Kodierung einer Anwendung, d.h. in der C-Source und den Beschreibungsdateien, nicht differenziert auf die möglichen Werte von FXDATE eingegangen werden muss, sollten folgende Grundsätze beachtet werden:

- Faustregel: Im Programmcode sollte die Verwendung von Datum-Literalen möglichst vermieden werden.
- In Beschreibungsdateien sind Datum-Literale, die z.B. als Defaultwert und in Wertvorgaben hinterlegt werden, entsprechend dem Format “dd.mm.yyyy” anzugeben; zum Einlesen wird die *FIX*-Routine fxrdefmtdate() verwendet.

Die Abkürzungen für Monats- und Wochentagsnamen entnimmt *FIX* der Meldungsdatei und stellt sie in den Variablen S_months und S_weekdays bereit.

3 Wahrheitswerte (FXTRUTHTYPE)

Die für die Werte *Nein* und *Ja* des Typs FXTRUTHTYPE benutzen Zeichencodes können im Programmtext durch Belegung der Variablen S_no (Vorbelegung ‘n’) und S_yes (Vorbelegung ‘j’) umgesetzt werden.

Wie S_no und S_yes dargestellt werden, kann durch Setzen der Umgebungsvariablen FXTRUTH gesteuert werden.

Der Wert von FXTRUTH muss hierzu aus zwei Zeichen bestehen, bei denen es sich um voneinander verschiedene Kleinbuchstaben des Zeichensatzes ISO 646:1983 handeln muss. Das erste Zeichen wird zur Darstellung von S_no, das zweite zur Darstellung von S_yes verwendet.

Ist die Umgebungsvariable FXTRUTH nicht gesetzt, verhält sich *FIX*, als besäße sie den Wert “nj”; genügt ihr Wert nicht obigen Anforderungen, erfolgt ein Programmabbruch.

Bei der Eingabe in ein FXTRUTHTYPE-Feld müssen die zur Visualisierung verwendeten Zeichen (oder die zu ihnen äquivalenten Großbuchstaben) benutzt werden. Gleiches gilt für die Reaktion auf jn_msg().

In Beschreibungsdateien sind Wahrheitswerte entsprechend der internen Repräsentation (S_no, S_yes) zu hinterlegen.

4 Sprachunterstützung für Werkzeuge

Eine anderssprachige Oberfläche von Tools kann nur durch die Erstellung eines zu `$FXDIR/de` äquivalenten Verzeichnisbaums und das Setzen der Umgebungsvariable FXTOOLHOME auf die Wurzel dieses Verzeichnisbaums erreicht werden.

Für die Programme **led**, **repgo** und **repprep** kann die Sprache Englisch verwendet werden. Dazu sind die Umgebungsvariable

```
FXTOOLHOME=$FXDIR/us
```

und

```
FXTOOLLANG=(irgendein Wert ungleich “”)
```

zu setzen. Dies beeinflusst jedoch nur die Oberfläche und die *kodierten* Meldungstexte. Andere sprachabhängige Eigenschaften sind über entsprechende Umgebungsvariablen einzustellen (FXTRUTH, …).

5 Interne Meldungstexte

Durch Setzen der Umgebungsvariablen

```
FXSTRERRLANG=us
```

werden interne Meldungen von *FIX* in Englisch ausgegeben.

Anhang B C-ISAM gestützte Selos

Bis Version 2.7 kannte *FIX* nur C-ISAM basierte Selos, die beim Einsatz von IBM Informix SE eine gegenüber dem Zugriff mittels SQL deutlich bessere Performance bieten. Mit dem überwiegenden Einsatz von IBM Informix Dynamic Server haben sie jedoch an Bedeutung verloren und werden daher nicht weiterentwickelt. So werden etwa die Typen DATETIME und INTERVAL nicht unterstützt.

Mit diesen Selos kann ein Join von Spalten ein oder mehrerer Basistabellen definiert werden. Die Tabellen müssen paarweise verschieden sein und jede muss eine mit einem Index versehene Spalte besitzen, die explizit aufgeführt und auch in die Auswahl aufgenommen werden muss. Zu jeder Tabelle kann eine Folge von einfachen Vergleichen zwischen ausgewählten Spalten und Feldwerten oder Konstanten angegeben werden, die alle erfüllt sein müssen, damit der Tabellenausschnitt beim Join berücksichtigt wird.

Hier hat die Beschreibungsdatei folgenden Aufbau:

selo [<objectname>] [**at** <ypos> <xpos>] [**size** <anz>] [**set** <daten>]
from <selection>
[**join** <column> **with** <selection>]*
end

wobei

<objectname> ::= <identifier>
<ypos> ::= <integer>
<xpos> ::= <integer>
<anz> ::= <unsigned>
<daten> ::= <integer> | <integer>:<integer>
<selection> ::= <table> **key** <column> <item> [, <item>]* [**where** <condition>]
<table> ::= <identifier>
<item> ::= <column> [< heading> [<format> [<target>]]]
<heading> ::= <string>
<format> ::= <string>
<target> ::= #<maskname>.<fieldname>
<condition> ::= <comparison> [**and** <condition>]
<comparison> ::= <column> <op> [<column> | #<field> | constant]
<column> ::= <identifier>
<field> ::= <identifier>
<op> ::= (**!=** | **<** | **<=** | **=** | **>=** | **>** | **matches** | **in**)
<constant> ::= <string>

Beispiel:

```
selo
from auftr key aufnr            Tabelle und Indexspalte
aufnr …,                        erste Spalte muss mit Indexspalte identisch sein
datum …,
bearb …,
kunr …
where bearb > " "               Bedingungen an erste Tabelle
join kunr                       zum Join verwendete Spalte der ersten Tabelle
```

```
with kunde key kunr          weitere Tabelle mit Indexspalte
kunr …,                      zum Join verwendete Spalte der zweiten Tabelle
name …
end
```

Bei einem Selo, das an ein Feld gebunden ist, kann der Anwender die Suche dadurch beschränken, dass er einen *Startwert* bzgl. der Indexspalte der ersten Tabelle angibt. Da *FIX* zur Erfassung des Startwertes das Feld benutzt, mit dem das Selo verknüpft ist, müssen sich der Typ des Feldes und der der Indexspalte entsprechen.

FIX rettet automatisch den bisherigen Inhalt des Feldes und löscht es. Der Anwender kann dann

- einen Wert eingeben, den *FIX* als Ausgangswert für die anfangs zu lesenden Sätze benutzt; hierbei kann mit der Taste f2 (CPYFIELD) der gerettete Feldwert als Startwert übernommen werden

oder aber

- mit der Taste f8 (KEYFIRST) am Anfang des verwendeten Index aufsetzen
- mit der Taste f0 (KEYLAST) am Ende des verwendeten Index aufsetzen

Anschließend liest *FIX* Datensätze, bis keine Sätze mehr gefunden werden *oder die maximale Satzzahl pro Seite erreicht ist*.

Enthält die Beschreibung keine Angaben zur Wertübernahme in ein Feld, wird nach einer Auswahl der Wert der ersten Spalte in das Feld übernommen, an das das Selo gebunden ist.

Für dbmust() nutzt *FIX* aus, dass auf der mit dem Feld assoziierten Spalte ein Index liegt, kann also sehr effizient herausfinden, ob ein Satz mit diesem Spaltenwert existiert.

Hinweise:

- Bei Join und Vergleich wird NULL als gewöhnlicher Wert behandelt, der kleiner als alle anderen Werte des Typs ist.
- Fehler bei Zugriff auf die Datenbank können einen Programmabbruch mittels fastexit() bewirken.

SQL-Unterstützung für C-ISAM-Selos

Das Aufkommen von IBM Informix Dynamic Server machte es erforderlich, eine Datenbeschaffung bereitzustellen, die die für IBM Informix SE entwickelten Selos weiterhin unterstützt, aber ohne C-ISAM auskommt.

FIX ist in der Lage, zu für C-ISAM entwickelten Selos parametrisierte SQL-Anweisungen zu generieren und diese zur Datenbeschaffung zu benutzen. Bei Versionen ohne C-ISAM-Unterstützung geschieht dies automatisch, bei solchen mit C-ISAM-Unterstützung nur, wenn die Umgebungsvariable NO_ISAM definiert ist.

Die Anweisung wird bereits beim Laden des Selos erzeugt, wobei die sonst mit Hilfe von C-ISAM ermittelte Strukturinformation über die beteiligten Tabellen und Spalten mittels einer describe-Anweisung beschafft wird.

