The Xerox Advanced ViewPoint Unit

Your Instructors This Week

Your lecturers and lab assistants throughout this course will be some combination of those listed below. Please feel free to ask questions. If you have any comments or suggestions about the course structure or content, please message Debbie at the e-mail address listed below. (E-mail messages help us keep track of all comments.)

Debbie MacKay <MacKay:OSBU North:Xerox>

Mark Hahn

Leslie Kanno

Goal of this Class

This class covers ViewPoint programming topics that are considered "advanced," which refers to subjects that are either more difficult to understand or are harder to find documentation on (or both).

After this class, you should be able to implement Selection and TIP managers, interact programmatically with documents, and improve the performance of your applications through program optimization.

Pre-requisites

This class is geared for the seasoned programmer who is already familiar with programming in the ViewPoint environment. Thorough knowledge of Mesa is assumed; in addition, you should also be familiar with the common ViewPoint interfaces (e.g., Attention, Containee, Display, FormWindow, MenuData, StarWindowShell, Window, XFormat, XString, et al). This class will be very difficult for you without a basic understanding of the environment.

We're Not Your Mother

The general format of the class is lecture in the morning, lab in the afternoon. The lab has no format: you are expected to complete your assignments (or at least try) but you can come and go as you wish.

Major No-No: Don't leave for three hours in the middle of the afternoon and then come back and expect us to stay here with you until 10 PM.

Day 1: TIP

Outline

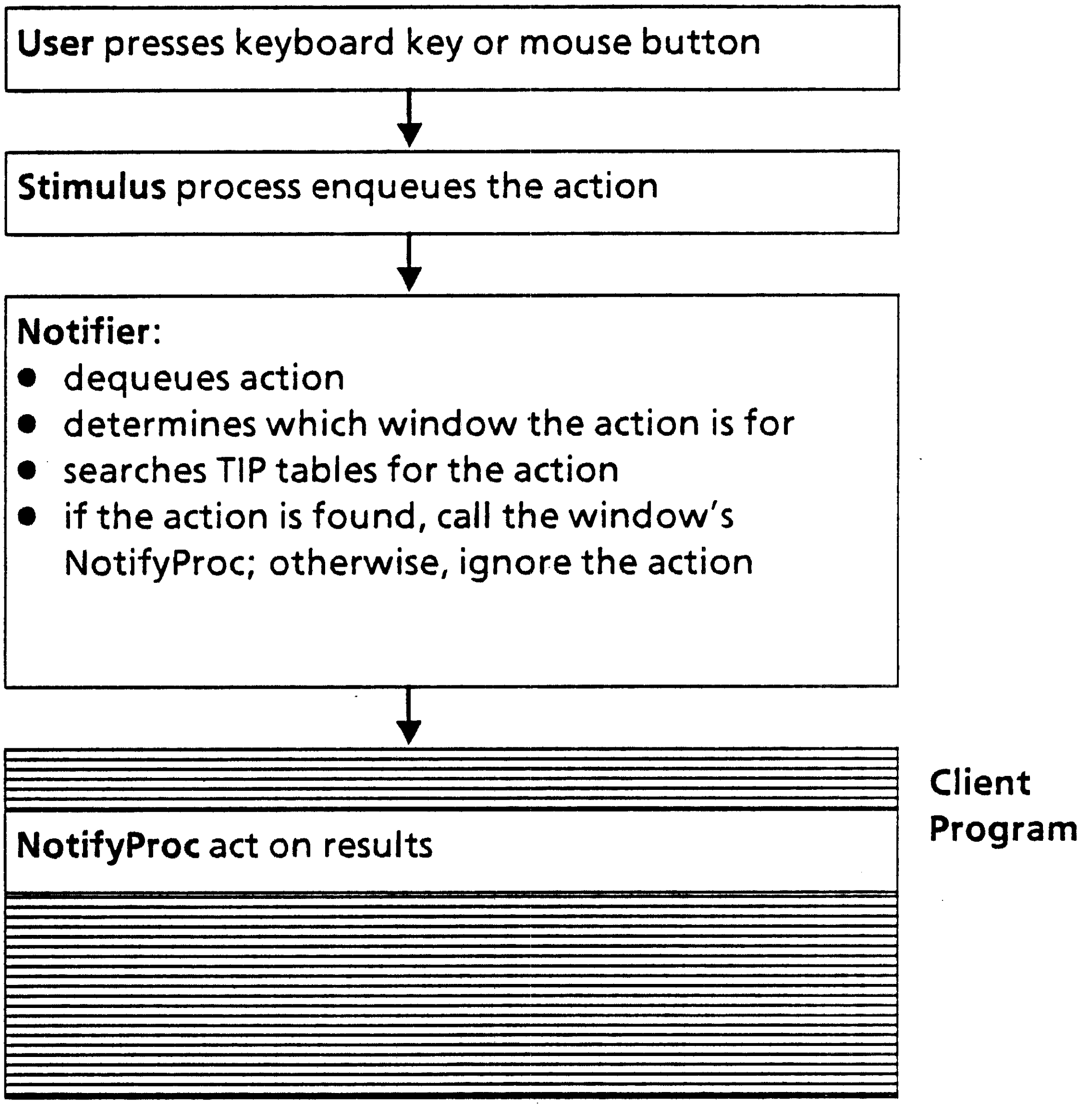
I. TIP
   A. Notification Mechanism
   B. Tables
      1. Structure of a TIP table
      2. The TIPStar Structure
      3. Installing and Removing tables
   C. NotifyProcs
   D. Input Focus
   E. Periodic Notification
   F. Setting the Manager and Call-Back Notification
   G. Attention and User Abort
   H. Character Translation
   I. Stuffing Input into a Window

Notification Mechanism

TIP (Terminal Interface Package) provides a mechanism to translate user's keyboard and mouse actions into client program actions. TIP must direct user input to one of the windows on the display and notify the program that implements that window's functionality.

This notification is accomplished by two processes, the **Stimulus** and **Notifier**. The Stimulus is a high-priority process that watches for keyboard and mouse actions; it queues these events for the Notifier process. The Notifier dequeues each event and associates it with a window. After determining the correct window for a user action, the Notifier searches for the action in a set of TIP tables. If the action is found, the Notifier passes the corresponding list of results to the window's underlying application via its NotifyProc. The application can then perform the appropriate actions in response.

Notification Mechanism (cont'd)



User presses keyboard key or mouse button

Stimulus process enqueues the action

Notifier:

- dequeues action
- determines which window the action is for
- searches TIP tables for the action
- if the action is found, call the window's NotifyProc; otherwise, ignore the action

NotifyProc act on results

Client Program

Tables

The first thing the Notifier process does, after determining which window the user's action is for, is locate the hardware action in a TIP table. TIP tables are structured much like Mesa SELECT statements. The left hand side of the table contains the hardware actions and the right hand side contains a list of results that will be passed to the application's NotifyProc.

```
SELECT TRIGGER FROM
  Point Down =>
    SELECT ENABLE FROM
      LeftShift Down => COORDS, ShiftedPoint;
    ENDCASE=> COORDS, PointClick;
  Adjust Down => COORDS, AdjustDown;
  ENDCASE...
```

OR

```
SELECT TRIGGER FROM
  S Down => TurnLeft;
  D Down => TurnRight;
  K Down => Forward;
  L Down => COORDS, Fire;
  ENDCASE...
```

TRIGGER and ENABLE Statements

TRIGGER statements check the next action from the Notifier's input queue and branch to the appropriate choice.

ENABLE statements indicate the current condition of the keyboard or mouse.

TRIGGER terms can appear in sequence, separated by ANDs and can be mixed with ENABLE terms separated by WHILEs. The complete syntax of TIP tables can be found in the *ViewPoint Programmer's Manual.*

lc

```
SELECT TRIGGER FROM
  V Down AND D UP => COORDS, Penicillin;
  DELETE Down WHILE NEXT Down => DeleteNextObject;
  ...
  ENDCASE
```

This statement could be read, "V goes down and then D goes up with no other actions in between". The next statement would match if the DELETE key goes down while the NEXT key is down.

Mouse Actions and Timeouts

You can recognize other useful actions in TIP tables such as mouse motion and when the mouse enters or exits a body window. In this example, note that the action is the mouse moving and the static condition is that the Point button is held down.

```
SELECT TRIGGER FROM
  MOUSE => SELECT ENABLE FROM
    Point Down => COORDS, PointMotion;
    ENDCASE => COORDS, Mouse;
  EXIT => COORDS, Exit;
  ENTER => COORDS, Enter;
ENDCASE...
```

You can also specify timeout conditions for user actions. A timeout following a trigger indicates a timing condition that must hold between this trigger and its predecessor. Timeouts are in milliseconds.

```
SELECT TRIGGER FROM
  Point Down =>
    SELECT TRIGGER FROM
      Point Up BEFORE 200 AND
        Point Down BEFORE 200 => COORDS, DoubleClick;
      Adjust Down BEFORE 300 => COORDS, PointAndAdjust;
    ENDCASE => COORDS, PointDown;
  Adjust Down => COORDS, AdjustDown;
ENDCASE...
```

Creating a Private TIP Table

ViewPoint supplies a chain of default TIP tables that your program can use. If you want to recognize additional actions, you can create a private TIP table or a private chain of tables for a particular application. Integrating a private TIP table requires that you first create a table, and then put that table into the system's chain of tables. You will see how to insert tables later; for now we will concentrate on creating tables with `TIP.CreateTable`.

```
TIP.CreateTable: PROCEDURE[
  file: XString.Reader,
  z: UNCOUNTED ZONE ← NIL,
  contents: XString.Reader ← NIL]
  RETURNS[table: TIP.Table];

TIP.Table: TYPE = LONG POINTER TO TIP.TableObject;
TIP.TableObject: TYPE;
```

Creating and Destroying Private TIP Tables (example)

```
table: TIP.Table ← NIL;

CreateTIPTable: PROC = {
  fileName:XString.ReaderBody ← XString.FromSTRING["Prog.TIP"L];
  contents: XString.ReaderBody ← XString.FromSTRING["
  SELECT TRIGGER FROM
    S Down => TurnLeft;
    D Down => TurnRight;
    K Down => Forward;
    L Down => Fire;
  ENDCASE...
  "L];
  table ← TIP.CreateTable[
    file: @fileName,
    contents: @contents! TIP.InvalidTable => RESUME];
  IF table = NIL THEN  {
    mh: XMessage.Handle = Defs.GetMessageHandle[];
    msg: XString.ReaderBody ← XMessage.Get[mh, Defs.kbadSyntax];
    Attention.Post[@msg]}];
  };
```

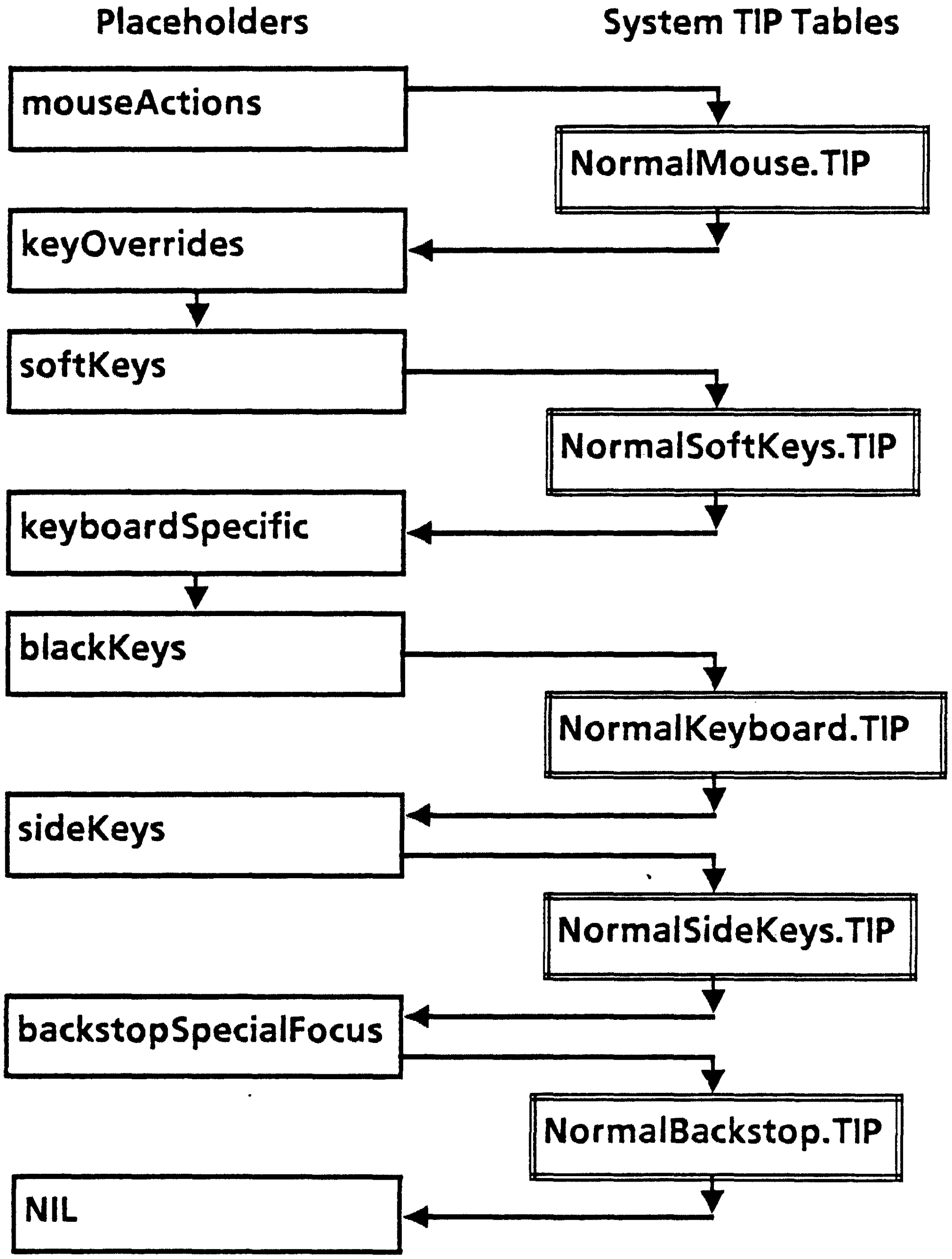
– raised if Prog.TIP exists but is invalid

-- Destroy the specified TIP table

```
DestroyTable: PROC = {TIP.DestroyTable[@table]};
```

Placeholders

ViewPoint provides a chain of empty tables called Placeholders. When the system is booted, a set of normal tables is installed; these tables contain all the basic key actions that most applications need.



TIPStar

The TIPStar interface provides procedures for manipulating the chain of TIP tables. Tables can be installed or removed from the chain of existing tables. It is often desirable to install a private table when the user clicks Point in an application's body window; similarly, the table can be removed when the mouse exits the window. In this fashion, your application can have special interpretations for actions and not affect other applications.