Beispiel:

Das auf Seite 525 wiedergegebene Selo ergibt folgende Grundanweisung

```
select auftr.aufnr, auftr.datum, auftr.bearb, auftr.kunr, kunde.kunr, kunde.name
from auftr, kunde
where auftr.kunr = kunde.kunr and auftr.bearb > " "
```

Die im Selo aufgeführten Spaltennamen werden mit dem Tabellennamen qualifiziert, wenn mehr als eine Tabelle am Selo beteiligt ist. Die Join-Spalte wird doppelt selektiert; die Join-Bedingungen erscheinen, verknüpft durch den Operator **and**, vor den tabellenspezifischen Bedingungen.

Folgende Unterschiede im Verhalten sind allerdings zu beachten:

- Die Indexinformation ist bei SQL nicht nutzbar, d.h.

wird kein Startwert angegeben, werden die Sätze in einer unbestimmten Reihenfolge geliefert,

wird ein Startwert angegeben und gibt es eine Spalte mit einer export-Beziehung zu diesem Feld, werden die Sätze bzgl. dieser Spalte geordnet.

- Da die Daten über einen Cursor gelesen werden müssen, ist ein Zurückblättern *vor* den Startwert nicht möglich; auch kann das Selo erheblich mehr Anlaufzeit benötigen (open cursor).
- Durch die Berücksichtigung von Zugriffsrechten und Sperren sowie durch den unterschiedlichen Zeitpunkt der Parameterauswertung und Datenbeschaffung - die Ergebnismenge liegt mit der ersten Leseoperation (fetch) fest - können sich Abweichungen hinsichtlich der selektierten Menge ergeben.
- Der Wert NULL wird SQL-konform behandelt.

Beim jedem Neuaufsetzen der Suche wird ein geeigneter Cursor bereitgestellt, und zwar, wenn kein Startwert angegeben wird, einer für die Grundanweisung, anderenfalls einer für die um die Bedingung

<expr> ≥ #<maskname>.<fieldname>

und eine Sortierangabe erweiterte Anweisung, wobei

<fieldname>	Name des Feldes, an das das Selo gebunden ist
<maskname>	Name der Maske, zu der dieses Feld gehört
<expr>	Ausdruck, der die Spalte liefert, die mit dem Feld durch Wertübergabe verknüpft ist

Beispiel:

```
select auftr.aufnr, auftr.datum, auftr.bearb, auftr.kunr, kunde.kunr, kunde.name
from auftr, kunde
where auftr.aufnr >= #AUFTR.AUFNR and auftr.kunr = kunde.kunr and auftr.bearb > " "
order by 1
```

Wird das Selo zur Feldprüfung und Datenbeschaffung verwendet (vgl. Seite 214), wird die Grundanweisung um die Bedingung

<expr> = #<maskname>.<fieldname>

erweitert. Aus Effizienzgründen wird eine vorhandene order by-Klausel unterdrückt.

Anhang C Advanced Features

Im folgenden werden zusätzliche Features erläutert. Einige haben zurzeit noch einen experimentellen Status. Die Bekanntgabe solcher erweiterter Funktionen soll Ihnen als *FIX*-Nutzer frühzeitig Hinweise auf Entwicklungstendenzen aufzeigen und die Möglichkeit zu einem versuchsweisen Einsatz eröffnen.

Die Entwicklungen der vergangenen Jahre in unserem Hause haben gezeigt, dass die so vorgestellten Funktionen meist in mehr oder minder abgewandelter Form in eine der nächsten Versionen Eingang finden und damit in den definierten und unterstützten *FIX*-Funktionsumfang aufgenommen werden.

1 Event-Testhilfe

Die *FIX*-Library beinhaltet zwei Funktionen, die es dem Entwickler erleichtern, die Events während der Bearbeitung eines Menüs oder einer Maske zu verfolgen.

Der Aufruf fxdbg_start() instruiert *FIX*, die bei den einzelnen Verarbeitungsschritten ankommenden Events in die Datei `fxdbg`*`ppppp`*`001` zu protokollieren, wobei *ppppp* für die Nummer des Prozesses steht. fxdbg_stop() beendet die Protokollierung (endgültig oder bis zu einem erneuten Aufruf der Funktion fxdbg_start()). Läuft das Programm nicht im Entwicklermodus (Schalter -dev), sind beide Funktionen wirkungslos.

2 Profiling von SQL-Anweisungen

IBM Informix ESQL/C bietet z.Z. leider kein Verfahren an, den “Aufwand” zu messen, den die Ausführung einer SQL-Anweisung verursacht. Um dem Entwickler dennoch eine Hilfe an die Hand zu geben, Engpässe in seiner Anwendung zu erkennen und Anweisungen aufzuspüren, bei denen sich eine Optimierung lohnen würde, stellt *FIX* eine Methode bereitet, um die Zeit zu protokollieren, die während der Abarbeitung der SQL-Anweisung verstreicht (real time).

Das Verfahren beruht auf folgender Idee:

Der ESQL/C-Präprozessor ersetzt SQL-Anweisungen in der Source durch Aufrufe von Funktionen der ESQL/C-Library, wobei die genaue Form der Ersetzung (Funktionsname, Parameter) zwischen den verschiedenen IBM Informix ESQL/C-Releases differiert. Diese Funktionsaufrufe kann man mit Hilfe des C-Präprozessors abfangen und auf eigene Funktionen (“Wrapper”) umlenken.

Beispiel:

Eine Anweisung der Form

EXEC SQL select … col_i … into … var_i … from … $tabelle_j$ … where *condition* ;

in der .ec-Datei ergibt einen Funktionsaufruf der Form

_iqslct(*v_cursor*, *v_txtp*, v_ *n1*, *v_p1*, *v_n2*, *v_p2*);

in der .c-Datei, wobei *v_cursor*, *v_txt*, *v_n1*, *v_p1*, *v_n2*, *v_p2* aus der Anweisung abgeleitete Argumente sind.

Definiert man nun in der .c-Datei ein Makro wie

```
#define _iqslct(p_cursor, p_txtp, p_n1, p_p1, p_n2, p_p2) \
                p_iqslct(_PROFILE_TAG_, p_cursor, p_txtp, p_n1, p_p1, p_n2, p_p2)
```

wobei _PROFILE_TAG_ für einen String steht, der die Source-Stelle genauer spezifiziert, z.B.

```
#define _PROFILE_TAG_          profkey(__FILE__, __LINE__) [1]
```

und stellt - in einem anderen Modul, das _iqslct() nicht als Makro definiert - eine Funktion

```
int p_iqslct(char *key, … p_cursor, … p_txtp, … p_n1, … p_p1, … p_n2, … p_p2)
{
    int ret;

    lies Stand der Uhr
    ret = _iqslct(p_cursor, p_txtp, p_n1, p_p1, p_n2, p_p2);
    lies Stand der Uhr erneut und lege gewonnene Information unter dem Schlüssel key ab
    return(ret);
}
```

zur Verfügung, die man in sein Programm einbindet, so kann man Information über die Dauer der Ausführung der Anweisung sammeln.

Für die gängigsten SQL-Anweisungen definiert *FIX* Makros der obigen Form in der Header-Datei `sqlprof.h`. Die zugehörigen Funktionen sind Bestandteil der *FIX*-Library.[2] Sie sind so ausgelegt, dass sie - in dynamisch verwalteten Datenstrukturen - zu jedem Schlüsselwert *key* jeweils die Zahl der Aufrufe sowie Minimum, Maximum und Summe der gemessenen Zeiten speichern. Die Zeiteinheit ist systemabhängig (im Allgemeinen ‘clock ticks’).

Um dieses Feature zu nutzen, muss in die .ec-Datei eine Zeile der Form

```
#include <sqlprof.h>
```

eingefügt werden. Sofern man das Makro _PROFILE_TAG_ nicht selbst definiert, greift die folgende in `sqlprof.h` enthaltene Definition:

```
#ifndef _PROFILE_TAG_
#define _PROFILE_TAG_          profkey(__FILE__,__LINE__)
#endif
```

Da die Makrodefinitionen in der Datei `sqlprof.h` in Präprozessor-Direktiven

1. Die *FIX*-Funktion profkey() hinterlegt in einem statischen Bereich, der bei jedem Aufruf überschrieben wird, einen aus ihren beiden Argumenten gebildeten String.

2. Im Verzeichnis `$FXDIRSYS/lib` finden sich außerdem entsprechend übersetzte Versionen derjenigen Module der *FIX*-Library, die die internen Datenbankoperationen abwickeln:
`fxcursor-p.o` - SQL-Selos und Choices,
`sql-p.o` - C-ISAM-Selos und
`perf-p.o` - perf().

```
#ifdef SQLPROF
#endif  /* SQLPROF */
```

eingebettet sind, greift der Profiling-Mechanismus nur, wenn der Compiler mit dem Schalter -DSQLPROF aufgerufen wird. Damit die zu dem jeweiligen IBM Informix ESQL/C-Release passenden Makros gültig gemacht werden (`sqlprof.h` enthält Makros für die Versionen 2.1x, 4.0x, 4.1x, 5.0x, 6.0x, 7.1x, 7.2x, 9.1x, 9.2x, 9.3x, 9.4x und 9.5x), muss außerdem das Makro INF geeignet definiert werden:

-DINF=*mn* für IBM Informix ESQL/C *m.n*x [1]

Durch Aufruf der Funktion

```
void printsqlprof(void)
```

werden die erstellten *profile records* in die Datei `/tmp/pprof.$$` geschrieben:

file,line SQL-stmt	#calls	time			
		total	min	max	avg
…	…	…	…	…	…

Bei einer großen Anwendung ist man häufig nur an Profiling-Daten zu bestimmten Teilen interessiert. Daher deaktiviert *FIX* bei einem Aufruf von

```
(void)set_profstate(0);
```

das Anlegen von profile records, bis wieder ein Aufruf

```
(void)set_profstate(1);
```

erfolgt. set_profstate() liefert den bisherigen Zustand als Ergebnis zurück.

Um besonders zeitintensive Anweisungen aufzuspüren, kann außerdem ein Schwellenwert (in Zeiteinheiten) spezifiziert werden, bei dessen Überschreiten ein Eintrag in `/tmp/pprof.$$` erfolgt:

```
extern int  prof_threshold;
```

Voreingestellt wird prof_threshold = 10 (Millisekunden).

1. *FIX*-Versionen für IBM Informix ESQL/C 7.1x werten im Code für das SQL-Profiling zusätzlich das Makro ESQLC_MINOR_RELEASE aus, da mit IBM Informix ESQL/C 7.12(.UC1) die Anzahl der Parameter von _iqexecute() gegenüber 7.11 von 7 auf 8 erhöht wurde, wodurch die Makrodefinition INF=71 nicht zur Differenzierung ausreicht. Wird IBM Informix ESQL/C 7.1*k* benutzt wird, muss beim Einschluss von `sqlprof.h` zusätzlich die Definition ESQLC_MINOR_RELEASE=*k* vorgenommen werden.

Anhang D Das *FIX*-Runtime-System

Das *FIX*-Runtime-System beinhaltet lediglich

Komponenten zur Installation

- `installfix`
- `cmd/serialfix`
- `spec/fxserial*`
- `install/`

Komponenten zur Registrierung

- `licensefix`
- `bin/fxlicense` und die von diesem Programm verwendeten Masken (`mfo/`), Choices (`cho/`), Layouts (`pan/`), Daten (`cho/`) und Meldungstexte (`etc/fix_msg`, `etc/messages*`)
- `spec/fxinstfile*`
- `cmd/sav_fxinstfile`

den Lizenz-Server

- `bin/liserver`, `bin/likill`

Konfigurationsdateien

- `etc/stdprofile`, `spec/p.default`, `etc/fxprofile`
- `fix.rc`
- `etc/*/charset`

Geräteanpassungen

- `etc/fxtermcap/`, `etc/fxkeycap/`, `etc/*/fxcharmap/`, `etc/*/fxkeymap/`

Info-Dateien

- `spec/PortID`
- `distrib/CONT-run, distrib/CONT-run.sys`
- diverse README

einige Basis-Tools

- `cmd/fxask`
- `cmd/fxdisplay`, `bin/fxdisplay`
- `cmd/hardcopy`
- `cmd/less`, `bin/less`
- `cmd/lpcat`, `bin/lpcat`
- `cmd/msgprep`, `bin/msgprep`
- `cmd/tcout, bin/tcout`, `cmd/tgoto`, `bin/tgoto`, `cmd/ntty`

- `bin/keyinit`, `bin/xcho`
- `cmd/path`
- `bin/datum`, `bin/jahr`, `bin/monat`, `bin/tag`

Standard-Meldungstexte

- `runtime/fx_texte`, `runtime/messages` *

Standard-Hilfetexte

- `runtime/hpd/`

Standard-Masken

- `mfo/kal.mfo`

Mit * markierte Dateien werden erst bei der Installation angelegt.

Anhang E Softwarevoraussetzungen

Nach unserem Kenntnisstand benötigen Sie je nach IBM Informix-Release folgende Lizenz-Produkte der Firma IBM. Die Angaben bitten wir gegebenenfalls bei Ihrem IBM Informix-Lieferanten zu verifizieren.

FIX-Entwicklungssystem

IBM Informix Client SDK (IBM Informix ESQL/C),
bei Versionen mit C-ISAM-Unterstützung zusätzlich C-ISAM-Development

"Reines" *FIX*-Runtime-System

IBM Informix Connect,
bei Versionen mit C-ISAM-Unterstützung ggf. C-ISAM-Runtime

Zusätzlich ist ein passendes IBM Informix-Backend-Produkt wie SE, Dynamic Server etc. erforderlich.

Anhang F Der Zeichensatz ISO 8859:15

Nachstehend sind für die Zeichen des Zeichensatzes IS0 8859:15, die über den ASCII-Zeichensatz hinausgehen und darstellbar sind, die Ersatzdarstellungen aufgeführt:

Zeichen [a]	ISO 8859:15		IBM437 [b]		GERMAN7		HP Roman 8 [c]	
(nonbreaking space)	\240	\xA0					\240	\xA0
¡	\241	\xA1					\270	\xB8
¢	\242	\xA2					\277	\xBF
£	\243	\xA3					\257	\xAF
(EURO)	\244	\xA4						
¥	\245	\xA5					\274	\xBC
(S mit caron)	\246	\xA6					\353	\xEB
§	\247	\xA7					\275	\xBD
(s mit caron)	\250	\xA8					\354	\xEC
©	\251	\xA9						
ª	\252	\xAA	\246	\xA6			\371	\xF9
«	\253	\xAB					\373	\xFB
¬	\254	\xAC						
–	\255	\xAD					\366	\xF6
®	\256	\xAE						
¯	\257	\xAF					\260	\xB0
°	\260	\xB0					\263	\xB3
±	\261	\xB1					\376	\xFE
2	\262	\xB2						
3	\263	\xB3						
(Z mit caron)	\264	\xB4						
µ	\265	\xB5					\363	\xF3
¶	\266	\xB6					\364	\xF4
·	\267	\xB7					\362	\xF2
(z mit caron)	\270	\xB8						
1	\271	\xB9						
º	\272	\xBA	\247	\xA7			\372	\xFA
»	\273	\xBB					\375	\xFD
Œ	\274	\xBC						

Zeichen [a]	ISO 8859:15		IBM437 [b]		GERMAN7		HP Roman 8 [c]	
œ	\275	\xBD						
Ÿ	\276	\xBE					\356	\xEE
¿	\277	\xBF					\271	\xB9
À	\300	\xC0	◊				\241	\xA1
Á	\301	\xC1	◊				\340	\xE0
Â	\302	\xC2	◊				\242	\xA2
Ã	\303	\xC3	◊				\341	\xE1
Ä	\304	\xC4	\216	\x8E	\133	\x5B	\330	\xD8
Å	\305	\xC5	\217	\x8F			\320	\xD0
Æ	\306	\xC6	\222	\x92			\323	\xD3
Ç	\307	\xC7	\200	\x80			\264	\xB4
È	\310	\xC8	◊				\243	\xA3
É	\311	\xC9	\220	\x90			\334	\xDC
Ê	\312	\xCA	◊				\244	\xA4
Ë	\313	\xCB	◊				\245	\xA5
Ì	\314	\xCC	◊				\346	\xE6
Í	\315	\xCD	◊				\345	\xE5
Î	\316	\xCE	◊				\246	\xA6
Ï	\317	\xCF	◊				\247	\xA7
(uppercase eth)	\320	\xD0					\343	\xE3
Ñ	\321	\xD1	\245	\xA5			\266	\xB6
Ò	\322	\xD2	◊				\350	\xE8
Ó	\323	\xD3	◊				\347	\xE7
Ô	\324	\xD4	◊				\337	\xDF
Õ	\325	\xD5	◊				\351	\xE9
Ö	\326	\xD6	\231	\x99	\134	\x5C	\332	\xDA
×	\327	\xD7						
Ø	\330	\xD8	◊				\322	\xD2
Ù	\331	\xD9	◊				\255	\xAD
Ú	\332	\xDA	◊				\355	\xED
Û	\333	\xDB	◊				\256	\xAE
Ü	\334	\xDC	\232	\x9A	\135	\x5D	\333	\xDB
(Y acute)	\335	\xDD					\261	\xB1
(uppercase thorn)	\336	\xDE					\360	\xF0
ß	\337	\xDF	\341	\xE1	\176	\x7E	\336	\xDE
à	\340	\xE0					\310	\xC8
á	\341	\xE1					\304	\xC4