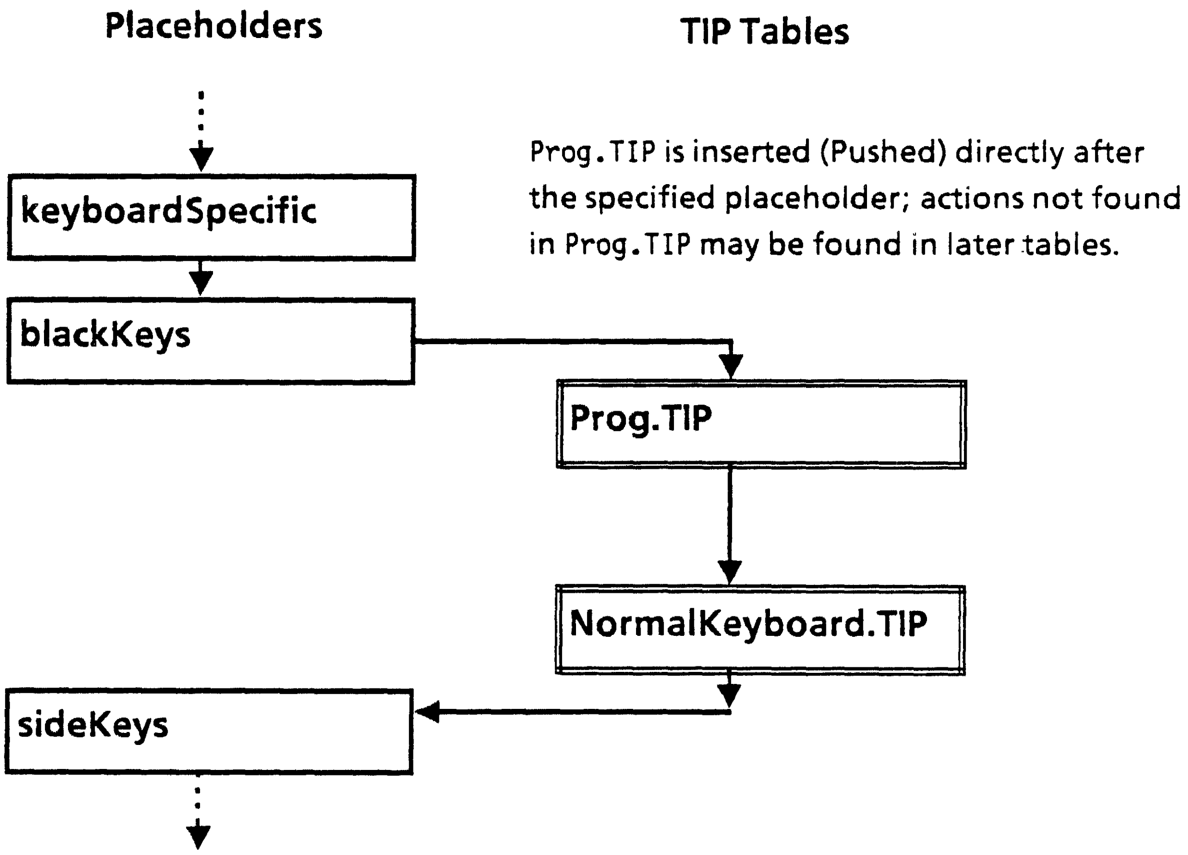
```
TIPStar.Placeholder: TYPE = {
  mouseActions, keyOverrides, softKeys, keyboardSpecific,
  blackKeys, sideKeys, backstopSpecialFocus};
```

```
SetUpTipTable: PUBLIC PROC[body: Window.Handle] = {
  fileName: XString.ReaderBody ← XString.FromSTRING["Prog.TIP"L];
  contents: XString.ReaderBody ← XString.FromSTRING["
  SELECT TRIGGER FROM
    S Down => TurnLeft;
    D Down => TurnRight;
  ENDCASE...
"L];
  table ← TIP.CreateTable[...]};

MyNotifyProc: TIP.NotifyProc = {
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      atom =>
      SELECT z.a FROM
        pointDown => IF NOT pushed THEN {
                       TIPStar.PushTable[blackKeys, table];
                       TIP.SetInputFocus[w: window, takesInput: TRUE];
                       pushed ← TRUE};
        exit => IF pushed THEN {
                TIPStar.PopTable[blackKeys, table];
                pushed ← FALSE};
        turnLeft => Defs.TurnLeft[goodGuy, window];
        turnRight => Defs.TurnRight[goodGuy, window];
        ENDCASE;
      ENDCASE;     -- WITH z: input
    ENDLOOP};
```

Pushing and Popping Tables

Let's examine the effects of clicking Point and EXITing the body window in the previous example. Clicking Point in the body window pushes a private TIP table, which passes the atom **TurnLeft** when the key **S** is pressed and the atom **TurnRight** when the **D** key is pressed. If a private table were not inserted into the chain of tables, a text character would have been passed to whichever window had the input focus.

When the mouse EXITs the body window, Prog.TIP is popped off the chain and the keys **S** and **D** go back to their normal interpretation.



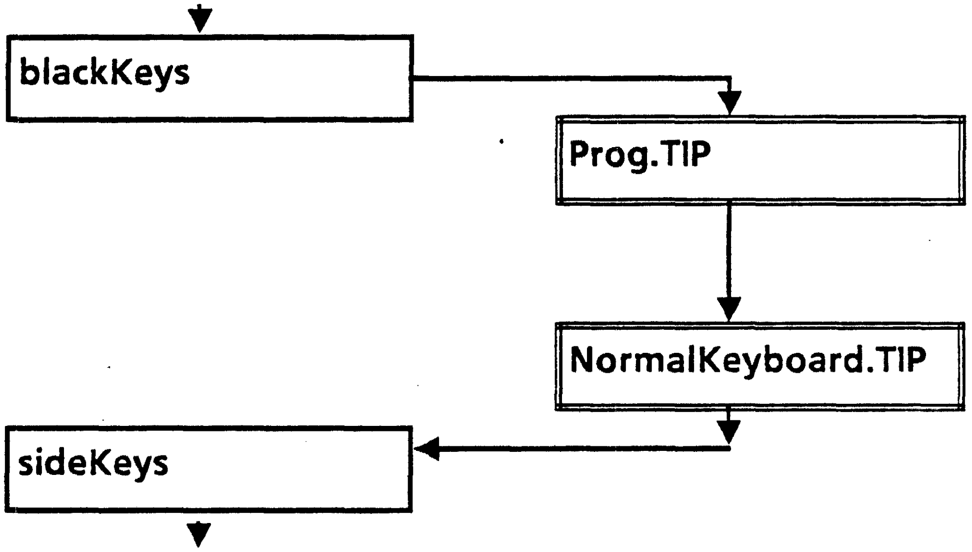
Prog.TIP is inserted (Pushed) directly after the specified placeholder; actions not found in Prog.TIP may be found in later tables.

Storing Tables

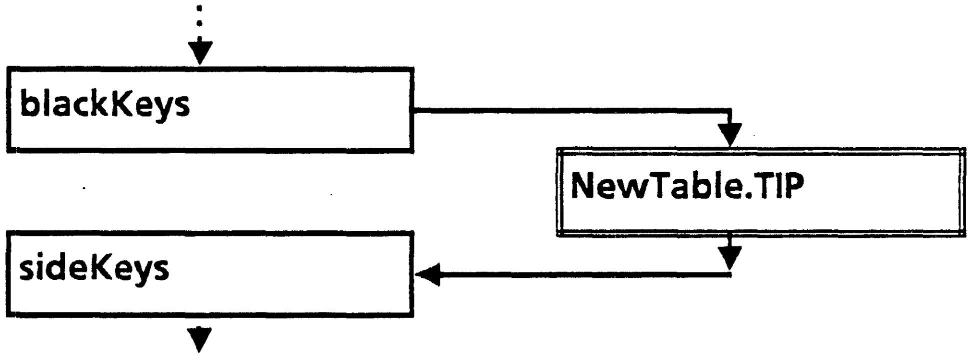
You can also replace all tables following a placeholder with a new TIP table(s) by calling `TIPStar.StoreTable`. StoreTable returns the replaced table(s) so the client can restore them later.

```
TIPStar.StoreTable: PROCEDURE[TIPStar.Placeholder, TIP.Table]
  RETURNS[TIP.Table];
```

Before NewTable.TIP stored



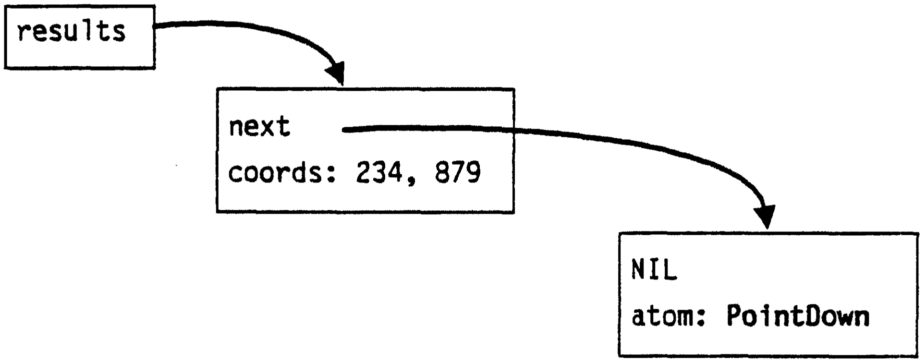
Both Prog.TIP and NormalKeyboard.TIP have been replaced by NewTable.TIP



Results Lists

After the Notifier has found a user action in a TIP table, it passes the results on the RHS of the table to the application's NotifyProc. It is then up to the NotifyProc to interpret the results and perform the desired actions. The results are in the form of a linked list of variant records.

```
TIP.Results: TYPE = LONG POINTER TO
   --READONLY-- TIP.ResultObject;

TIP.ResultObject: TYPE = RECORD[
  next: TIP.Results,
  body: SELECT type: * FROM
    atom => [a: ATOM],                     -- constant in table
    bufferedChar => NULL,                  -- not seen by NotifyProcs
    coords => [place: Window.Place],--window relative
    int => [i: LONG INTEGER],              -- constant in table
    key => [key: KeyName, downUp: DownUp],
    nop => NULL,                           -- not seen by NotifyProcs
    string => [rb: XString.ReaderBody],-- char or constant
    time => [time: System.Pulses],
  ENDCASE];
```



Atoms

An atom is a one-word object that corresponds to a textual string. By passing atoms, objects may be named textually without paying the expense of actually storing, copying, and comparing the strings themselves. TIP tables often contain text strings that correspond to atoms on their right hand sides; these strings are converted into one word atoms and are passed into the clients NotifyProcs as atom variants.

Your application must declare atoms that correspond exactly to the ones in the TIP tables. You create an atom by calling **Atom.MakeAtom**, passing a Mesa string that is equivalent to the one in the TIP table.

```
Atom.ATOM: TYPE[1];

Atom.null: ATOM = LOOPHOLE[0];

Atom.MakeAtom: PROCEDURE[pName: LONG STRING]
  RETURNS[atom: Atom.ATOM];
```

NotifyProcs

The NotifyProc interprets only those results it is interested in and ignores the rest. Typically, the NotifyProc will use the results to modify the application's data and to call various procedures.

```
mouse, pointMotion, enter, exit, delete: Atom.ATOM ← Atom.null;

InitAtoms: PROC = {
  mouse ← Atom.MakeAtom["Mouse"L];
  pointMotion ← Atom.MakeAtom["PointMotion"L];
...
  };

MyNotifyProc: TIP.NotifyProc = {
<<[window: Window.Handle, results: Results]>>
  data: Defs.Data ← Defs.GetContext[window];
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      coords => data.place ← z.place;
      atom =>
      SELECT z.a FROM
        mouse => IF data.lastMouse = neither THEN DisplayBitmap[window,data]
                 ELSE PointMotion[window, data];
        pointMotion => PointMotion[window, data];
        enter => EnterWindow[window, data];
        exit => ExitWindow[window, data];
        delete => Defs.Delete[window, data];
      ENDCASE;
    ENDCASE; -- WITH z: input
  ENDLOOP;
  };
```

NotifyProcs (cont'd)

You can associate your NotifyProc with a chain of TIP tables by calling **TIP.SetTableAndNotifyProc**; this would usually follow the table creation code. The Notifier will check your table before searching the default NormalKeyboard.TIP table. If you push your table, **ALL** applications will "see" this table, so you must pop the table before other applications are invoked.

The call to **TIPStar.NormalTable** returns the table at the head of the chain of TIP tables; this is the appropriate table to use when calling SetTableAndNotifyProc. StarWindowShell.CreateBody will implicitly make this call to SetTableAndNotifyProc.

```
SetUpTIPTable: PROC[body: Window.Handle]= {
  table ← TIP.CreateTable[...];
  TIPStar.PushTable [blackKeys, table];
  TIP.SetTableAndNotifyProc [
    window: body,
    table: TIPStar.NormalTable[],
    notify: MyNotifyProc];
  };
```

Input Focus

Mouse actions are typically sent to the window containing the cursor, but keystrokes are sent to the window containing the input focus. Since you will often want to interpret keystrokes, you will need to know how to set the input focus. Other applications will take the the input focus after the cursor leaves your body window (i.e. when the user selects an icon).

```
TIP.SetInputFocus: PROCEDURE[
  w: Window.Handle,
  takesInput: BOOLEAN,
  newInputFocus: TIP.LosingFocusProc ← NIL,
  clientData: LONG POINTER ← NIL];

TIP.LosingFocusProc: TYPE = PROCEDURE[
  w: Window.Handle, data:LONG POINTER];
```

```
MyNotifyProc: TIP.NotifyProc = {
  data: Defs.Data ← GetContext[window];
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      atom => SELECT z.a FROM
        pointDown => {TIPStar.PushTable [blackKeys, table];
          TIP.SetInputFocus[
           w: window,
           takesInput: FALSE,
           newInputFocus: LostInputFocus,
             clientData: data]};
  ...
  };

LostInputFocus: TIP.LosingFocusProc = {
<<PROC[w: Window.Handle, data: LONG POINTER]>>
  <<perform actions when window loses input focus>>
  };
```

Periodic Notification

You may have a background process that needs to perform some operation in the Notifier process (such as obtaining the current selection). If your program was not running in the Notifier, then the user, or some other application, could change the current selection in the middle of your operation. By calling TIP.CreatePeriodicNotify, you can have your NotifyProc called at set intervals from within the Notifier process.

When you no longer need to be notified, you can cancel the periodic notification mechanism.

```
TIP.PeriodicNotify: TYPE[1];

TIP.CreatePeriodicNotify: PROC[
  window: Window.Handle,
  results: TIP.Results,
  milliSeconds: CARDINAL,
  notifyProc: TIP.NotifyProc ← NIL]
  RETURNS[TIP.PeriodicNotify];

TIP.CancelPeriodicNotify: PROC[TIP.PeriodicNotify]
  RETURNS[null: TIP.PeriodicNotify];
```

Periodic Notification (cont'd)

This sample code registers a NotifyProc that displays the current time every second. While this type of operation does not need to be performed in the Notifier, it is simple to understand.

```
results: TIP.ResultObject;
periodicNotify: TIP.PeriodicNotify ← TIP.nullPeriodicNotify;
globalPlace: Window.Place;

MyNotifyProc: TIP.NotifyProc = {
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      atom => SELECT z.a FROM
        currTime => {
          rb: XString.ReaderBody ← GetCurrentTime[];
          [] ← SimpleTextDisplay.StringIntoWindow[string: @rb,...]};
        ENDCASE;
      ENDCASE; -- WITH z: input
    ENDLOOP;
  };

-- The "results" object must continue to exist after the call to
-- CreatePeriodicNotify or the terrible things will happen.
StartTime: MenuData.MenuProc = {
  results ← [next: NIL, body: atom[a: currTime]];
  periodicNotify ← TIP.CreatePeriodicNotify[
    window: StarWindowShell.GetBody[[window]],
    results: @results,
    milliSeconds: 1000,
    notifyProc: MyNotifyProc];
  };

StopTime: MenuData.MenuProc = {
  periodicNotify ← TIP.CancelPeriodicNotify[periodicNotify];
  };
```

Managers

TIP managers are used to send *all* user input through a specified table and NotifyProc rather than through the window containing the cursor or input focus. Managers, in effect, override the normal behavior of the system and are usually in use for only a short time. For example, an application that wishes to capture a bitmap image of the root window might create a manager to avoid having objects selected during the capture.

```
TIP.Manager: TYPE = RECORD[
  table: TIP.Table,
  window: Window.Handle,
  notify: TIP.NotifyProc];

TIP.GetManager: PROCEDURE RETURNS[current: TIP.Manager];

TIP.ClearManager: PROCEDURE = INLINE ...;

TIP.SetManager: PROCEDURE[new: TIP.Manager]
  RETURNS[old: TIP.Manager];
```

Manager Example

This example contains excerpts from a program that copies a bitmap from the display. If no manager were used, MyNotifyProc would not be called for actions that occured over another window.

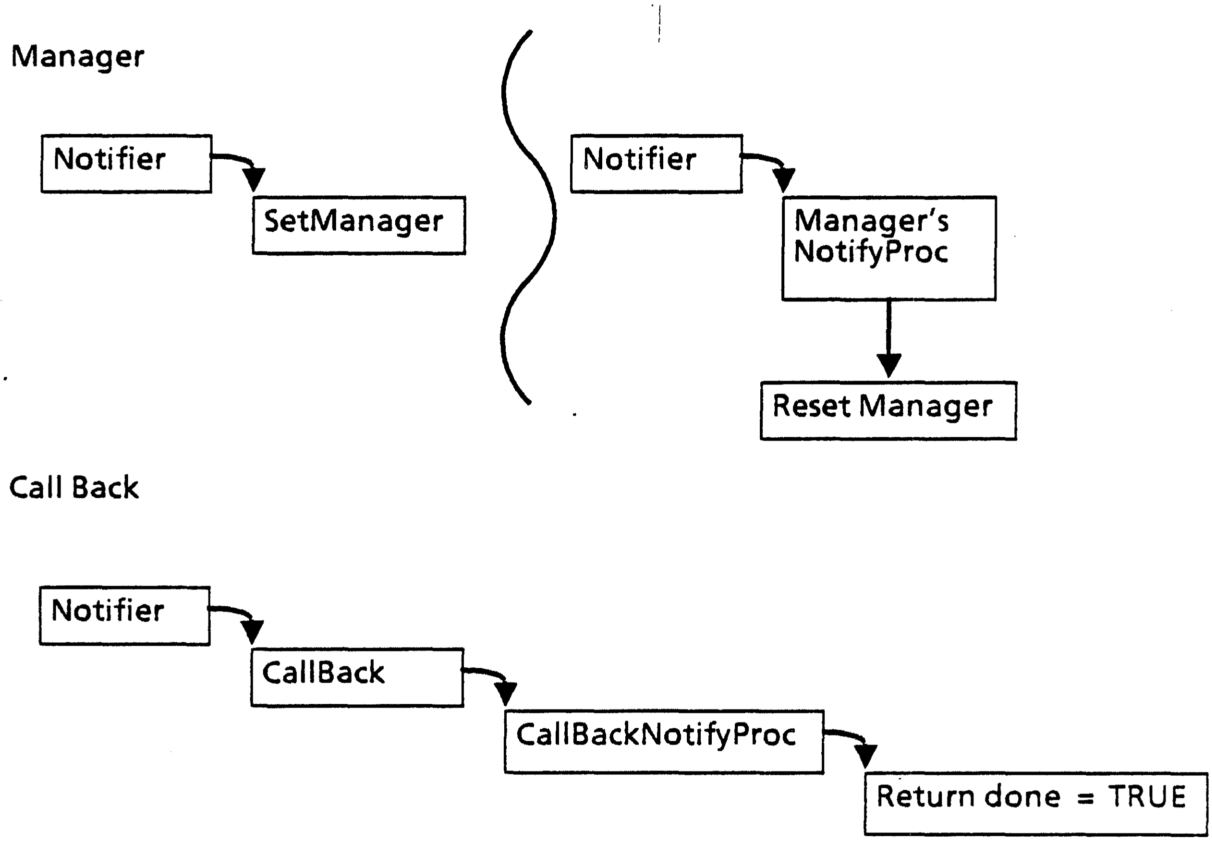
```
MyManager, Old: TIP.Manager ← TIP.nullManager;
table: TIP.Table ← InitTable[];

MyNotifyProc: TIP.NotifyProc = {
  place: Window.Place;
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      coords => place ← z.place;
      atom => SELECT z.a FROM
        confirm => ConfirmBox[window, place];
        start => StartBox[window, place];
        ENDCASE;
      ENDCASE; -- WITH z: input
    ENDLOOP;
  };

-- Called to invoke the manager
Start: MenuData.MenuProc = {
  body: Window.Handle ← StarWindowShell.GetBody[[window]];
  MyManager ← [table: table, window: body, notify: MyNotifyProc];
  Old ← TIP.SetManager[new: MyManager];
  };

-- Saves the upper left corner to the box
StartBox: PROC[window:Window.Handle, place: Window.Place] = {
  data: Defs.Data ← Defs.GetContext[window];
  data.upperLHCorner ← place;
  };

-- Copies the bitmap specified by the upper left corner and the current position
ConfirmBox: PROC[window:Window.Handle, place: Window.Place] = {
  CopyBitmap[window, place];
  [] ← TIP.SetManager[new: Old];
  };
```

Call-Back Notification

Call-back notification is similar to having a manager except that the client's call stack is not unwound. All input that matches the client's TIP table is directed to the `TIP.CallBackNotifyProc`.

```
TIP.CallBack: PROCEDURE[
  window: Window.Handle,
  table: TIP.Table,
  notify: TIP.CallBackNotifyProc];
```

```
TIP.CallBackNotifyProc: TYPE = PROCEDURE[
  window: Window.Handle, results: TIP.Results]
  RETURNS[done: BOOLEAN];
```



Comparison of Managers and Call-Back Notification

Call-Back Notification Example

MyNotifyProc is associated with the application's body window and starts the call-back notification when the user clicks Point. The Start procedure does not return until the **cancel** or **confirm** atom is encountered and the **TIP.CallBackNotifyProc** returns **TRUE**.

```
MyNotifyProc: TIP.NotifyProc = {
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      atom => SELECT z.a FROM
                point => Start[window];
                ENDCASE;
      ENDCASE; -- WITH z: input
    ENDLOOP;
  };

MyCallBackNotifyProc: TIP.CallBackNotifyProc = {
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      atom => SELECT z.a FROM
        start =>   StartBox[window,TIP.GetPlace[StarDesktop.GetWindow[]]];
        confirm => {ConfirmBox[window,TIP.GetPlace[StarDesktop.GetWindow[]]];
                   RETURN[TRUE]};
        cancel =>  {CancelBox[window]; RETURN[TRUE]};
        ENDCASE;
      ENDCASE; -- WITH z: input
    ENDLOOP;
  RETURN[FALSE];
  };

Start: PROC[window:Window.Handle] = {
  TIP.CallBack[window: window, table: table, notify: MyCallBackNotifyProc];
  };
```

User Abort

There are two methods in which your program can recognize the user pressing the STOP key: notification by call-backs or by checking an abort bit. The advantage of call-back procs is that you don't have to keep checking the abort bit; however, the call-back proc must be able to stop the application. By polling the abort bit, you will be able to stop the process more easily at the cost of higher overhead. We will discuss call-backs first.

```
TIP.SetAttention: PROCEDURE[
  Window.Handle, attention: TIP.AttentionProc];
TIP.AttentionProc: TYPE = PROCEDURE[window: Window.Handle];
```

```
continue: BOOLEAN ← FALSE;

Count: PROCEDURE[...] = {
  WHILE continue DO
    << do some work >>
  ENDLOOP};

-- This proc is called when the user presses the STOP key
MyAttentionProc: TIP.AttentionProc = {continue ← FALSE};

-- Begin recognizing user aborts with a call-back proc
SetAttention: PROC[body: Window.Handle] = {
  TIP.SetAttention[window: body, attention: MyAttentionProc]};

Start: MenuData.MenuProc = {
  continue ← TRUE;
  Process.Detach[FORK Count[...]]};
```

User Abort (cont'd)

Here's another version of the tool that uses polling instead of call-backs. You can programmatically reset the abort bit; otherwise, it is reset with the next keystroke.

```
-- This procedure must periodically call the UserAbort procedure
Count: PROCEDURE[body: Window.Handle] = {
  WHILE NOT TIP.UserAbort[body] DO
    << do some work >>
  ENDLOOP;
  };

-- Set user abort flag for the window to FALSE
ResetAbort: MenuData.MenuProc = {
  body: Window.Handle ← StarWindowShell.GetBody[[window]];
  TIP.ResetUserAbort[body];
  };

-- Enable user aborts for specified window
SetAbort: MenuData.MenuProc = {
  body: Window.Handle ← StarWindowShell.GetBody[[window]];
  TIP.SetUserAbort[body];
  };

Start: MenuData.MenuProc = {
  Process.Detach[FORK Count[StarWindowShell.GetBody[[window]]]];
  };
```

Character Translation

When any of the black keys on the keyboard are pressed, the result generated by the system NormalKeyboard.TIP table is a BUFFEREDCHAR. If you want to construct a character from this keystroke, you will need to write a character translator procedure. A character translator is a call-back proc that is called by TIP whenever a result of type BUFFEREDCHAR is encountered, and it can map the key to whatever character the client chooses.

```
TIP.CharTranslator: TYPE = RECORD[
  proc: TIP.KeyToCharProc,
  data: LONG POINTER];

TIP.KeyToCharProc: TYPE = PROCEDURE[
  keys: LONG POINTER TO TIP.KeyBits,
  key: TIP.KeyName,
  downUp: TIP.DownUp,
  data: LONG POINTER,
  buffer: XString.Writer];

TIP.KeyBits: TYPE = LevelIVKeys.KeyBits;
LevelIVKeys.KeyBits: TYPE =
  PACKED ARRAY LevelIVKeys.KeyName OF LevelIVKeys.DownUp;
LevelIVKeys.DownUp = KeyboardWindow.KeyStations.DownUp;

TIP.KeyName: TYPE = LevelIVKeys.KeyName;
LevelIVKeys.KeyName: TYPE = MACHINE DEPENDENT {
  ...Four, Six, E, Seven,...};

TIP.SetCharTranslator: PROCEDURE[
  table: TIP.Table,
  new: CharTranslator]
  RETURNS[old: CharTranslator];
```

Character Translation (cont'd)

In order to translate a **BUFFEREDCHAR** you must either map the current key pressed (**key**) or use the current state of the keyboard (determined by the value **keys**) to determine what character to generate. If you only want to map the current key, all you need to do is perform a simple translation between the enumerated value **key**, and the desired character in some character set. If you want to take into account multiple key positions (e.g. **PROP'S** down and **Q** down) then you will have to use the **keys** argument.

After generating a character(s) within your **TIP.KeyToCharProc**, you should append them to the buffer passed in. The buffer will be passed to your **TIP.NotifyProc**, as a **string** variant, at which time it can be added to the data structure of your application.

```
SomeKeyToCharProc: TIP.KeyToCharProc = [
<<keys: LONG POINTER TO TIP.KeyBits,...buffer: XString.Writer]>>
  char: XChar.Character;
  code: XCharSet0.Codes0;
  IF keys[LeftShift] = down OR keys[RightShift] = down THEN
    code ← SELECT key FROM
      A => upperA,
      ...
      ENDCASE => upperZ
  ELSE
    code ← SELECT key FROM
      A => lowerA,
      ...
      ENDCASE => lowerZ;
  char ← XCharSet0.Make[code];
  XString.AppendChar[buffer, char];
  };
```

Character Translation Example

```
oldTrans: TIP.CharTranslator ← [proc: NIL, data: NIL];

MyKeyToCharProc: TIP.KeyToCharProc = {
<<keys: LONG POINTER TO TIP.KeyBits,...buffer: XString.Writer]>>
  code: XCharSet0.Codes0 ← SELECT key FROM
    A => upperA,
    ...
    ENDCASE => upperZ;
  char: XChar.Character ← XCharSet0.Make[code];
  XString.AppendChar[buffer, char];
  };

NotifyProc: TIP.NotifyProc = {
<<PROC[window: Window.Handle, results: TIP.Results]>>
  data: Defs.Data ← Defs.GetContext[window];
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      string => {
        XString.AppendReader[to: @data.wb, from: @z.rb!
          XString.InsufficientRoom => {
            XString.ExpandWriter[w: @data.wb, extra: 20]; RETRY}];
            DisplayText[window]};
      atom => SELECT z.a FROM
        enter => Enter[window];
        exit => Exit[window];
        ENDCASE;
      ENDCASE; -- WITH z: input
    ENDLOOP;
  };

-- Establish a new translator for this window and save the old one
Enter: PROC[window:Window.Handle] = {
  translator: TIP.CharTranslator ← [proc: MyKeyToCharProc, data: NIL];
  TIP.SetInputFocus[w: window, takesInput: TRUE];
  oldTrans ← TIP.SetCharTranslator[
    table:TIPStar.GetTable[TIPStar.Placeholder.blackKeys], new: translator];
  };

-- restore old translator when mouse exits window
Exit: PROC[window:Window.Handle] = {
  [] ← TIP.SetCharTranslator[
    table: TIPStar.GetTable[TIPStar.Placeholder.blackKeys], new: oldTrans];
  };
```

Stuffing Input into a Window

TIP provides procedures that drive the type-in mechanism as if the characters were coming from the user. It is possible to stuff single characters, XStrings, Mesa STRINGs, and the current selection. When these objects are stuffed, your TIP.NotifyProc is called with a result of type string.

```
MyNotifyProc: TIP.NotifyProc = {
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      string => {
       XString.AppendReader[to: @wb, from: @z.rb! XString.InsufficientRoom =>
         {XString.ExpandWriter[w: @wb, extra: 20]; RETRY}];
          Window.InvalidateBox[window: window, box: [[0,0],[1000,1000]]];
          Window.Validate[window: window]};
      atom => SELECT z.a FROM
               enter => Enter[window];
               exit => Exit[window];
               ENDCASE;
      ENDCASE; -- WITH z: input
    ENDLOOP;
  };

StuffChar: MenuData.MenuProc = {
  body: Window.Handle ← StarWindowShell.GetBody[[window]];
  [] ← TIP.StuffCharacter[window: body, char: XCharSet0.Make[upperH]];
  };

StuffSelection: MenuData.MenuProc = {
  body: Window.Handle ← StarWindowShell.GetBody[[window]];
  [] ← TIP.StuffCurrentSelection[window:body];
  };

StuffString: MenuData.MenuProc = {
  body: Window.Handle ← StarWindowShell.GetBody[[window]];
  rb: XString.ReaderBody ← XString.FromSTRING["sample string"L];
  [] ← TIP.StuffString[window: body, string: @rb];
  };
```

Day 2: Selection

Outline

I. Selection
   A. Classes of selection clients
   B. Converting a selection
      1. Standard targets
      2. Unique targets
      3. Valid target determination
      4. Enumeration
   C. Copying and Moving a selection
   D. Managing the current selection
      1. ConvertProc
      2. ActOnProc
      3. ValueProcs
         a. ValueFreeProc
         b. ValueCopyMoveProc
   E. Saving a selection

Selection

The Selection interface defines the abstraction that is the user's "current selection."

Selection provides procedures that allow someone (other than the originator of the selection) to request information relating to the selection and get this information in a particular format.

Selection

There are two classes of clients who use the Selection interface.

Requestors only want to obtain the value of the current selection in some format.

Managers want to modify the current selection (i.e., change what it is that's selected) and thus require a thorough knowledge of Selection.

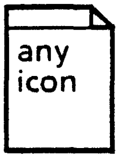
Selection

The following chart shows some examples of selection situations. Who're the requestor and manager in each situation?