Zeichen [a]	ISO 8859:15		IBM437 [b]		GERMAN7		HP Roman 8 [c]	
â	\342	\xE2					\300	\xC0
ã	\343	\xE3	◊				\342	\xE2
ä	\344	\xE4	\204	\x84	\173	\x7B	\314	\xCC
å	\345	\xE5	\206	\x86			\324	\xD4
æ	\346	\xE6	\221	\x91			\327	\xD7
ç	\347	\xE7	\207	\x87			\265	\xB5
è	\350	\xE8					\311	\xC9
é	\351	\xE9	\202	\x82			\305	\xC5
ê	\352	\xEA					\301	\xC1
ë	\353	\xEB					\315	\xCD
ì	\354	\xEC					\331	\xD9
í	\355	\xED					\325	\xD5
î	\356	\xEE					\321	\xD1
ï	\357	\xEF					\335	\xDD
(lowercase eth)	\360	\xF0					\344	\xE4
ñ	\361	\xF1	\244	\xA4			\267	\xB7
ò	\362	\xF2					\312	\xCA
ó	\363	\xF3					\306	\xC6
ô	\364	\xF4					\302	\xC2
õ	\365	\xF5	◊				\352	\xEA
ö	\366	\xF6	\224	\x94	\174	\x7C	\316	\xCE
÷	\367	\xF7						
ø	\370	\xF8	◊				\326	\xD6
ù	\371	\xF9					\313	\xCB
ú	\372	\xFA					\307	\xC7
û	\373	\xFB					\303	\xC3
ü	\374	\xFC	\201	\x81	\175	\x7D	\317	\xCF
(y acute)	\375	\xFD					\262	\xB2
(lowercase thorn)	\376	\xFE					\361	\xF1
ÿ	\377	\xFF	\230	\x98			\357	\xEF

a. Bei Zeichen, die von PostScript nicht unterstützt werden, ist in Klammern der Name angegeben.

b. Zeichen mit dem Eintrag ◊ können je nach landesspezifischer Variante ebenfalls zulässig sein. *Buchstaben* ohne einen Eintrag werden in diesem Zeichensatz nicht unterstützt.

c. Dieser Zeichensatz ist für *FIX* nicht relevant.

Hinweis

FIX 2.9.4 verwendete bis Dezember 2000 noch den Zeichensatz IS0 8859:1, der sich von IS0 8859:15 an einigen Positionen unterscheidet:

Zeichen	ISO 8859:1		IBM437		GERMAN7		HPRoman8	
¤	\244	\xA4					\272	\xBA
(broken bar)	\246	\xA6						
¨	\250	\xA8					\253	\xAB
´	\264	\xB4					\250	\xA8
¸	\270	\xB8						
$^1/_4$	\274	\xBC					\367	\xF7
$^1/_2$	\275	\xBD					\370	\xF8
$^3/_4$	\276	\xBE					\365	\xF5

Anhang G Migrationshinweise

Die folgenden Abschnitte weisen auf Unterschiede der *FIX*-Version 3.1.0 gegenüber der Vorversion hin. Sie sollen Hilfestellung bei der Umstellung einer Anwendung geben, die mit *FIX*-Version 3.0.0 entwickelt wurde.

1 Neue Features

1.1 Neue Attribute von Feldern

Felder vom Typ FXCHARTYPE unterstützen eine neue Feldeigenschaft ASCIIONLY (vgl. Seite 44). Wird sie vergeben, werden bei der Eingabe nur ASCII-Zeichen akzeptiert.

1.2 Insert-Modus bei FXCHARTYPE-Feldern

FIX verwendet bedingt durch seine Herkunft für die Erfassung von FXCHARTYPE-Feldern standardmäßig einen *Overwrite-Modus*, d.h. Zeicheneingabe manipuliert das Zeichen unter der Schreibmarke. Da dies bei grafischen Oberflächen untypisch ist, unterstützt *FIX* seit Version 3.1.0 alternativ einen *Insert-Modus*. Vgl. hierzu “Bedienungsmodi” auf Seite 34.

Wie der Insert-Modus aktiviert oder im Programm abgeschaltet wird, ist auf Seite 449 bzw. auf Seite 444 beschrieben, wie man den Benutzer den Modus zur Laufzeit wählen lassen kann, auf Seite 447.

1.3 Editieren von Dateien mittels editor()

Die Funktion editor() ist nun auch bei Einsatz des grafischen Frontends *FIX/Win* nutzbar. In diesem Fall liest *FIX* die Datei und übermittelt ihren Inhalt und eine zu FXCHARSET passende Windows-Codepage-Nummer an das Frontend. Nach dem Editvorgang liefert *FIX/Win* den neuen Dateiinhalt (entsprechend dieser Codepage), der dann von *FIX* in die Datei zurückgeschrieben wird.

Zum Editvorgang im grafischen Frontend *FIX/Win* vgl. die *FIX/Win*-Dokumentation.

1.4 Profiling von SQL-Anweisungen

Hier wird nun auch das API von IBM Informix ESQL/C 3.00 unterstützt (vgl. `fix/include/sqlprof.h`).

1.5 Konfigurierbares Verfahren zur Bildschirmaktualisierung

Durch das Zusammenspiel der neuen Funktion sync_refresh() (Seite 391) , der neuen Ressource SkipSyncRefresh (Seite 448) und der neuen globalen Variable S_skip_sync_refresh (Seite 446) erhält eine *FIX*-Anwendung größeren Einfluß auf die Bildschirmaktualisierung. Interessant ist dies für Anwendungen, die eine geringe Bandbreite oder eine langsame Verbindung zwischen dem Ausgabegerät / grafischem Frontend und dem Rechner, auf dem die *FIX*-Anwendung läuft, aufweisen.

Beschrieben ist dieses Feature in “Konfigurierung” auf Seite 274.

Die aus den Vorversionen bekannte Funktion refresh() existiert natürlich weiterhin: refresh() entspricht allerdings sync_refresh("refresh", FALSE[1]).

Hinweis::

Soll eine bestehende Anwendung so umgestellt werden, dass sie das Unterdrücken von Bildschirmaktualisierungen ermöglicht, sollten alle im Anwendungscode vorhandenen refresh()-Aufrufe näher untersucht werden. Es muss dann jeweils entschienen werden, ob vor dem Aufruf der Wert der Variablen S_skip_sync_refresh geändert werden sollte oder ob der refresh()-Aufruf durch einen der Funktion sync_refresh() mit geeignetem booleschen Argument ersetzt werden sollte.

Anwendungen, die das Feature nicht nutzen, können unverändert bleiben, da der für dieRessource SkipSyncRefresh eingestellte Wert unerheblich ist, solange im Anwendungscode nicht die Variable S_skip_sync_refresh explizit auf einen Wert ungleich 0 gesetzt wird.

2 Die Struktur des *FIX*-Verzeichnisses

Bisher unmittelbar im *FIX*-Verzeichnis `fix` liegende Unterverzeichnisse, die ausschließlich Dateiressourcen für die zu *FIX* gehörenden Programme enthalten, wurden in einem neuen Verzeichnis `fix/de` zusammengefasst (vgl “Die Struktur des FIX-Verzeichnisses” auf Seite 51). Dies erleichert - in Verbindung mit der neuen Umgebungsvariablen FXTOOLHOME - ihre “Portierung” zu einer anderen Sprache und/oder einem anderen Zeichensatz (vgl. hierzu etwa `fix/us`). Jeder “FXTOOLHOME”-Verzeichnisbaum[2] besitzt auch ein eigenes Laufzeit-Verzeichnis `runtime`, um die *FIX*-Programme unabhängig von dem für Anwendungen verwendeten Laufzeit-Verzeichnis `fix/runtime` (→ Umgebungsvariable FXRUNTIMEDIR) zu machen.