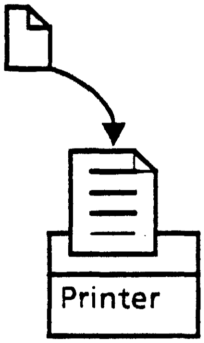
Situation	Requestor	Manager
User selects text in a document		
User drops a document (that was on the desktop) onto a file drawer		
Application takes the current icon selection & loads its contents into a window		

Selection Example

The goal of Selection is for the requestor to never care what program is managing the selection. All that matters is whether the selection can be put in a proper form or not.



The user copies or moves an icon and drops it on a printer.



The printer implementation only cares whether the selection can be converted to an Interpress Master.

Converting a Selection

The most common operation performed by a selection requestor is to get the value of the current selection by calling Selection.Convert:

```
Selection.Convert: PROCEDURE[
   target: Selection.Target,
   zone: UNCOUNTED ZONE ← NIL]
   RETURNS[value: Selection.Value];

Selection.Target: TYPE = MACHINE DEPENDENT {
  window(0), shell,subwindow, string, length, position,
  integer, interpressMaster, file, fileType, token, help,
  interscriptScript, interscriptFragment, serializedFile,
  name, firstFree, last(1777B)};

Selection.Value: TYPE = RECORD[
   value: LONG POINTER,
   ops: LONG POINTER TO Selection.ValueProcs ← NIL,
   context: LONG UNSPECIFIED ← 0];
```

The Target is the TYPE of data the selection should be converted to. The Value is a RECORD containing a pointer to the converted selection.

always call Selection.Free after .Convert

Example Using Selection.Convert

An important point to remember is that Convert returns a read-only value; you can only inspect it. Also, any storage associated with that value must be freed when you are finished.

```
streamHandle: Stream.Handle ← GetStreamToSomeFile[];
xfo: XFormat.Object;
xfo ← XFormat.StreamObject[streamHandle];

-- NIL is returned if the selection cannot be converted
savedString: Selection.Value ← Selection.Convert[string];
IF savedString.value = NIL THEN {
  Stream.Delete[streamHandle];
  RETURN};
XFormat.Reader[@xfo, LOOPHOLE[savedString.value]];
Stream.Delete[streamHandle];
Selection.Free[@savedString];
```

Freeing a Selection

As you saw on the previous page, the requestor is responsible for freeing the temporary storage allocated in the *manager's* domain by calling Selection.Free. This call invokes a manager-supplied call-back procedure to free the storage. This call-back procedure may be a no-op procedure if no temporary resources were allocated when converting the selection.

```
Selection.Free: PROCEDURE[v: Selection.ValueHandle];
Selection.ValueHandle: TYPE = LONG POINTER TO Selection.Value;
```

Converting a Selection to a New Target

The requestor can create a new Target by using Selection.UniqueTarget. The type associated with each Target is determined by system-wide convention.

```
Selection.UniqueTarget: PROCEDURE RETURNS[Selection.Target];
```

This procedure returns a Target within the range of Selection.Target. You should use private target types judiciously since they severely limit the exchange of information between applications.

```
myTarget: Selection.Target ← Selection.UniqueTarget[];
```

It's important to remember that a new target is useless without a selection manager to implement the target.

Can You Convert the Selection

Not all selections will convert to all target types; thus, you may want to ask if a given selection will convert. To do this, you should call `Selection.CanYouConvert`, which returns a **BOOLEAN** specifying whether the value will convert to the particular target type.

```
-- this type of procedure is generally called from the canYouTakeSelection arm
-- of a Containee.GenericProc

CanITake: PROCEDURE RETURNS[yes: BOOLEAN] = {
  -- take anything that is a string, token, or integer
  RETURN[
    Selection.CanYouConvert[target: string, enumeration: FALSE] OR
    Selection.CanYouConvert[target:integer,enumeration:FALSE] OR
    Selection.CanYouConvert[target: token, enumeration: FALSE]];
  };
```

To determine the difficulty that the manager would have in attempting to convert to the specified target, you can call `Selection.HowHard`.

```
Selection.HowHard: PROCEDURE[
   target: Selection.Target,
   enumeration: BOOLEAN ← FALSE]
   RETURNS[difficulty: Selection.Difficulty];

Selection.Difficulty: TYPE = {
  easy, moderate, hard, impossible};
```

Enumerating Selections

A selection is often a collection of items (several files in a folder) or a single large item that can be split up (e.g, a LONG STRING that can be broken up). A requestor can ask that each item or part of such a selection be converted to some Target by calling **Selection.Enumerate**.

```
-- called from GenericProc when selection Copyed or Moved to icon.
-- This proc appends the current selection to the end of a file
Absorb: PROCEDURE[data: Containee.DataHandle] = {

  AbsorbString: Selection.EnumerationProc = {
  <<[element:Selection.Value, data: Selection.RequestorData]>>
    XFormat.Reader[@xfo, LOOPHOLE[element.value]];
    Selection.Free[@element];
    };

  xfo: XFormat.Object;
  fileStream: NSFileStream.Handle ← GetStream [data];
  Stream.SetPosition[fileStream, NSFileStream.GetLength[fileStream]];
  xfo ← XFormat.StreamObject [fileStream];
  SELECT TRUE FROM
    Selection.CanYouConvert[target: string, enumeration: FALSE] =>
      [] ← AbsorbString[Selection.Convert[string], NIL];
    Selection.CanYouConvert[target: string, enumeration: TRUE] =>
      [] ← Selection.Enumerate[AbsorbString, string, NIL];
  ENDCASE;
  Stream.Delete[fileStream];
  };
```

Copying and Moving the Value of the Selection

Convert and Enumerate return read-only Values. The selection manager owns the storage. The requestor must not alter the value.

If the requestor wants to keep the value or pass the value to another process, it must call Copy, Move, or CopyMove. These three procedures call a manager-supplied procedure that allows the requestor to alter the value of the selection without affecting the selection manager.

If Move is called, the value is also deleted from the manager's storage domain. After a Move or Copy, the requestor owns the Value. The requestor should eventually Free the storage.

```
Selection.Copy: PROCEDURE[
  v: Selection.ValueHandle, data: LONG POINTER] = INLINE {
    Selection.CopyMove[v, copy, data]};

Selection.Move: PROCEDURE[
  v: Selection.ValueHandle, data: LONG POINTER] = INLINE {
    Selection.CopyMove[v, move, data]};

Selection.CopyMove: Selection.ValueCopyMoveProc;

Selection.ValueCopyMoveProc: TYPE = PROCEDURE[
  v: Selection.ValueHandle, op: Selection.CopyOrMove,
    data: LONG POINTER];

Selection.CopyOrMove: TYPE = {copy, move};
```

Copying and Moving the Value of the Selection (cont'd)

After calling `Copy`, `Move`, or `CopyMove`, the requestor then owns `v.value↑` and can alter it.

A requestor may call these procedures after calling `Convert` or from an `EnumerationProc` while doing an `Enumerate`. `data` is passed to the manager; what it points to depends on the particular `Target`. `data` often points to a destination container for the copied value.

don't forget Selection.Free

Managing the Current Selection

The fundamental operation performed by a selection manager is to become the current manager by calling Selection.Set, which takes a ConvertProc, an ActOnProc, and a long pointer (ManagerData).

```
Selection.Set: PROCEDURE[
  pointer: Selection.ManagerData,
  conversion: Selection.ConvertProc,
  actOn: Selection.ActOnProc];
```

The ConvertProc

ConvertProc is called to obtain the value of the selection whenever a requestor calls Selection.Convert or Selection.Enumerate.

It's also called to determine what Targets the selection can be converted to whenever a requestor calls CanYouConvert, Query, or HowHard.

```
Selection.Set: PROCEDURE[pointer: Selection.ManagerData,
  conversion: Selection.ConvertProc, actOn: Selection.ActOnProc];
```

```
Selection.Convert: PROCEDURE[
  target: Selection.Target, zone: UNCOUNTED ZONE ← NIL]
  RETURNS[value: Selection.Value];
```

```
Selection.ConvertProc: TYPE = PROCEDURE[
  data: Selection.ManagerData,     -- LONG POINTER
  target: Selection.Target,        -- type to convert selection to
  zone: UNCOUNTED ZONE,
  info: Selection.ConversionInfo ← [convert[ ]]]
  RETURNS[value: Selection.Value];
```

```
Selection.Value: TYPE = RECORD[
  value: LONG POINTER,
  ops: LONG POINTER TO Selection.ValueProcs ← NIL,
  context: LONG UNSPECIFIED ← 0];
```

The ConvertProc "ConversionInfo"

`ConversionInfo` is a variant record passed to the `ConvertProc` that indicates which operation to perform: convert, enumeration, or query.

```
Selection.ConversionInfo: TYPE = RECORD[
  SELECT type: * FROM
    convert => NULL,
    enumeration => [
      proc: PROC[Selection.Value] RETURNS[stop: BOOLEAN]],
    query => [
      query: LONG DESCRIPTOR FOR ARRAY OF
        Selection.QueryElement],
    ENDCASE];
```

```
Selection.QueryElement: TYPE = RECORD[
  target: Selection.Target, enumeration: BOOLEAN ← FALSE,
  difficulty: Selection.Difficulty ← TRASH];
```

The ConvertProc "value"

When the requestor calls `Selection.Convert`, the manager's `ConvertProc` gets called with `convert ConversionInfo`. If the ConvertProc can convert `data`, the `value.value` returned should point to the converted selection value. Additionally,`value.ops` should point to a record containing a `ValueFreeProc` and a `ValueCopyMoveProc`, and `value.context` can be used as "client data."

If the manager doesn't support conversion to the requested target, the `ConvertProc` should return `nullValue`.

```
Selection.ConvertProc: TYPE = PROCEDURE[
  data: ManagerData, target: Target,
  zone: UNCOUNTED ZONE,
  info: ConversionInfo ← [convert[ ]]]
  RETURNS[value: Value];
Value: TYPE = RECORD[
  value: LONG POINTER,
  ops: LONG POINTER TO ValueProcs ← NIL,
  context: LONG UNSPECIFIED ← 0];
ValueProcs: TYPE = RECORD[
  free: ValueFreeProc ← NIL,
  copyMove: ValueCopyMoveProc ← NIL];
ValueFreeProc: TYPE = PROCEDURE[v: ValueHandle];
ValueCopyMoveProc: TYPE = PROCEDURE[
  v: ValueHandle, op: CopyOrMove, data: LONG POINTER];
CopyOrMove: TYPE = {copy, move};
```

Sample ConvertProc

The following sample ConvertProc supports conversions of the current selection to the targets `string`, `integer`, and `fileType`. Enumeration is not supported.

```
MyConvertProc: Selection.ConvertProc = [
<<[data: Selection.ManagerData, target:Selection.Target, zone:UNCOUNTED ZONE,
info: Selection.ConversionInfo] RETURNS[value: Selection.Value]>>

  string: XString.Reader ← NARROW[data, XString.Reader];
  WITH i: info SELECT FROM
    query =>  {FOR x: CARDINAL IN [0..LENGTH[i.query]) DO
                 i.query[x].difficulty ←
                   IF i.query[x].enumeration THEN impossible
                   ELSE SELECT i.query[x].target FROM
                          string => easy,
                          integer, fileType => moderate,
                          ENDCASE => impossible;
                 ENDLOOP;
               RETURN[Selection.nullValue]};
    convert => SELECT target FROM
                 string => << do something here >>
                 integer, fileType =>  << do something here >>
               ENDCASE => RETURN[Selection.nullValue];
    ENDCASE => RETURN[Selection.nullValue];
  ];
```

The ActOnProc

`ActOnProc` is called to perform various `Actions` on the selection, such as `mark`, `unmark`, and `clear`.

```
Selection.Set: PROCEDURE[pointer: Selection.ManagerData,
  conversion: Selection.ConvertProc, actOn: Selection.ActOnProc];
```

```
Selection.ActOnProc: TYPE = PROCEDURE[
  data: Selection.ManagerData,   -- LONG POINTER (usually points to a record)
  action: Selection.Action]      -- Perform some action on current selection
  RETURNS[cleared: BOOLEAN];     -- should be TRUE if selection was cleared
```

Actions the ActOnProc Can Perform

```
Selection.Action: TYPE = MACHINE DEPENDENT {
  clear(0), mark, unmark, delete, clearIfHasInsert, save,
  restore, firstFree, last(255)};
```

`clear` unselects the current selection by freeing any associated private data, undoing TIP notification changes, etc.

`mark` highlights the current selection; if the selection is already highlighted, this is a no-op.

`unmark` dehighlights the current selection; if the selection is already not highlighted, this is a no-op.

`delete` deletes the contents of the current selection. The selection manager may decide against actually deleting it.

`clearIfHasInsert` same as unmark plus clear, but only if the input focus is in the selection. This action is used when a secondary selection has been completed (for copy-from); if the place to which the secondary selection is to be copied (the input focus) is within the selection itself, the selection is cleared after obtaining its contents and before the insertion takes place.

`save` unselects the current selection, but does not necessarily free any associated private data since the selection is expected to be restored later. This action will often be a no-op, but the manager might need to undo a special TIP notifier, for example.

`restore` restores a previously saved selection.

`firstFree` is used internally by `UniqueAction` and should not be used by clients.

Sample ActOnProc

The following sample ActOnProc shows a basic format for handling actions:

```
MyActOnProc: Selection.ActOnProc = {
<<[data: Selection.ManagerData, action:Selection.Action] RETURNS[cleared: BOOLEAN]>>

  string: XString.Reader ← NARROW[data, XString.Reader];
  SELECT action FROM
    mark =>    << highlight current selection &/or set data >>;
    unmark =>  << dehighlight current selection &/or set data >>;
    clear =>  cleared ← TRUE;
    save =>   NULL;
    ENDCASE => {mh: XMessage.Handle ← Defs.GetMessageHandle[];
                msg: XString.ReaderBody ←
                  XMessage.Get[mh, Defs.kunknownAction];
                Attention.Post[@msg]};
  };
```

The ManagerData

The `ManagerData` passed to `Set` is passed back to the `ConvertProc` and the `ActOnProc`. Typically, `ManagerData` identifies exactly what portion of the manager's domain is currently selected. For example, if the current selection is some text in a document, the actual manager is the document application, which has some `ManagerData` that indicates exactly which characters are currently selected.

When a manager calls `Selection.Set`, the previous manager is told to `ActOn[clear]` and Selection forgets about the previous manager. *Hence, there is only one selection at a time.*

Other Selection Procedures

```
Selection.Clear: PROCEDURE[unmark: BOOLEAN ← TRUE];
```

Clear clears the current selection. The only time unmark should be FALSE is if the caller knows that the area of the screen containing the selection is going to be repainted soon anyway.

```
   Clear[unmark: TRUE] is equivalent to
      { ActOn[unmark]; ActOn[clear] };
```

```
Selection.ActOn: PROCEDURE[action: Selection.Action];
```

ActOn asks the manager to perform the indicated action.

ValueProcs

What happens when a requestor calls `Selection.Copy` or `Selection.Free`? It's highly dependent on what the selection is and the type to which that selection was converted. So that the requestor doesn't have to worry about these details, the selection manager provides ValueProcs.

The `Value` produced by a manager's `ConvertProc` contains a pointer to two procedures, a `ValueFreeProc` and a `ValueCopyMoveProc`. The `ValueFreeProc` is called when a requestor calls `Selection.Free` so that the manager can release any resources that were allocated when the selection was converted. The manager's `ValueCopyMoveProc` is called when the requestor calls `Copy`, `Move`, or `CopyMove`. The `ValueCopyMoveProc` should copy or move the converted selection value so that the manager no longer owns the resources associated with the value. `context` may be used to store data for the `ValueFreeProc` and the `ValueCopyMoveProc`.

ValueProcs (cont'd)

If the converted selection value can be copied or moved, the manager must return a `ValueCopyMoveProc` with the `Value`. For example, the `Targets` `string` and `file` can be moved or copied, while it doesn't make sense to move or copy the `Targets` `window` and `fileType`. The `ValueCopyMoveProc` modifies the `Value` such that the requestor may then make changes to `value↑` without affecting the selection manager. If a `Move` is performed, the item is also deleted from the manager's domain. The interpretation of the data given to a `ValueCopyMoveProc` depends on the manager; a typical use is to specify a destination for the object.

```
Selection.ValueHandle: TYPE = LONG POINTER TO Selection.Value;

Selection.Value: TYPE = RECORD[
  value: LONG POINTER,                    -- pointer to the current selection
  ops: LONG POINTER TO Selection.ValueProcs ← NIL,
  context: LONG UNSPECIFIED ← 0];
```

ValueFreeProc

```
Selection.ValueFreeProc: TYPE = PROCEDURE[
  v: Selection.ValueHandle];
```

If any resources were allocated to produce the converted selection value, they should be released in the manager's `ValueFreeProc` when the requestor calls `Selection.Free`. The parameter `v` is a long pointer to the `Value` that represents the converted selection.

```
Selection.FreeStd: Selection.ValueFreeProc;
```

This procedure assumes that `v.context` is a zone and that `v.value` can be freed by calling `v.context.FREE[@v.value]`. `Free` calls `FreeStd` when the Value has `ops = NIL` or `ops.free = NIL`.

```
Selection.NopFree: Selection.ValueFreeProc;
```

This procedure should be used as the `ops.free` for a `Value` involving no temporary storage owned by the selection manager.

ValueCopyMoveProc

```
Selection.ValueCopyMoveProc: TYPE = PROCEDURE[
  v: Selection.ValueHandle,
  op: Selection.CopyOrMove,
  data: LONG POINTER];

Selection.CopyOrMove: TYPE = {copy, move};
```

The procedures Copy, Move, and CopyMove invoke the Value's ValueCopyMoveProc, which should modify Value so that it no longer involves manager-owned storage. If the requestor calls Move, then the item is also deleted from the manager's domain. data is the data parameter that the requestor passed to Copy, Move, or CopyMove, and is often a container for the copied value.

For managers that want to support Copy but not Move (i.e., the manager refuses to delete the selection), an attempt to Move should raise Selection.Error[invalidOperation].

If the operation is permitted but nevertheless fails (e.g., due to an NSFile error), Selection.Error[operationFailed] should be raised.

Save and Set

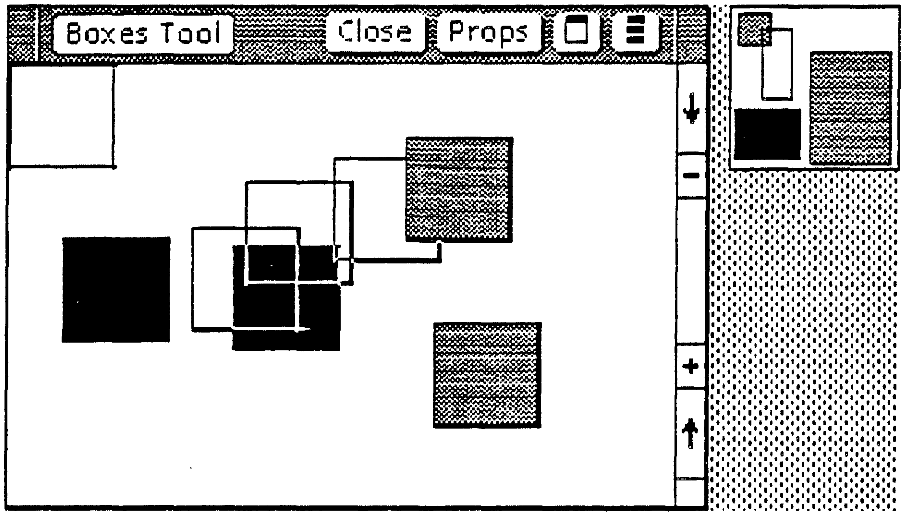
Selection supports the notion of a "saved selection." A client can become the current manager by calling Selection.SaveAndSet, which does a Set but also saves the previous selection. Later, the manager that did the SaveAndSet can do a Selection.Restore to restore the previous selection.

```
Selection.SaveAndSet: PROCEDURE[
  pointer: Selection.ManagerData,
  conversion: Selection.ConvertProc,
  actOn: Selection.ActOnProc, unmark: BOOLEAN ← TRUE]
  RETURNS[old: Selection.Saved];

Selection.Saved: TYPE [6];

Selection.Restore: PROCEDURE[
  saved: Selection.Saved, mark, unmark: BOOLEAN ← TRUE];

Selection.Discard: PROCEDURE[
  saved: Selection.Saved, unmark: BOOLEAN ← TRUE];
```

SaveAndSet is the same as Set except that the existing selection, if there is one, is told to ActOn[save] rather than ActOn[clear]. That is, the existing selection is expected to retain any private state so that it can later be restored via Selection.Restore. If it subsequently turns out that the saved selection is never going to be restored, it should be given to Selection.Discard so that the former selection manager will have a chance to discard any associated private data. A saved selection must always be eventually given to either Restore or Discard; furthermore, once that has been done, the Selection.Saved must not be used for anything else.

Selection Example

The following code implements the selection of a unique target ("boxes").



Boxes tool and icon

```
BoxesDefs: DEFINITIONS = {
  z: UNCOUNTED ZONE;
  Shading: TYPE = {white, gray, black};

  SelectionPtr: TYPE = LONG POINTER TO SelectionData;
  SelectionData: TYPE = RECORD [
    window: Window.Handle,   -- window containing the box
    place: Window.Place ← [0, 0],   -- window relative box pos
    dims: Window.Dims ← [40, 40],   -- box dims
    shading: Shading ← white,       -- changed via props command
    marked: BOOLEAN ← FALSE,        -- used by ActOnProc
    zone: UNCOUNTED ZONE ← NIL];    -- zone where rec is allocated

  Data: TYPE = LONG POINTER TO DataObject;
  DataObject: TYPE = RECORD [
    count: CARDINAL ← 0,  -- currently in use (seq is kept compressed)
    boxes: SEQUENCE maxlength: CARDINAL OF SelectionPtr];
  . . .

  }...
```

```
BoxesImpl: PROGRAM = {
  -- We create our own target type
  box: PUBLIC Selection.Target = Selection.UniqueTarget[];

  opsProcs: Selection.ValueProcs ←
    [free: Selection.NopFree, copyMove: CopyMoveBox];

  MyNotifyProc: TIP.NotifyProc = {
  <<[window: Window.Handle, results: Results]>>
    boxNum: CARDINAL;
    place: Window.Place;
    data: Defs.Data ← GetContext [window];
    FOR input: TIP.Results _ results, input.next UNTIL input = NIL DO
      WITH z: input SELECT FROM
        coords => place ← z.place;
        atom => SELECT z.a FROM
          pointDown => {   -- select or deselect a box
            TIP.SetInputFocus[w: window,takesInput: FALSE];
            boxNum ← Defs.OverBox[window, data, place]; -- are we over a box?
            IF boxNum # LAST[CARDINAL] THEN   -- over some box so select it
              Defs.SelectBox[window, data, place, boxNum]
            ELSE Selection.Clear[]}; -- not over a box so deselect current box
          copyDown => Defs.CopyBox[window,data,place];
          moveDown => Defs.MoveBox[window,data,place];
          copy     => Defs.Copy[window];
          move     => Defs.Move[window];
          delete   => Defs.Delete[window];
          ENDCASE;
        ENDCASE; -- WITH z: input
      ENDLOOP;
    };

  -- This proc establishes a new current selection. The call to Set will unmark and clear the old
  -- selection. Set then automatically calls  the ActOnProc with mark for the new selection.
  SelectBox: PUBLIC PROC[
    wh: Window.Handle, data: Defs.Data, place: Window.Place,
    boxNum: CARDINAL] = {
      Selection.Set[
        pointer: data[boxNum],
        conversion: ConvertSelection,
        actOn: ActOnSelection];
    };
```

```
-- mark or unmark the current selection. If the client wants to delete the selection then delete is
-- invoked, which deletes the box from the manager's domain and clears the selection manager.
ActOnSelection: Selection.ActOnProc = {
  <<[data: Selection.ManagerData, action: Selection.Action]
  RETURNS[cleared: BOOLEAN ← FALSE]>>
  selectionData: Defs.SelectionPtr ← NARROW[data, Defs.SelectionPtr];
  SELECT action FROM
    mark => IF ~selectionData.marked THEN
      {InvertBox[selectionData]; selectionData.marked ← TRUE};
    unmark => IF selectionData.marked THEN
      {InvertBox[selectionData]; selectionData.marked ← FALSE};
    delete => IF selectionData.marked THEN
      {InvertBox[selectionData]; selectionData.marked ← FALSE;
       DeleteBox[selectionData];
       RETURN[TRUE]};
    clear => RETURN[TRUE];
    ENDCASE => {
      mh: XMessage.Handle = Defs.GetMessageHandle[];
      msg: XString.ReaderBody ← XMessage.Get [mh, Defs.kunknownAction];
      Attention.Post[@msg]};
  };

InvertBox: PROC[selectionData: Defs.SelectionPtr] = {
  Display.Invert[
    window: selectionData.window,
    box: [selectionData.place,
          [selectionData.dims.w + 1, selectionData.dims.h + 1]]];
  };

ConvertSelection: Selection.ConvertProc = {
  <<[data: Selection.ManagerData, target: TARGET, zone: UNCOUNTED ZONE,
  info: Selection.ConversionInfo] RETURNS [value: Selection.Value]>>
  selectionData: Defs.SelectionPtr ← NARROW[data, Defs.SelectionPtr];
  WITH i: info SELECT FROM
    query => FOR c: CARDINAL IN [0..LENGTH[i.query]) DO
        i.query[c].difficulty ← IF (i.query[c].target = box) OR
                                   (i.query[c].target = window) THEN easy
                                 ELSE impossible
        ENDLOOP;
    convert => {
      SELECT target FROM
        window => RETURN[[
          selectionData.window,Selection.nopFreeValueProcs]];
        box =>
          RETURN[[value: selectionData,
                  ops: @opsProcs,  -- nop free proc
                  context: LOOPHOLE[zone]] ];
        ENDCASE => RETURN[Selection.nullValue]};
    enumeration => RETURN[Selection.nullValue];
    ENDCASE;
  RETURN[Selection.nullValue];
  };
```

doesn't check selection

^ AND (i.query[c].enumeration = FALSE

```
-- If there is a current selection then set cursor to copy; otherwise post a message to the user
Copy: PROC[window: Window.Handle] = {
  val: Selection.Value ← Selection.Convert[box];
  IF val.value = NIL THEN {
    mh: XMessage.Handle = Defs.GetMessageHandle[];
    msg: XString.ReaderBody ← XMessage.Get [mh, Defs.kunknownAction];
    Attention.Post[@msg];
    RETURN};
  Selection.Free[@val]; -- should always call even if it's a nop
  Cursor.Set[copy];      -- set cursor to copy mode
  [] ← TIPStar.SetMode[mode: copy];
  };

-- called if in copy mode and user performs a point down
-- a copy of the box is made and placed at "place"
CopyBox: PROC[
  window: Window.Handle, data:Defs.Data, place: Window.Place] = {
  value: Selection.Value ← Selection.Convert[target: box];
  boxPtr: Defs.SelectionPtr;
  IF value.value = NIL THEN {
    mh: XMessage.Handle = Defs.GetMessageHandle[];
    msg: XString.ReaderBody ← XMessage.Get [mh, Defs.kunknownAction];
    Attention.Post[@msg];
    RETURN};
  Selection.Copy[v: @value, data: data]; -- make copy store in value.value
  boxPtr ← NARROW[value.value, Defs.SelectionPtr]; -- loophole to make usable
  AddToManager[window, data, boxPtr, place, @value]; --add to manager's domain
  Selection.Free[@value];
  Cursor.Set[textPointer];
  [] ← TIPStar.SetMode[mode: normal];
  };

-- when the user copies or moves an object to a new location,
-- it must be added to the manager's data.
AddToManager: PROC[
  window: Window.Handle, data: Defs.Data, boxPtr: Defs.SelectionPtr,
  place: Window.Place, v: Selection.ValueHandle] = {
  data[data.count] ← boxPtr;
  data[data.count].window ← window;  -- the box belongs to the dest window
  boxPtr.place ← place;  -- give copied box a new location
  SelectBox[window, data, place, data.count]; -- set the new current selection
  data.count ← data.count + 1;
  Window.InvalidateBox[
    window: window,
    box: [boxPtr.place, boxPtr.dims]];
  Window.Validate[window];
  };
```

```
-- make copy of the current selection and store it in v.value. If the call was a copy then simply return.
-- If the call was a move then delete the information from the manager's domain storage is used
  CopyMoveBox: Selection.ValueCopyMoveProc = {
    <<[v: Selection.ValueHandle, op: Selection.CopyOrMove, data: LONG POINTER]>>
    z: UNCOUNTED ZONE ← LOOPHOLE[v.context];
    old: Defs.SelectionPtr ← v.value;   -- keep pointer to manager's object
    -- allocate another box element that will be returned to caller
    new: Defs.SelectionPtr ← v.value ← z.NEW[Defs.SelectionData ← old↑];
    new.zone ← z;   -- save the zone that we allocated from

    IF op = move THEN {
      Selection.Clear[unmark: FALSE];  -- remove the selection manager
      DeleteBox[old];       -- delete the object from the manager's domain
      };
    };

  -- Delete the box from the manager's domain and redisplay the window.
  DeleteBox: PROC[box: Defs.SelectionPtr] = {
    window: Window.Handle ← box.window;
    mydata: Defs.Data ← Defs.GetContext[window];
    i: CARDINAL;
    Window.InvalidateBox[  -- this area will need repainting after deletion
      window: window,
      box: [box.place, box.dims]];
    FOR i IN [0..mydata.count) DO   -- first find the element and delete it
      IF mydata[i] = box THEN {
        mydata[i].zone.FREE[@mydata[i]];
        EXIT};
      ENDLOOP;
    FOR i IN [i..mydata.count -1) DO  -- compact data structure
      mydata.boxes[i] ← mydata.boxes[i + 1];
      ENDLOOP;
    mydata.count ← mydata.count - 1;
    Window.Validate[window];
    };
```

```
Move: PUBLIC PROC[window: Window.Handle] = {
  val: Selection.Value ← Selection.Convert[box];
  IF val.value = NIL THEN {
    mh: XMessage.Handle = Defs.GetMessageHandle[];
    msg: XString.ReaderBody ← XMessage.Get [mh, Defs.kunknownAction];
    Attention.Post[@msg];
    RETURN};
  Selection.Free[@val];
  Cursor.Set[move];
  [] ← TIPStar.SetMode[mode: move];
  };

-- When the user does a pointDown while moving a box we must add
-- the box to the manager's data, thus giving the manager control.
MoveBox: PUBLIC PROC[
  window: Window.Handle, data:Defs.Data, place: Window.Place] ={
  boxPtr: Defs.SelectionPtr;
  val: Selection.Value ← Selection.Convert[box];
  IF val.value = NIL THEN {
    mh: XMessage.Handle = Defs.GetMessageHandle[];
    msg: XString.ReaderBody ← XMessage.Get [mh, Defs.kunknownAction];
    Attention.Post[@msg];
    RETURN};
  Selection.Move[@val, NIL];
  boxPtr ← NARROW[val.value, Defs.SelectionPtr];
  AddToManager[window, data, boxPtr, place, @val]; -- add to manager's domain
  Selection.Free[@val];
  Cursor.Set[textPointer];
  [] ← TIPStar.SetMode[mode: normal];
  };

-- delete the box from the manager's domain
Delete: PUBLIC PROC[window: Window.Handle] = {
  Selection.ActOn[delete];
  };

}...
```