`fix/de` [NEU]

Standard-“FXTOOLHOME”.

`fix/us` [NEU]

alternatives “FXTOOLHOME”; derzeit nur für die Programme **led** und **rled** verwendbar.

So ergeben sich folgende Verlagerungen, wobei in den Verzeichnissen Dateien teilweise auch modifiziert, umbenannt oder hinzugefügt wurden:

FIX 3.0.0	*FIX* 3.1.0
`fix/cho`	`fix/de/cho`
`fix/tool_us/cho`	`fix/us/cho`
`fix/hpd`	`fix/de/hpd`
`fix/tool_us/hpd`	`fix/us/hpd`

1. Nach unseren Erfahrungen enthalten Anwendungsprogramme häufig überflüssige Aufrufe von refresh(), so dass FALSE als die sinnvollere Einstufung erschien.
2. Ein “FXTOOLHOME”-Verzeichnis kann als eine Art “FXHOME” für *FIX* selbst angesehen werden.

FIX 3.0.0	*FIX* 3.1.0
`fix/men`	`fix/de/men`
`fix/mfo`	`fix/de/mfo`
`fix/tool_us/mfo`	`fix/us/mfo`
`fix/pan`	`fix/de/pan`
`fix/tool_us/pan`	`fix/us/pan`
`fix/sel`	`fix/de/sel`
`fix/tool_us/sel`	`fix/us/sel`
`fix/etc/fix_msg`	`fix/de/fix_msg`
`fix/etc/fx_texte.us`	`fix/us/fix_msg`
`fix/etc/messages`	`fix/de/messages`, `fix/us/messages`
`fix/runtime`	`fix/runtime` (für Anwendungen), `fix/de/runtime` (für *FIX*-Tools)
`fix/tool_us/runtime`	`fix/us/runtime` (für *FIX*-Tools wie **led**)
`fix/tools_us`	(entfällt)

Vgl. hierzu auch die geänderte Bedeutung der Umgebungsvariable FXTOOLLANG in Abschnitt 4.

3 Initialisierung von *FIX*-Programmen

Die Reihenfolge der von fxinit() vorgenommenen Initialisierungsschritte (vgl. “Initialisierung von FIX” auf Seite 129) hat sich gegenüber der Vorversion geändert. Aus technischen Gründen wird der Zeichensatz, den die *FIX*-Installation benutzt, bereits unmittelbar nach der Bestimmung des *FIX*-Verzeichnisses (Umgebungsvariable FXDIR) ermittelt, in-den die Umgebungsvariable FXCHARSET ausgewertet wird.

4 Sprachunterstützung für Werkzeuge

Der Wert der Umgebungsvariable FXTOOLLANG hat keinen Einfluß mehr darauf, welche Dateiressourcen *FIX*-Tools verwenden. Für eine anderssprachige Oberfläche von Tools muss das in Abschnitt 4 auf Seite 522 beschriebene Verfahren gewählt werden.

FXTOOLLANG beeinflusst allerdings weiterhin bei den Layouteditoren **led** und **rled** die von fxstrerror() verwendete Sprache. Dies gilt jedoch nicht mehr für den - zum *FIX*-Runtime-System gehörenden - Lizenz-Server. Um die (wenigen) deutschen Meldungen des Lizenz-Servers stattdessen in Englisch zu erhalten, bleibt nur die Möglichkeit, die Umgebungsvariable FXSTRERRLANG (auf einen beliebigen Wert) zu setzen.

5 Das Entwicklermenü fxm

fxm weist eine Reihe von - in der Praxis unbedeutenden -Einschränkungen auf, die die Stabilität verbessern.

Die in den Masken von **fxm** benutzten Felder verweigern generell die Eingabe von Nicht-ASCII-Zeichen. Einzige Ausnahmen sind

- das Feld zur Eingabe der Argumente in der durch “Source→Programm ausführen” aufgeblendeten Maske

und

- das Feld zur Eingabe des Kommandos in der durch “Usw.→Kommando ausführen” aufgeblendeten Maske.

Alle von **fxm** zur Beschaffung von Datei- oder Verzeichnisvorschlägen verwendeten Choices in `de/cho` benutzen ein neues Hilfsscript `cmd/do/getdirentries.sh` , anstatt unmittelbar das Programm **ls** aufzurufen. Das Script dient dazu, die brauchbaren Vorschläge herausfiltern; insbesondere werden nur solche Verzeichniseinträge berücksichtigt, die ausschließlich aus den ASCII-Zeichen ! (0x21) bis ~ (0x7E) gebildet sind (vgl. Hinweis auf Seite 61).

Die durch “FIX-Masken→generieren Maske” aufgeblendete Maske enthält ein Feld zur Eingabe einer Datenbanktabelle. Die auf diesem Feld angebotene Auswahl (`cho/relations.cho`) benutzt zur Beschaffung der Tabellenvorschläge das Hilfsscript `cmd/do/getrelations`. Bei Versionen für IBM Informix benutzt dieses Script statt **isql** ein neues Hilfsprogramm **getdb_tables** und berücksichtigt nur solche Tabellen, deren tabid größer-gleich 100 ist und die einen Namen besitzen, der ausschließlich aus den ASCII-Zeichen ' ' (0x20) bis ~ (0x7E) gebildet ist.

Das Hilfsprogramm **obgen** (vgl. Seite 63) akzeptiert ein optionales zweites Argument *layouttype* mit möglichen Werten old-txt (Vorbelegung) und old-bin. **fxm** ruft bei Anwahl des Menüpunktes “FIX-Masken→Layout-Editor” oder “Menü→Layout-Editor” **obgen** aus Kompatibilitätserwägungen mit zweitem Argument old-bin auf.

Beispiele:

Für $1 gleich xxx.mfo (und $2 gleich old-txt) generiert **obgen** die Dateien `mfo/xxx.mfo`

```
mask XXX “XXX Maske” pan/xxx.pan
22 80 0 0
END_OF_MFO
```

und `pan/xxx.pan`

```
<Version 293>
<Geometry 22 80>
<CharacterSet 'unspecified'>
```

Für $1 gleich xxx.mfo und $2 gleich old-bin generiert **obgen** die Dateien `mfo/xxx.mfo`

```
mask XXX “XXX Maske” mly/xxx.mly
22 80 0 0
END_OF_MFO
```

und `mly/xxx.mly`; Letztere enthält 22×80 attributlose Leerzeichen.

Die durch “Source→Programm ausführen” aufgeblendete Maske enthält ein Feld zur Eingabe eines Programmnamens. Die auf diesem Feld angebotene Auswahl (`cho/bins.cho`) benutzt zur Beschaffung der Programmvorschläge das Hilfsscript `cmd/do/getprograms`. Dieses Script berücksichtigt nur noch solche Dateien, die für den Benutzer ausführbar sind (nur Versionen für Unix) und die einen Namen besitzen, der ausschließlich aus den ASCII-Zeichen ' ' (0x20) bis ~ (0x7E) gebildet ist.

Das von “Usw.→DB-Tool aufrufen” aufgerufene Script `cmd/do/ex-dbtool` verwendet bei Versionen für IBM Informix nun **dbaccess** (statt **isql**), wenn die Umgebungsvariable FXDBTOOL nicht gesetzt ist.

fxm entfernt die Umgebungsvariable CDPATH aus der Umgebung, um die störende Ausgabe des Verzeichnisnamens zu vermeiden, wenn ausgeführte Shellkommandos mittels **cd** das Verzeichnis wechseln.

6 Die Erstellung und Bearbeitung von Menüs und Masken mittels led oder rled

led und **rled** wurde hinsichtlich Konfigurierbarkeit, Optik und Bedienung stark überarbeitet.

Zur Konfiguration mittels Ressourcen vgl. “Konfiguration” auf Seite 119 und “Aufruf von Programmen zur Vor- und Nachbehandlung” auf Seite 120.

Hinweis:

Die alte (nicht dokumentierte) Umgebungsvariable LED_AFTER_PAN_SAVE wird zwar weiterhin noch unterstützt. Statt ihrer sollte jedoch die neue Ressource AfterSavePanCmd (vgl. Seite 120) verwendet werden.

Die Programme fordern nicht mehr unbedingt ein Dateiargument (vgl Seite 119).

Zur Benutzung unter einem grafischen Frontend vgl. Seite 121 .