Day 3: DocInterchange and You

Outline

I. DocInterchange
   A. Creating a New Document
      1. Document Creation
      2. Appending
         a. Text
         b. Fields
         c. Paragraphs
         d. Page Breaks
         e. Page Format Characters
         f. Frames
         g. Modes
      3. Constants and Defaults
      4. FinishCreation

   B. Document Enumeration
      1. Opening
      2. Call-back Procs
      3. Closing

DocInterchange

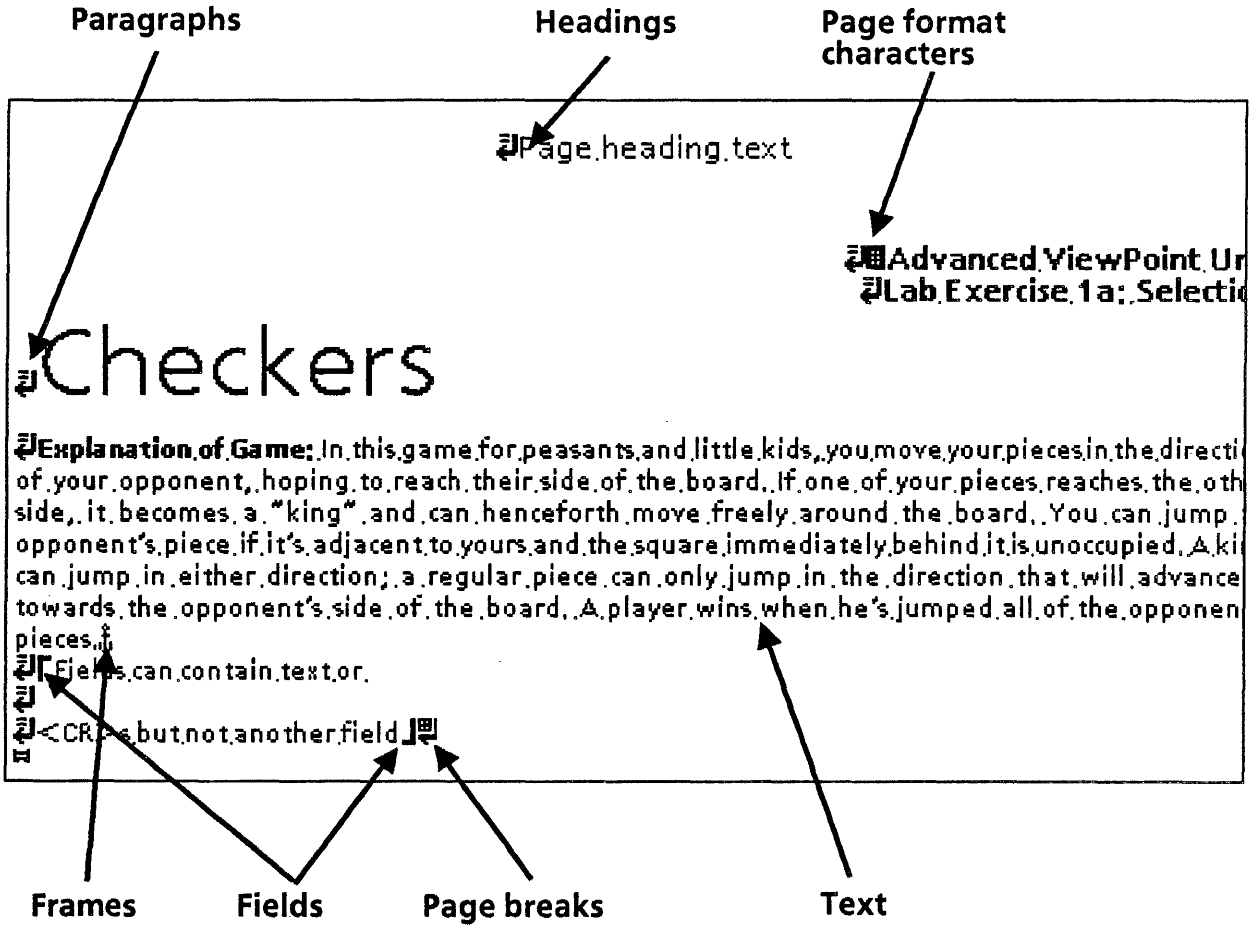
`DocInterchangeDefs` provides client programs with the ability to create or enumerate ViewPoint documents. Document structures such as text, fields, headings/ footings, and tiles (paragraph, page breaks, etc.) are supported. `DocInterchangeDefs` does not support creation or enumeration of frame content (e.g. tables, graphics); these objects are supported by their own interfaces.

We will also discuss the `DocInterchangePropsDefs` interface, which defines:

Page Properties
Field Properties
Font Properties
Paragraph Properties

DocInterchange

This slide shows some of the objects that an application can append to a document with DocInterchangeDefs. In addition, you can manipulate the objects' properties using additional interfaces.



Page.heading.text

Advanced.ViewPoint.Ur
Lab.Exercise.1a:.Selecti

Checkers

Explanation.of.Game:.In.this.game.for.peasants.and.little.kids,.you.move.your.pieces.in.the.directi of.your.opponent,.hoping.to.reach.their.side.of.the.board..If.one.of.your.pieces.reaches.the.oth side,.it.becomes.a."king".and.can.henceforth.move.freely.around.the.board..You.can.jump. opponent's.piece.if.it's.adjacent.to.yours.and.the.square.immediately.behind.it.is.unoccupied..A.ki can.jump.in.either.direction;.a.regular.piece.can.only.jump.in.the.direction.that.will.advance towards.the.opponent's.side.of.the.board..A.player.wins.when.he's.jumped.all.of.the.opponen pieces.

Fields.can.contain.text.or.

<CR>s.but.not.another.field

Document Creation

The three steps in document creation are: starting creation, appending objects to the document, and finishing creation. In order to start creating a document, you call `DocInterchangeDefs.StartCreation`. `NIL` values in the call to `StartCreation` cause the defaults to be used for the specified properties.

In examples, you will see the abbreviation **DI** and **DIP** used in place of DocInterchangeDefs and DocInterchangePropsDefs respectively.

```
DI.StartCreation: PROC[
  paginateOption: DI.PaginateOption ← compress,
  wantHeadingHandles, wantFootingHandles: BOOL ← FALSE,
  initialFontProps: DIP.ReadonlyFontProps ← NIL,
  initialParaProps: DIP.ReadonlyParaProps ← NIL,
  initialPageProps: DIP.ReadonlyPageProps ← NIL]
  RETURNS[
    doc: DI.Doc,
    leftHeading, rightHeading: DI.Heading,
    leftFooting, rightFooting: DI.Footing,
    status: DI.StartCreationStatus];
```

```
DI.Doc: TYPE = LONG POINTER TO DI.DocObject;
DI.DocObject: TYPE;
```

```
DI.PaginateOption: TYPE = {
  none, simple, compress, firstAvailable, lastAvailable(255)};
```

```
DI.StartCreationStatus: TYPE = MACHINE DEPENDENT {
  ok(0), notEnoughDiskSpace, notEnoughVM, firstAvailable, lastAvailable(255)};
```

Document Creation (cont'd)

While StartCreation looks complicated, all arguments to the procedure can be defaulted and the only essential return argument is doc. doc can be thought of as a handle to the unfinished document that is used when appending objects to that document. We will discuss the xxxProps arguments in detail later.

You can request that StartCreation return headings and footings for the new document; these can then be set using the append routines.

The paginateOption should be set to either simple or compress (if the document will be more than a few pages long); otherwise, the user will experience poor editing performance. simple pagination gives the outward signs of pagination but the document is not in optimized form. compress paginates the document and places it in optimized form.

Appending Objects

Once you have created a doc, you can begin appending objects to it (e.g. text characters). You append these objects to a DocInterchangeDefs.TextContainer, which is a variant record containing any of the different text-accepting areas of a document. Thus, if you opt for having headings or footings returned during StartCreation, you can append text to them using the same procedures as you would to append text to the body of a document.

```
DI.TextContainer: TYPE = RECORD[
  var: SELECT type: * FROM
    caption => [h: DI.Caption],
    doc => [h: DI.Doc],
    field => [h: DI.Field],
    heading => [h: DI.Heading],
    footing => [h: DI.Footing],
    spare1 => [h: DI.SpareTC],   -- spares are for future compatability
    spare2 => [h: DI.SpareTC],
    spare3 => [h: DI.SpareTC],
    spare4 => [h: DI.SpareTC],
  ENDCASE];

DI.SpareTC: TYPE = LONG UNSPECIFIED;
```

Appending Characters

To append a character, you call `DocInterchangeDefs.AppendChar`.

```
DI.AppendChar: PROC[
  to: DI.TextContainer,
  char: XChar.Character,
  fontProps: DIP.ReadonlyProps ← NIL,
  nToAppend: CARDINAL ← 1];
```

Appending XStrings

You can append XString.Readers to a text container by calling DocInterchangeDefs.AppendText. For efficiency, you should specify the appropriate context if it is known; otherwise, specify XString.unknownContext.

```
DI.AppendText: PROC[
  to: DI.TextContainer,
  text: XString.Reader,
  textEndContext: XString.Context,
  fontProps: DIP.ReadonlyProps ← NIL];
```

Appending Fields

Appending fields to a TextContainer is as simple as appending characters; however, the field itself may contain additional objects. Notice that the call to DocInterchangeDefs.AppendField will not only append a field to some TextContainer, but it will also return the field to the client. The client can then append items to the field (except another field).

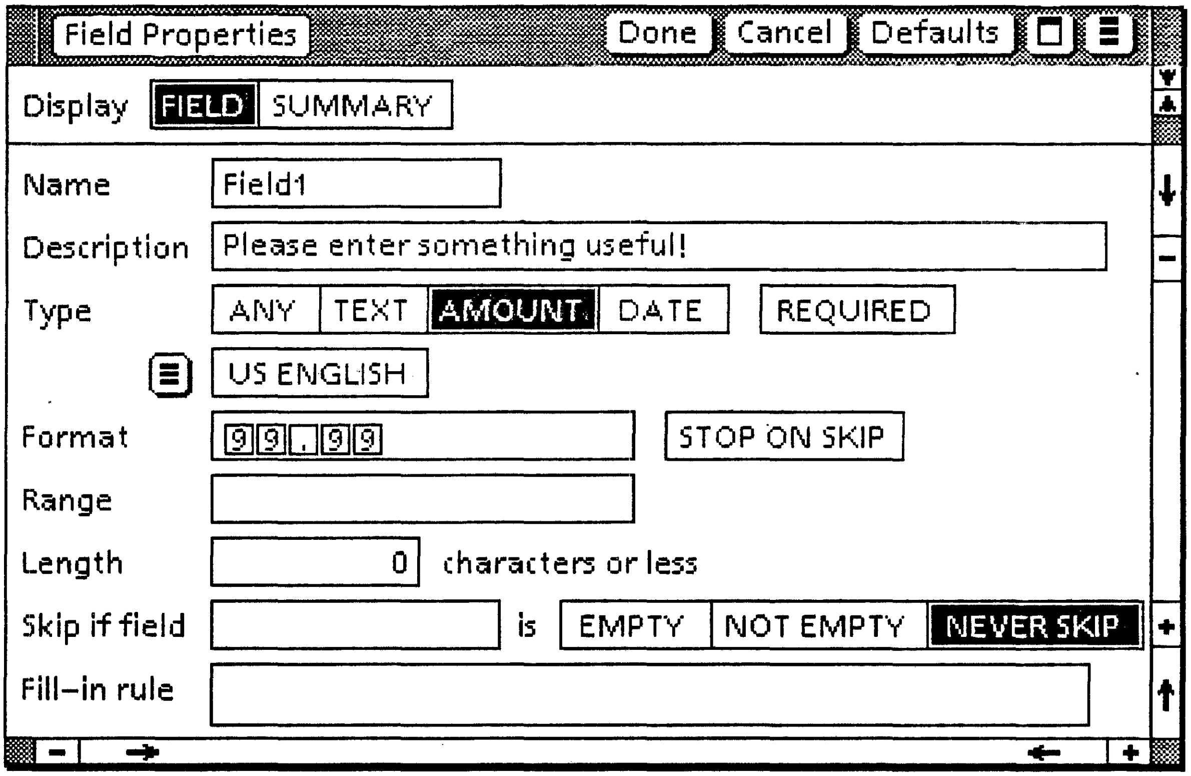
When you finished appending information, the field must be released.

```
DI.AppendField: PROC[
  to: DI.TextContainer,
  fieldProps: DIP.ReadonlyProps,
  fontProps: DIP.ReadonlyProps ← NIL]
  RETURNS[field: DI.Field];
```

```
DI.Field: TYPE = LONG POINTER TO DI.FieldObject;
DI.FieldObject: TYPE;
```

```
DI.ReleaseField: PROC[fieldPtr: LONG POINTER TO DI.Field];
```

ViewPoint Field Properties



Display [✓ FIELD] SUMMARY

Name Field1

Description Please enter something useful!

Type ANY TEXT [✓ AMOUNT] DATE REQUIRED

US ENGLISH

Format 99.99 STOP ON SKIP

Range

Length 0 characters or less

Skip if field is EMPTY NOT EMPTY [✓ NEVER SKIP]

Fill-in rule

Field Properties

```
DIP.FieldProps: TYPE = LONG POINTER TO FieldPropsRecord;
DIP.ReadonlyFieldProps: TYPE = LONG POINTER TO READONLY DIP.FieldPropsRecord;

DIP.FieldPropsRecord: TYPE = RECORD[
  language: MultiNational.Language,
  length: CARDINAL,
  required: BOOL,
  skipIf: DIP.SkipIfChoiceType,
  stopOnSkip: BOOL,
  type: DIP.FieldChoiceType,
  fillInRule,  -- changing fillInRule implies changing fillInRuleRuns
  description,
  format,
  name,
  range,
  skipIfField: XString.ReaderBody,
  fillInRuleRuns: DIP.FontRuns,
  spare1: LONG CARDINAL];

DIP.FieldChoiceType: TYPE = MACHINE DEPENDENT {
  any(0), text, amount, date, firstAvailable, lastAvailable(255)};

DIP.SkipIfChoiceType: TYPE = MACHINE DEPENDENT {
  empty(0), notEmpty, never, always, firstAvailable, lastAvailable(255)};
```

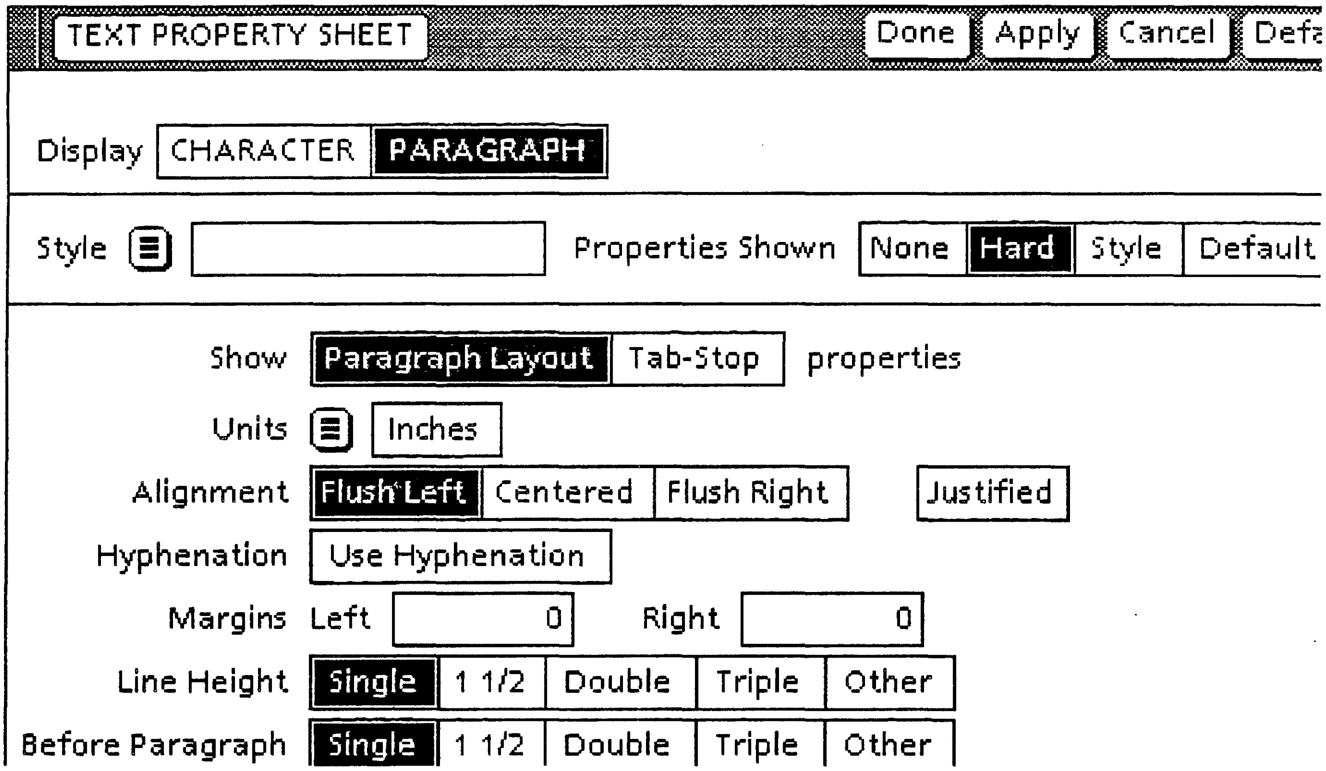
Appending Paragraphs

You can append a paragraph character (with DocInterchangeDefs.AppendNewParagraph) to a text container and specify initial properties for the paragraph. You can then change the paragraph properties later (i.e. in the middle of a paragraph) with DocInterchangeDefs.SetCurrentParagraphProps.

```
DI.AppendNewParagraph: PROC[
  to: DI.TextContainer,
  paraProps: DIP.ReadonlyProps ← NIL,
  fontProps: DIP.ReadonlyProps ← NIL,
  nToAppend: CARDINAL ← 1];

DI.SetCurrentParagraphProps: PROC[
  textContainer: DI.TextContainer,
  paraProps: DIP.ReadonlyParaProps];
```

Specifying Paragraph Properties

Paragraph properties are divided into two classes: basic properties and tab stops.



Display CHARACTER [✓ PARAGRAPH]

Style

Properties Shown None [✓ Hard] Style Default

Show [✓ Paragraph Layout] Tab-Stop properties

Units Inches

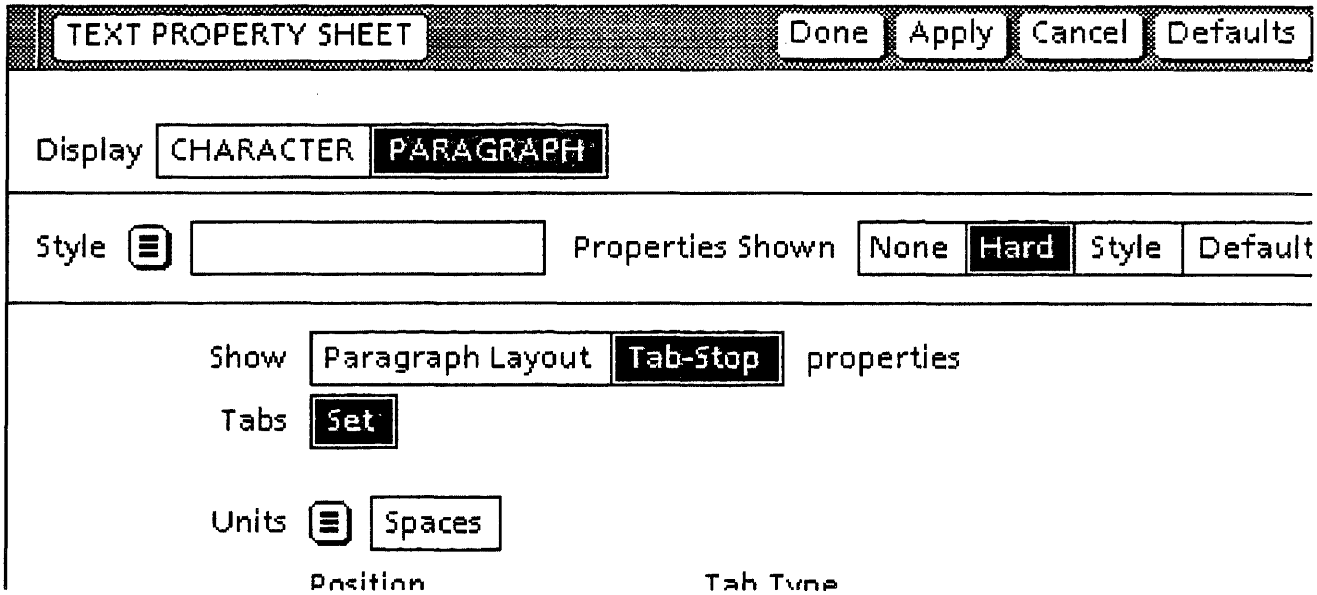
Alignment [✓ Flush Left] Centered Flush Right Justified

Hyphenation Use Hyphenation

Margins Left 0 Right 0

Line Height [✓ Single] 1 1/2 Double Triple Other

Before Paragraph [✓ Single] 1 1/2 Double Triple Other



Display CHARACTER [✓ PARAGRAPH]

Style

Properties Shown None [✓ Hard] Style Default

Show Paragraph Layout [✓ Tab-Stop] properties

Tabs [✓ Set]

Units Spaces

Position Tab Type

Specifying Paragraph Properties (cont'd)

You can specify paragraph properties via the `DocInterchagePropsDefs` interface; if you default the `paraProps` argument, the new paragraph character will inherit the properties of the previous paragraph.

```
DIP.ParaProps: TYPE = LONG POINTER TO DIP.ParaPropsRecord;
DIP.ReadonlyParaProps: TYPE = LONG POINTER TO READONLY DIP.ParaPropsRecord;

DIP.ParaPropsRecord: TYPE = RECORD[
  basicProps: DIP.BasicPropsRecord,
  tabStops: DIP.TabStops,
  spare1: LONG CARDINAL];  -- spare is for future expansion

DIP.BasicProps: TYPE = LONG POINTER TO DIP.BasicPropsRecord;
DIP.ReadonlyBasicProps: TYPE = LONG POINTER TO READONLY DIP.BasicPropsRecord;

DIP.BasicPropsRecord: TYPE = RECORD[
  preLeading, postLeading,
  leftIndent, rightIndent, lineHeight: CARDINAL,
  paraAlignment: DIP.ParaAlignment,
  justified, hyphenated, keepWithNextPara: BOOL,
  language: MultiNational.Language,
  streakSuccession: DIP.StreakSuccession,
  defaultTabStopSpacing: DIP.DefaultTabStopSpacing,
  defaultTabStopAlignment: DIP.TabStopAlignment,
  spare1: LONG CARDINAL];

DIP.ParaAlignment: TYPE = MACHINE DEPENDENT {
  left(0), center, right, firstAvailable, lastAvailable(255)};

DIP.StreakSuccession: TYPE = MACHINE DEPENDENT {
  leftToRight(0), rightToLeft, firstAvailable, lastAvailable(255)};
```

Specifying Tab Properties

Up to 100 tab stops may be associated with each paragraph object.

```
DIP.nTabsMax: CARDINAL = 100;

DIP.TabStops: TYPE = LONG DESCRIPTOR FOR ARRAY OF DIP.TabStop;

DIP.TabStop: TYPE = RECORD[
  dotLeader: BOOLEAN,
  tabStopOffset: DIP.TabStopOffset,
  tabStopAlignment: DIP.TabStopAlignment,
  spare1: LONG CARDINAL];

DIP.TabStopAlignment: TYPE = MACHINE DEPENDENT {
  left(0), center, right, decimal, firstAvailable, lastAvailable(255)};

DIP.TabStopOffset: TYPE = CARDINAL;

DIP.DefaultTabStopSpacing: TYPE = CARDINAL;
```

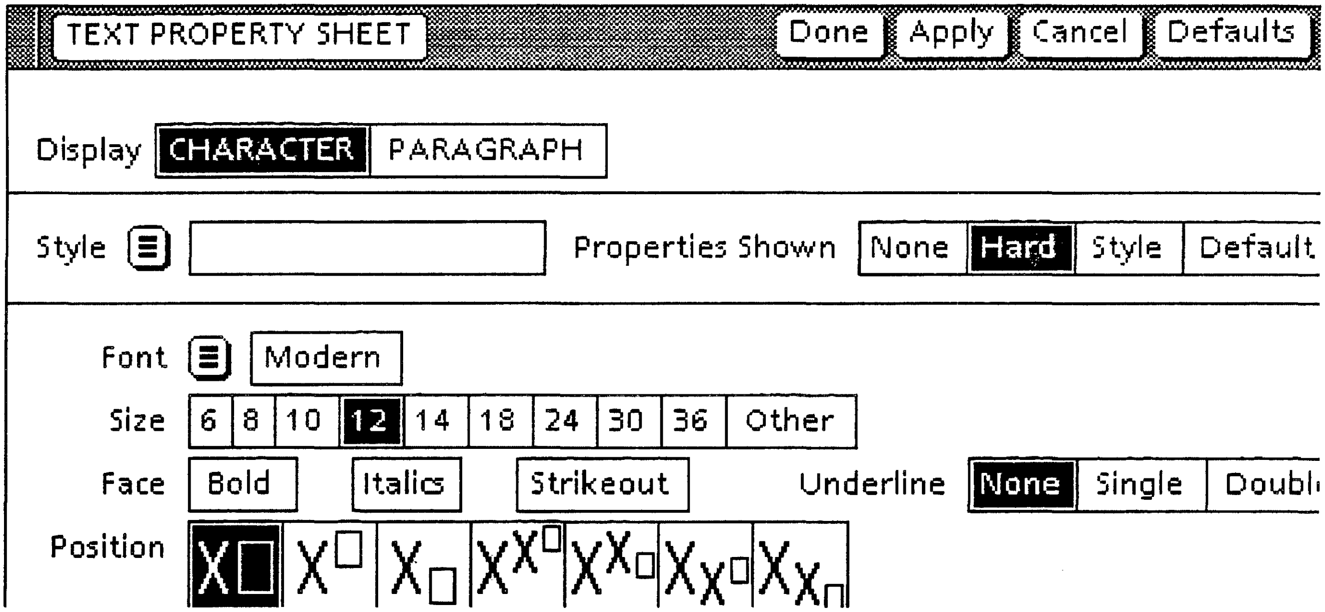
Appending Page Breaks

Page breaks can only be appended to a `DocInterchangeDefs.Doc`; they have font properties that can either be client specified or defaulted.

```
DI.AppendPageBreak: PROC[
  to: DI.Doc, fontProps: DIP.ReadonlyProps ← NIL];
```

Specifying Font Properties

Numerous fonts and sizes may be specified; however, the font that you choose must be on the local machine.



Display [✓ CHARACTER] PARAGRAPH

Style □ Properties Shown None [✓ Hard] Style Default

Font Modern

Size 6 8 10 [✓ 12] 14 18 24 30 36 Other

Face Bold Italics Strikeout Underline [✓ None] Single Doubl

Position

Specifying Font Properties (cont'd)

If you default fontProps, then the font properties are inherited from the previous objects appended. You should note that all features of each font may not be implemented (e.g. modern is not fixed pitch). Font properties should be set only if they are known fonts; otherwise, you might create black boxes instead of characters.

```
DIP.FontProps: TYPE = LONG POINTER TO DIP.FontPropsRecord;
DIP.ReadonlyFontProps: TYPE = LONG POINTER TO READONLY DIP.FontPropsRecord;

DIP.FontPropsRecord: TYPE = RECORD[
  fontDesc: DIP.FontDescription,
  offset: INTEGER,
  foregroundBackground: DIP.ForegroundBackground,
  nUnderlines: CARDINAL,
  strikeout: BOOL,
  placement: DIP.Placement,
  toBeDeleted, revised: BOOL,
  width: DIP.Width,
  spare1: LONG CARDINAL];

DIP.FontDescription: TYPE = RECORD[
  family: DIP.Family,
  designVariant: DIP.DesignVariant,
  posture: DIP.Posture,
  weight: DIP.Weight,
  pointSize: CARDINAL,
  serifness: DIP.Serifness,
  spare1: LONG CARDINAL];
```

Specifying Font Properties (cont'd)

```
DIP.DesignVariant: TYPE = MACHINE DEPENDENT {
  null(0), roman, italic, firstAvailable, lastAvailable(255)};

DIP.Family: TYPE = MACHINE DEPENDENT {
  century(0), frutiger(1), titan(2), pica(3), trojan(4), ... firstUnused(48),
  lastUnused(510), backstop(511)};

DIP.ForegroundBackground: TYPE = MACHINE DEPENDENT {
  null(0), blackOnWhite, whiteOnBlack, firstAvailable, lastAvailable(255)};

DIP.Placement: TYPE = MACHINE DEPENDENT{
  null(0), sub, subSub, subSuper, super, superSub, superSuper, userSpecified,
  firstAvailable, lastAvailable(255)};

DIP.Posture: TYPE = MACHINE DEPENDENT {
  null(0), upright, slanted, backslanted, firstAvailable, lastAvailable(255)};

DIP.Serifness: TYPE = MACHINE DEPENDENT {
  null(0), serif, sansSerif, firstAvailable, lastAvailable(255)};

DIP.Weight: TYPE = MACHINE DEPENDENT {
  null(0), ultraLight, extraLight, light, semiLight, medium, semiBold, bold,
  extraBold, ultraBold, firstAvailable, lastAvailable(255)};

DIP.Width: TYPE = MACHINE DEPENDENT {
  proportional(0), quarter, third, half, threeQuarters, full, firstAvailable,
  lastAvailable(255)};
```

Appending Page Format Characters

You can append a page format character to a document by calling DocInterchangeDefs.AppendPFC. This call will optionally return headings or footings that the client can fill in with calls to AppendText, AppendNewParagraph, or AppendChar. In addition, the client must specify page properties for the new PFC.

```
DI.Heading: TYPE = LONG POINTER TO DI.HeadingObject;
DI.HeadingObject: TYPE;

DI.Footing: TYPE = LONG POINTER TO DI.FootingObject;
DI.FootingObject: TYPE;

DIP.ReadonlyProps: TYPE = LONG POINTER TO READONLY DIP.PropsRecord;

DI.AppendPFC: PROC[
  to: DI.Doc,
  pageProps: DIP.ReadonlyPageProps,
  wantHeadingHandles, wantFootingHandles: BOOL ← FALSE,
  fontProps: DIP.ReadonlyFontProps ← NIL]
  RETURNS[
    leftHeading, rightHeading: DI.Heading,
    leftFooting, rightFooting: DI.Footing];
```