Beschreibungsmodus

Das nicht veränderbare Feld der Maske “Feld-Daten”, das die Eigenschaften klartextlich darstellt, erlaubt analog zum Feld *Eigenschaften* ebenfalls über f9 ebenfalls eine Auswahl der Eigenschaften.[1]

WYSIWYG-Modus

Die Statuszeile zeigt weiterhin die Tasten zur Attributumschaltung und ihre Erklärungen an. Allerdings zeigt *FIX* jetzt oberhalb der Taste an, ob das zugehörige Attribut aktiviert ist (“ on ”), und oberhalb der Erklärung, ob das jeweilige Attribut für die aktuelle Position gesetzt ist (“ on ”).

FIX zeigt jetzt oberhalb der Tastenerklärung “Position Mode” an, ob der Position Mode aktiviert ist (“ on “) oder man sich im Edit Mode befindet.

Der Position Mode besitzt einen Untermodus, den *Size Mode*. Zum Size Mode vgl. Seite 115.

Das aktuelle Element wird optisch hervorgehoben (vgl. hierzu die Ressourcen ActiveEntryAttr und ActiveFieldAttr auf Seite 119).

Im Edit Mode des **led** und im **rled** ist die Sondertaste f6 entfallen.

Beim Schreiben eines Zeichens in der letzten Spalte findet kein Zeilen- und Spaltenwechsel statt.

7 Neue Werkzeuge zur Layoutkonversion

FIX enthält zwei neue Werkzeuge zur Konversion von Layoutbeschreibungen: **mlyconv** und **panconv**. Sie sind im Kapitel “Konversion von Layouts” auf Seite 123 näher beschrieben.

8 Semigrafik

Für den für Füllstellen in Feldern mit *f*->displen > *f*->len verwendeten Grafikcode ‘^’ fehlte bislang die Ersatzdarstellung; nunmehr ist als Ersatzdarstellung das Zeichen ‘^’ des Standardzeichensatzes voreingestellt (vgl. Seite 271).

1. Dazu wird das Feld *Eigenschaften* besucht und dessen Choice gestartet. Nach einer Auswahl wird wieder das ursprüngliche Feld besucht.

9 Registrierung

Bei der Registrierung erlaubt *FIX* als “Name des Lizenznehmers” und “Ort der Installation” (und auch bei “Eingesetzte Hardware”, “Datenbanksystem” und “Lizensierte Produkte”) nur noch Angaben, die ausschließlich aus ASCII-Zeichen bestehen. Die entsprechenden Felder der von **licensefix** benutzten Erfassungsmaske berücksichtigen dies, indem sie die Eingabe anderer Zeichen abweisen. Verletzt eine eingelesene Lizenzinformation die obige Bedingung, können die betroffenen Felder nicht vorwärts verlassen werden.

10 Änderungen im *FIX*-Verzeichnis

Dieser Abschnitt führt detailliert alle Änderungen im *FIX*-Dateibaum auf, die für den Anwendungsentwickler von Interesse sein könnten.[1] In den Abschnitten zu den einzelnen Unterverzeichnissen wird meist zunächst auf Komponenten des Runtime-Systems, dann auf die des Enwicklungssystems und schließlich auf die Beispielanwendungen eingegangen.

`fix.rc`

enthält Hinweise auf zusätzliche, für diese Version irrelevante Ressourcen (TrailingNonASCIIBlanksMode).

`Makefile` (für Installation)

Die Datei

- berücksichtigt die neue Lage der deutschen Meldungsdateien für *FIX*-Tools in `de`,
- berücksichtigt auch die englischen Meldungsdateien für *FIX*-Tools in `us`,
- berücksichtigt, dass zusätzlich das Programm **getdb_tables** gebunden werden muss (vgl. Seite 551).

`fd`, `fw`,
`demostart` [NEU]

Die zur Benutzung der mitgelieferten Beispielanwendungen dienenden Scripts `fd` und `fw` stützen sich auf ein neues Hilfscript `demostart`.

Das Verzeichnis `fix/etc`

`etc/stdprofile`

verwendet dbaccess statt isql als Ersatzwert für FXDBTOOL.

`etc/fxprofile`

entfernt auch die Umgebungsvariablen FXTOOLHOME und FXRUNTIMEDIR aus der Umgebung.

Die bisher in diesem Verzeichnis liegenden Meldungsdateien `etc/fix_msg` und `etc/messages` für *FIX*-Tools wurden in das neue Verzeichnis `fix/de` verlagert.

Die Laufzeitumgebung in `fix/runtime`

`runtime/fx_texte`, `runtime/messages`

Der Kopftext der Datei `runtime/fx_texte` ist nicht mehr zeichensatzabhängig.

Die Meldungen wurden modifiziert und erweitert. Neben Meldungen, die nur von *FIX*-eigenen Programmen benutzt werden (10077-10080, 10089, 10098-10099, 10970-10975), betrifft dies die Meldungen

10132

1. Kleine Korrekturen wie die Behebung von Schreibfehlern oder die Anpassung von Versionsnummern bleiben unerwähnt.

10345
10560-10561
10570-10576
10580-10589
Von *FIX*-Entwicklern vorgenommene Übersetzungen sind entsprechend anzupassen.

Die Scripts in `fix/cmd`

`cmd/less`

behandelt den in der Umgebungsvariable FXPAGER hinterlegten Programmpfad mit Vorrang vor `${FXDIRSYS}/bin/less`.

`cmd/load` (Versionen für IBM Informix)

verwendet dbaccess statt isql als Ersatzwert für FXDBTOOL.

`cmd/mkfixenv`

verwendet bei Versionen für IBM Informix dbaccess statt isql als Ersatzwert für FXDBTOOL.

trägt beim Anlegen der Datei `fx_texte` einen zeichensatzunabhängigen Kopftext ein.

generiert beim Anlegen der Datei `makefile` einen erweiterten Kopf mit einer Vereinbarung ESQLCFLAGS= und zwei hinweisende Kommentaren (vgl. `gen/all/MakeHeader.pat` auf Seite 550).

`cmd/mlyconv` [NEU]

startet das gleichnamige Programm aus `$FXDIRSYS/bin` (vgl. `mlyconv` auf Seite 73 und Seite 550).

`cmd/panconv` [NEU]

startet das gleichnamige Programm aus `$FXDIRSYS/bin` (vgl. `panconv` auf Seite 73 und Seite 550).

`cmd/run`

differenziert beim Aufruf von **exec** zwischen dem Programmnamen ($1) und weiteren Argumenten.

`cmd/selo` (Versionen für IBM Informix)

verwendet dbaccess statt isql als Ersatzwert für FXDBTOOL.

`cmd/styp`

berücksichtigt die neue Feldeigenschaft ASCIIONLY.

`cmd/unload` (Versionen für IBM Informix)

verwendet dbaccess statt isql als Ersatzwert für FXDBTOOL.

Die Header-Dateien in `fix/include`

Einige Header-Dateien enthalten vom Definiertheitszustand des Makros UCS_1993 abhängigen Code. Dieses Makro darf zum Zeitpunkt des Einschlusses der Datei nicht definiert sein, da die anderenfalls vorgenommenen Vereinbarungen für diese *FIX*-Version nicht zutreffen.

`include/fix/accept.h`

definiert ein neues Makro ASCIIONLY für die entsprechende Feldeigenschaft (vgl. “Neue Attribute von Feldern” auf Seite 541).

Deklariert die neue globale Variable S_field_insert_mode (vgl. Seite 444).

`include/fix/command.h`

Durch Verwendung eines Guard-Makro ist jetzt auch ein Mehrfacheinschluss unschädlich.

`include/fix/extern.h`

Durch Verwendung eines Guard-Makro ist jetzt auch ein Mehrfacheinschluss unschädlich.

`include/fix/fixwin.h`

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code.

`include/fix/fxtypes.h`

deklariert zusätzlich die globale Variable S_MB_CUR_MAX (vgl. Seite 443) und verwendet sie bei der Definition der Makros

FXCHARSIZE(l)

FXGRAPHICSSIZE(l)

`include/fix/keys.h`

definiert die numerischen Werte für Events relativ zu einem Basiswert K_BASE. K_BASE ist allerdings so gewählt, dass sich keine Veränderung der numerischen Werte ergibt.

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code.

`include/fix/libfix.h`

schließt zusätzlich die Header-Datei `proto/char_pto.h` und, sofern `wchar.h` eingeschlossen zu sein scheint, auch `proto/mb_pto.h` ein.

`include/fix/obj.h`

Datentyp menue:

Der Datentyp von men_curr_indent (intern) wurde von int nach short geändert.

Neue Komponente men_curr_xoff (intern).

Neue Makros (nur zum internen Gebrauch):

DISPPOS_UNCHANGED

DISPPOS_TO_RIGHT

DISPPOS_TO_LEFT

Datentyp field:

Der Datentyp von disppos (intern) wurde von int nach int:12 geändert.

Neue Komponente dispscroll (intern).

Neue Komponente infolen: sie enthält die Größe des Speicherplatzes, der an info bereitsteht.

`include/fix/release.h`

Das Makro FIX_VERSION wird nun mit dem Wert 310 definiert.

`include/proto/char_pto.h` [NEU]

Neue Header-Datei, die die neue Funktion

int fx_char_copy(char **dest*, int *destsize*, char **src*, int *srcsize*, BOOLEAN *trimright*, int *maxcnt*, int *padchar*, BOOLEAN *addzero*)

(vgl. Seite 424) deklariert.

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code.

`include/proto/fixwin_pto.h`

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code.

`include/proto/fxctype_pto.h`

deklariert zusätzlich die neue Funktion

int fxisblank(int *code*)

(vgl. Seite 427).

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code.

`include/proto/fxstring_pto.h`

Die Deklaration der nicht mehr in der *FIX*-Library enthaltenen Funktion strnswap() entfällt.

`include/proto/keys_pto.h`

deklariert zusätzlich die neue Funktion

Event get_lastinput(void)

(vgl. Seite 287).

`include/proto/mask_pto.h`

Die früher hier zu findenden Deklarationen der Funktionen

int layout_getchar(obj ∗*objp*, int *y*, int *x*)

int layout_getcharset(obj ∗*objp*, int *y*, int *x*)

int layout_getvideo(obj ∗*objp*, int *y*, int *x*)

wurden nach `include/proto/obj_pto.h` verschoben.

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code

`include/proto/mb_pto.h` [NEU]

Neue Header-Datei, die bis auf das Guard-Makro (MB_PTO_INCLUDED) keine Vereinbarungen enthält.

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code.

`include/proto/obj_pto.h`

enthält die früher in `include/proto/mask_pto.h` enthaltenen Deklarationen der Funktionen

int layout_getchar(obj ∗*objp*, int *y*, int *x*)

int layout_getcharset(obj ∗*objp*, int *y*, int *x*)

int layout_getvideo(obj ∗*objp*, int *y*, int *x*)

Deklariert zusätzlich die neuen Funktionen

int layout_getattr(obj ∗*objp*, int *y*, int *x*)

int layout_setchar(obj ∗*objp*, int *y*, int *x*, int *c*)

int layout_setattr(obj ∗*objp*, int *y*, int *x*, int *attr*)

(vgl. Seite 372, Seite 374 und Seite 374).

`include/proto/output_pto.h`

deklariert zusätzlich die neuen Funktionen

void sync_refresh(char ∗*dbgmsg*, BOOLEAN *force*)

int putstrwidth(char ∗s)

(vgl. Seite 391 und Seite 434).

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code.

`include/proto/selo_pto.h`

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code.

`include/exec.h`

Durch Verwendung eines Guard-Makro ist jetzt auch ein Mehrfacheinschluss unschädlich.

`include/msgread.h`

Durch Verwendung eines Guard-Makro ist jetzt auch ein Mehrfacheinschluss unschädlich.

`include/sqlprof.h`

unterstützt nun auch das API von IBM Informix ESQL/C 3.00.

`include/testflags.h`

Die Deklaration der globalen Variable B_test, die bereits in `include/fix/basics.h` deklariert wird, ist entfallen.

Hinzugekommen ist ein neues Schalter-Bit (8) für den Schalter -test (vgl. Seite 128), das detailliertere Meldungen bewirkt, wenn *FIX* unzulässige Zeichen in (FX)CHARTYPE- oder (FX)GRAPHICSTYPE-Werten erkennt.

`include/video.h`

Die Definition des Makros VIDEOMASK wurde verändert.

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code.

Die Musterdateien für die Source-Generierung in `fix/gen`

`gen/all/MakeHeader.pat`

Diese Musterdatei gilt für den neuen SourceStyle "V3.1.0" (Voreinstellung). Die makefile-Vorlage unterscheidet sich gegenüber der vorigen Version durch einen erweiterten Kopf mit einer Vereinbarung ESQLCFLAGS= und zwei hinweisende Kommentaren (→ `cmd/mkfixenv`).

`gen/all/MakeHeader_29X.pat` [NEU]

Diese Musterdatei stimmt mit der Datei `gen/all/MakeHeader.pat` der Vorversion überein und wird für den SourceStyle "V2.9.3" und "V2.9.4" verwendet.

`gen/mask/GnrcRule1.pat`
`gen/mask/GnrcRule4.pat`
`gen/mask/GnrcRule4c.pat`

Diese Musterdateien gelten für den neuen SourceStyle "V3.1.0" (Voreinstellung). Er verwendet modifizierte Regeln zum Precompilieren. Dem Precompiler-Aufruf wurde $(ESQLCFLAGS) hinzugefügt, um generell geltende Einstelllungen mitgeben zu können.

`gen/mask/GnrcRule1_29X.pat` [NEU]
`gen/mask/GnrcRule4_29X.pat` [NEU]
`gen/mask/GnrcRule4c_29X.pat` [NEU]

Diese Musterdateien stimmen mit den bis auf den Namensanteil `_29X` gleichlautenden Dateien der Vorversion überein und werden für SourceStyle"V2.9.4" verwendet.

Die Code-Vorlagen und Beispiele in `fix/src`

Einige Beispiele enthalten vom Definiertheitszustand des Makros UCS_1993 abhängigen Code, der für diese *FIX*-Version nicht zutrifft.

`src/chdisplay.c`

Die Datei enthält vom Definiertheitszustand des Makros UCS_1993 abhängigen Code.

`src/main.c`, `src/perfdemo.c`

Die Definition des Makros FIXRELEASE wurde an die Versionsnummer angepasst.

Die Online-Dokumentation in `fix/doc`

Sie berücksichtigt die neuen Funktionen.

`$FXDIRSYS/bin`

`bin/mlyconv` [NEU]

Zu diesem Programm siehe “Neue Werkzeuge zur Layoutkonversion” auf Seite 545 und “mlyconv” auf Seite 123.

`bin/panconv` [NEU]

Zu diesem Programm siehe “Neue Werkzeuge zur Layoutkonversion” auf Seite 545 und “panconv” auf Seite 123.

`bin/getdbtables` [NEU]

Neues Hilfsprogramm für **fxm**.

Bei Versionen für UNIX wird dieses Programm erst bei der Installation erzeugt (→ `Makefile`, `dev_objs/getdb_tables.o`).

`bin/tabinfo`

Dieses Hilfsprogramm für **fxm** besitzt einen neuen Schalter -db_locale. Wird er angegeben, wird zunächst eine Zeile der Form DB_LOCALE=*DB_LOCALE der Datenbank* ausgegeben.

`$FXDIRSYS/dev_objs`

`dev_objs/getdb_tables.o` [NEU]

Modul, aus dem bei der Installation das Hilfsprogramm **getdb_tables** gebunden wird (→ `Makefile` auf Seite 546).

`$FXDIRSYS/etc`

`compiler.ccg` (nur Windows)

Zu Änderungen vgl. Abschnitt 16 auf Seite 555.

11 Die Anwendung demo

Dieser Abschnitt führt Änderungen der Beispielanwendung im Verzeichnis `demo` auf, die für den *FIX*-Administrator oder den Anwendungsentwickler von Interesse sein könnten.

`demo/mkdemo`,
`demo/.profile`

Der Datenbankname wurde an die Versionsnummer angepasst und lautet nun demo310.

`demo/fix.rc`

enthält zusätzlich Hinweise auf Alternativen für die Ressource fgmask.SourceStyle, die ab dieser Version die neue Vorbelegung “V3.1.0” besitzt, und setzt die neue Ressource fgmask.BytesPerCharMethod (vgl. Seite 450).

`demo/c/makefile*`

entsprechen im Aufbau dem neuen Source-Stil “V3.1.0”:

- erweiterter Dateikopf (ESQLCFLAGS, Hinweis auf BPC).
- Precompiler wird mit $(ESQLCFLAGS) aufgerufen.

`demo/c/*.h`, `demo/c/*.ec`, `demo/c/*.c`

entsprechen in der Art, wie Hostvariablen des Typs char[] deklariert werden, dem neuen Source-Stil “V3.1.0”.

Die Definition des Makros FIXRELEASE wurde an die Versionsnummer angepasst.

`demo/chp`

enthält als Platzhalter statt einer Datei AUFTR_AUFNR nun READ_ME.

`demo/hpd`

Die Hilfetexte wurden an die neue Rechtschreibung angepasst (nur ß vs. ss).

12 Die Anwendung windemo

Dieser Abschnitt führt Änderungen der Beispielanwendung im Verzeichnis `windemo` auf, die für den *FIX*-Administrator oder den Anwendungsentwickler von Interesse sein könnten.

Die Anwendung telefon (Beschreibungsdateien, Quellcode, Regeln) ist entfallen.

`windemo/mkdemo`,
`windemo/.profile`

Der Datenbankname wurde an die Versionsnummer angepasst und lautet nun windemo310.

`windemo/men/*.men`

Die Menübeschreibungen benutzen statt binärer Layouts nun textuelle Layouts aus `windemo/pan`.

In `windemo/men/demo.men` ist die Möglichkeit zum Starten von Reports ist entfallen.

Die Menüs `windemo/men/men1.1.men` und `windemo/men/tree.men` sind entfallen.

`windemo/mfo/*.mfo`

Die Maskenbeschreibungen benutzen statt binärer Layouts nun textuelle Layouts aus `windemo/pan`.

`windemo/pan/*.pan` [NEU]

Dies sind die Layoutdateien zu den Menü- und Maskenbeschreibungen in `windemo/men` und `windemo/mfo`.

`windemo/mly`

enthält nur noch eine READ_ME-Datei.

`windemo/hpd`

Die Hilfetexte wurden an neue Rechtschreibung angepasst (nur ß vs. ss).

`windemo/cmd`

Die Scripts zum Aufruf des REP sind entfallen.

`windemo/rpp`

Das Verzeichnis ist entfallen.

13 *FIX*-Library

13.1 Globale Variablen

Die *FIX*-Library enthält einige neue globale Variablen:

- char *S_charset: readonly
  siehe Seite 443.
- int B_CodesetRequired :readonly
  siehe Seite 439.
- int S_MB_CUR_MAX: readonly
  siehe Seite 443.
- int S_TrailingNonASCIIBlanksMode: readonly

siehe Seite 443.

- int S_chlen_to_dblen_factor: readonly
  siehe Seite 443.
- int S_field_insert_mode: r/w
  siehe Seite 444.
- int S_skip_sync_refresh: r/w
  siehe Seite 446.

13.2 Funktionen

Die *FIX*-Library enthält einige neue Funktionen:

- int fx_char_copy(char **dest*, int *destsize*, char **src*, int *srcsize*, BOOLEAN *trimright*, int *maxcnt*, int *padchar*, BOOLEAN *addzero*)
  siehe Seite 424.
- int fxisblank(int *code*)
  siehe Seite 427.
- Event get_lastinput()
  siehe Seite 287.
- int layout_getattr(obj **objp*, int *y*, int *x*)
  siehe Seite 372.
- int layout_setchar(obj **objp*, int *y*, int *x*, int *c*)
  siehe Seite 374.
- int layout_setattr(obj **objp*, int *y*, int *x*, int *attr*)
  siehe Seite 374.
- int putstrwidth(char **s*)
  siehe Seite 434.
- int sync_refresh(char **dbgmsg*, BOOLEAN *force*)
  siehe Seite 391.

14 Source-Generierung

FIX 3.1.0 verwendet standardmäßig einen neuen Stil “V3.1.0” für Sourcen (vgl. Ressource SourceStyle).

14.1 Source-Generierung für Masken

Der neue Stil “V3.1.0” unterscheidet sich von dem zuvor verwendeten Stil “V2.9.4” durch die Art, wie Hostvariablen für FXCHARTYPE- und FXGRAPHICSTYPE-Felder definiert werden:

	V2.9.4	V3.1.0
FXCHARTYPE-Feld der Länge n	char[n’], wobei n’ gleich n + 1	char[n * factor + 1]

	V2.9.4	V3.1.0
FXGRAPHICSTYPE-Feld der Länge n	fixchar[n'], wobei n' gleich n	fixchar[n * factor]

Hierbei ist *factor* wahlweise eine numerische Konstante oder das Makro BPC, je nachdem, ob die neu eingeführte Ressource fgmask.BytesPerCharMethod den Wert "constant" (Voreinstellung) oder "macro" besitzt. Im Falle "constant" entfällt die (oben unterstrichene) Multiplikation, wenn die numerische Konstante den Wert 1 besitzt.

Der Wert der Konstante wird von der *FIX*-Version bestimmt (→ Variable S_MB_CUR_MAX) und ist in dieser Version 1.

Um die Einstellung "macro" nutzen zu können, wurde auch die makefile-Datei modifiziert. Durch

- eine zusätzliche Vereinbarung
  ESQLCFLAGS =
  # Achtung: je nach Variante der Generierung muss -EDBPC=<n> hinzugefuegt werden.
  (vgl.`cmd/mkfixenv`, `gen/all/MakeHeader.pat`)
- einen Hinweis bei der Definition von CFLAGS
  CFLAGS = $(FXCFLAGS) -I$(SRC)
  # Achtung: je nach Variante der Generierung muss -DBPC=<n> hinzugefuegt werden.
  (vgl.`cmd/mkfixenv`, `gen/all/MakeHeader.pat`)

und

- eine Erweiterung der Regel zum Aufruf des Precompilers
  $(ESQLC) $(ESQLCFLAGS) *source*.ec
  (vgl.`gen/mask/GnrcRule1.pat`, `gen/mask/GnrcRule4.pat`, `gen/mask/GnrcRule4c.pat`)

wurden die Voraussetzung dafür geschaffen, dem Precompiler und dem C-Compiler einen Wert für das Makro BPC mitzuteilen.

Generierung im Stil der Vorversion

Wird die Ressource [fgmask.]SourceStyle explizit auf den Wert "V2.9.4" gesetzt, werden die Hostvariablen und die makefile-Datei wie bei *FIX* 3.0.0 generiert. *FIX* verwendet in diesem Fall spezielle Musterdateien mit dem Namensbestandteil `_29X` aus `gen`.

14.2 Source-Generierung für Menüs

Der neue Stil "V3.1.0" unterscheidet sich von dem zuvor verwendeten Stil "V2.9.4" nur darin, dass beim Anlegen einer makefile-Datei der gleiche Kopf wie bei der Source-Generierung für Masken verwendet wird (vgl. Abschnitt 14.1).

Generierung im Stil der Vorversion

Wird die Ressource [fgmenu.]SourceStyle explizit auf den Wert "V2.9.4" gesetzt, wird die makefile-Datei wie bei *FIX* 3.0.0 generiert. *FIX* verwendet in diesem Fall spezielle Musterdateien mit dem Namensbestandteil `_29X` aus `gen`.

15 Kommunikation mit dem grafischen Frontend *FIX/Win*

FIX 3.1.0 verwendet zur Kommunikation mit einem grafischen Frontend ein gegenüber der Vorversion stark modifiziertes Protokoll. Daher kommt als Frontend nur *FIX/Win* 3.1.0 oder höher infrage. Umgekehrt kann mit *FIX/Win* 3.1.0 kein *FIX*-Programm bedient werden, das mit einer *FIX*-Version kleiner 3.1.0 übersetzt wurde.

Zur Möglichkeit, Textdateien zu editieren, vgl. Seite 541.

16 *FIX* für Windows

Die mitgelieferte Steuerdatei `etc/compiler.ccg` für das MKS-Toolkit-Programm **cc.exe**, das einen UNIX-artigen Aufruf des Microsoft C-Compilers ermöglicht, wurde modifiziert.

Zum einen ist jetzt die Kombination der Schalter -c und -o möglich, d.h. die Erzeugung einer `.obj`-Datei mit einem von der `.c`-Datei abweichenden Namen. Hierfür wird auf den /Fo-Schalter von Visual C/C++ zurückgegriffen.

Zum anderen benutzt die *FIX*-Library die Funktionen CharUpper und CharLower aus dem Win32-API, wodurch das Einbinden der `user32.dll` erforderlich wird. Daher wurde die Definition von Libs entsprechend erweitert.

Hingegen wurde `advapi32.dll` aus der Definition von Libs entfernt, da diese Bibliothek bereits in der SPECLIBS-Angabe in `m.default` aufgeführt ist.

Stichwortverzeichnis

1

A

B

C

D

E

F

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z