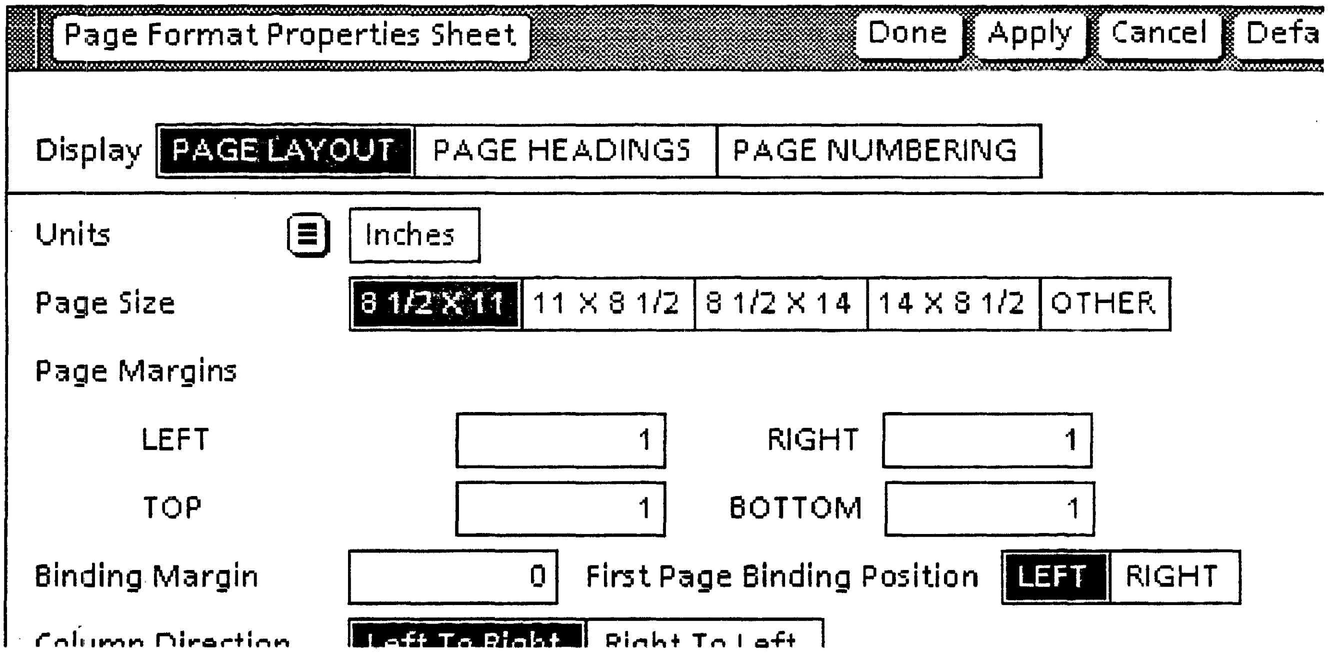
left = right ⇒ right = NIL

Appending Page Format Characters (cont'd)

If you specify a heading or footing, you must call the corresponding release procedures, DocInterchangeDefs.ReleaseHeading and DocInterchangeDefs.ReleaseFooting, to insure against space leaks. Even if you do not append any information to the headings or footings, they must still be freed.

```
DI.ReleaseHeading: PROC[headingPtr: LONG POINTER TO DI.Heading];

DI.ReleaseFooting: PROC[footingPtr: LONG POINTER TO DI.Footing];
```

Specifying Page Format Character Properties



Display [✓ PAGE LAYOUT] PAGE HEADINGS PAGE NUMBERING

Units Inches

Page Size [✓ 8 1/2 X 11] 11 X 8 1/2 8 1/2 X 14 14 X 8 1/2 OTHER

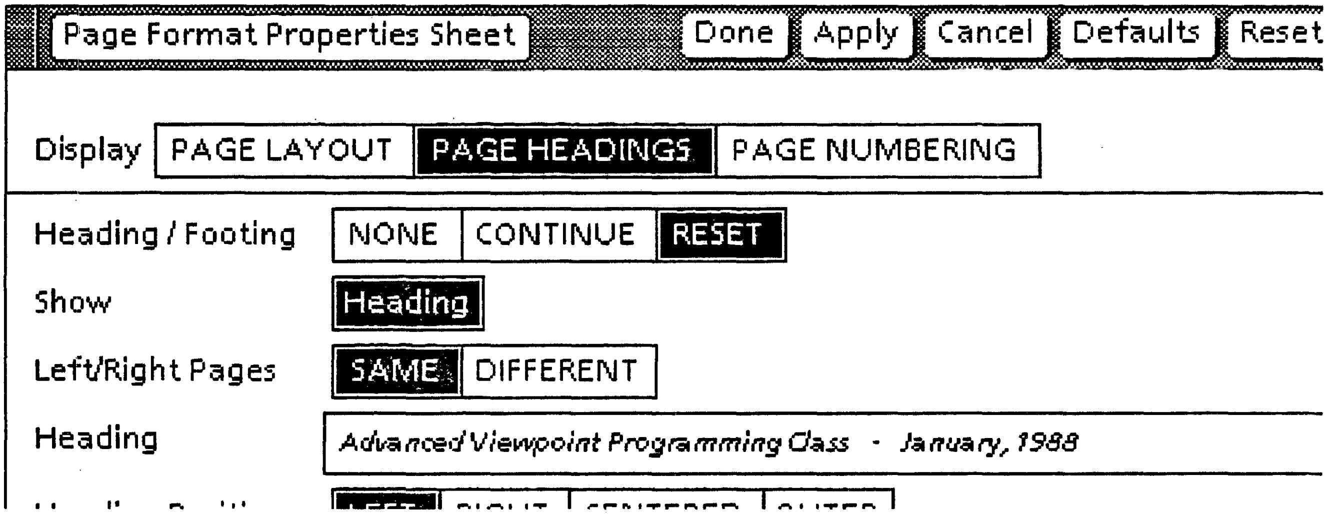
Page Margins

LEFT 1 RIGHT 1

TOP 1 BOTTOM 1

Binding Margin 0 First Page Binding Position [✓ LEFT] RIGHT

Column Direction [✓ Left To Right] Right To Left



Display PAGE LAYOUT [✓ PAGE HEADINGS] PAGE NUMBERING

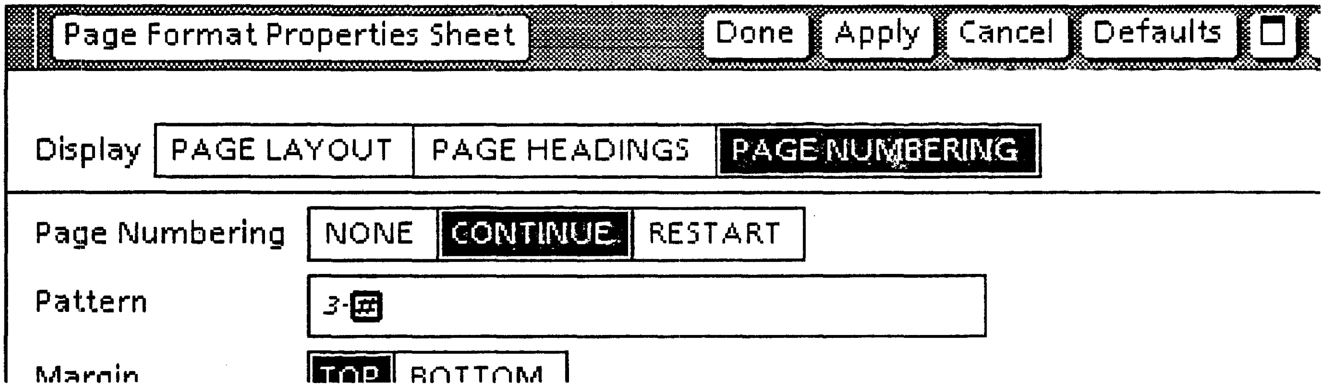
Heading / Footing NONE CONTINUE [✓ RESET]

Show [✓ Heading]

Left/Right Pages [✓ SAME] DIFFERENT

Heading *Advanced Viewpoint Programming Class - January, 1988*

Heading Position [✓ LEFT] RIGHT CENTERED OUTER



Display PAGE LAYOUT PAGE HEADINGS [✓ PAGE NUMBERING]

Page Numbering NONE [✓ CONTINUE] RESTART

Pattern *3-*#

Margin [✓ TOP] BOTTOM

Specifying Page Format Character Properties (cont'd)

There are a large number of PFC properties that you can specify, but not all features are implemented.

```
DIP.PageProps: TYPE = LONG POINTER TO DIP.PagePropsRecord;
DIP.ReadonlyPageProps: TYPE = LONG POINTER TO READONLY DIP.PagePropsRecord;

DIP.PagePropsRecord: TYPE = RECORD[
  pageDims: DIP.PageDims,  -- layout
  topMarginHeight, bottomMarginHeight,
  leftMarginWidth, rightMarginWidth: CARDINAL,
  startingPageSide: DIP.PageSide,
  bindingMarginWidth: CARDINAL,
  nColumns: CARDINAL, -- column structure
  balancedColumns, unequalColumnWidths: BOOL,
  columnSpacing: CARDINAL,
  columnWidths: DIP.ColumnWidths,
  startingPageNumber: CARDINAL, -- page numbering
  pageNumberFormat: DIP.NumberFormat,
  restartPageNumbering: BOOL,
  startingLineNumber, -- line numbering
  lineNumberInterval: CARDINAL,
  lineNumberFormat: DIP.NumberFormat,
  lineNumberLocation: DIP.LineNumberLocation,
  headingStartsOnThisPage, headingSameOnLeftRightPages,
  footingStartsOnThisPage,footingSameOnLeftRightPages: BOOL,
  spare1: LONG CARDINAL];
```

Specifying Page Format Character Properties (cont'd)

```
DIP.PageDims: TYPE = MACHINE DEPENDENT RECORD[w, h: CARDINAL];

DIP.PageSide: TYPE = MACHINE DEPENDENT {
  nil(0), left, right, firstAvailable, lastAvailable(255)};

DIP.NumberFormat: TYPE = MACHINE DEPENDENT {
  cardinal(0), lowerCaseLetter, upperCaseLetter, lowerCaseRoman,
  upperCaseRoman, firstAvailable, lastAvailable(255)};

DIP.LineNumberLocation: TYPE = MACHINE DEPENDENT {
  leftMargin(0), rightMargin, outerMargin, bothMargins, firstAvailable,
  lastAvailable(255)};

DIP.ColumnWidths: TYPE = LONG POINTER TO DIP.ColumnWidthsRecord;

DIP.ColumnWidthsRecord: TYPE = RECORD[
  length: CARDINAL,
  spare1: LONG CARDINAL,
  widths:SEQUENCE maxLength:CARDINAL OF DIP.ColumnWidthRecord];

DIP.ColumnWidthRecord: TYPE = RECORD[
  w: CARDINAL,
  spare1: LONG CARDINAL];
```

Appending Frames

Some of the types of anchored frames that you will typically append to a document are: bitmaps, graphics, and tables. DocInterchangeDefs allows to you append these frames to a document but does not allow you to specify their content. Each of these objects has a separate interface for manipulating its content; graphics and table frames will be discussed in detail later.

```
DI.AppendAnchoredFrame: PROC[
  to: DI.Doc,
  type: DI.AnchoredFrameType,
  anchoredFrameProps: DI.ReadonlyFrameProps,
  content: DI.Instance ← instanceNil,
  wantTopCaptionHandle, wantBottomCaptionHandle,
  wantLeftCaptionHandle, wantRightCaptionHandle: BOOL ← FALSE,
  anchorFontProps: DIP.ReadonlyFontProps ← NIL]
  RETURNS[
    anchoredFrame: DI.Instance, topCaption, bottomCaption,
    leftCaption, rightCaption: DI.Caption];
```

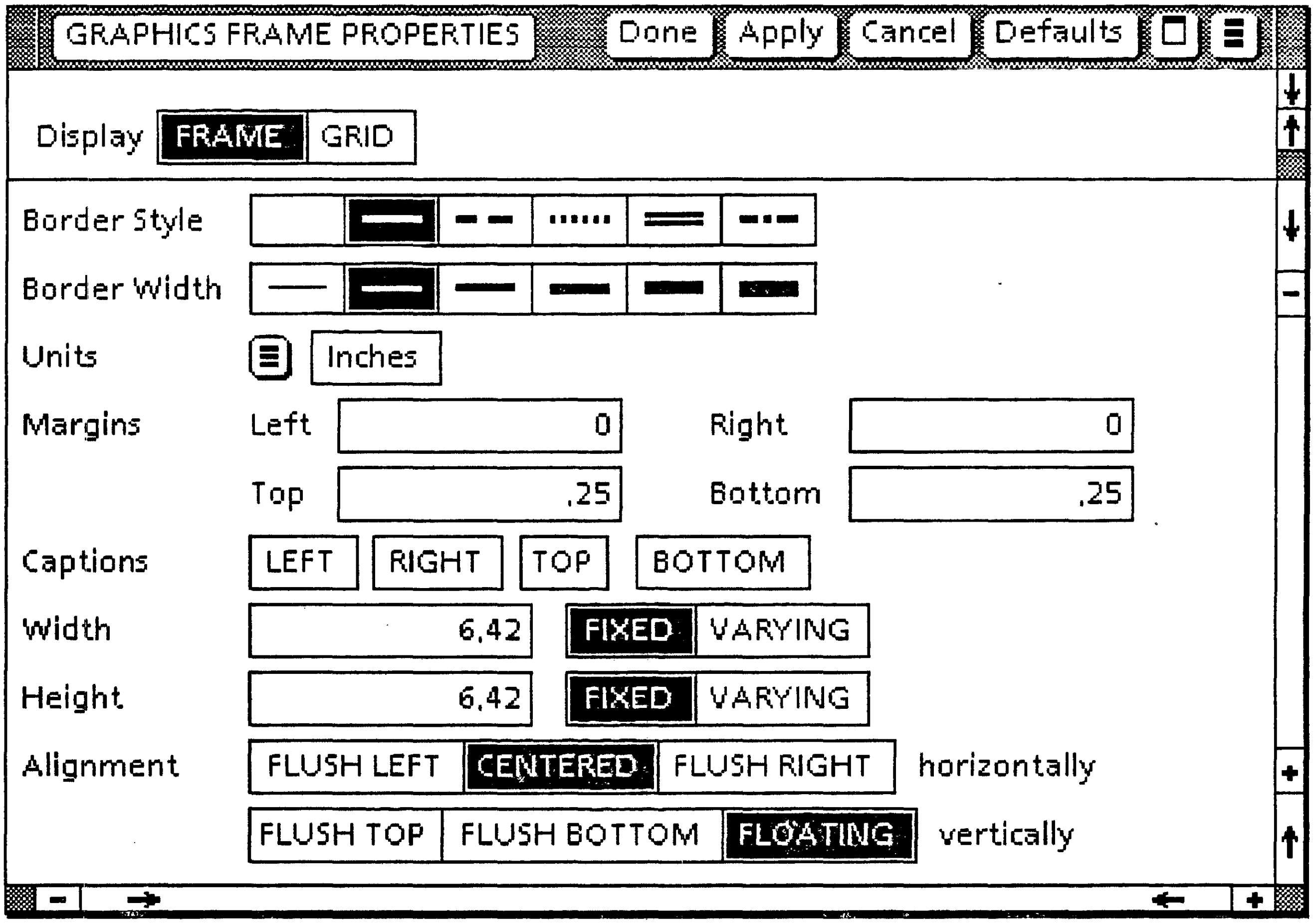
```
DI.AnchoredFrameType: TYPE = MACHINE DEPENDENT {
  nil(0), bitmap, cuspButton, equation, graphics, IMG, table, text, illustrator,
  firstAvailable, lastAvailable(255)};
```

```
DI.Caption: TYPE = LONG POINTER TO DI.CaptionObject;
```

```
DI.ReleaseCaption: PROC[captionPtr: LONG POINTER TO DI.Caption];
```

Frame Properties

All frames have a set of properties that are described by the DocInterchangePropsDefs interface. Objects that have frames include: graphics, tables, and charts.



Display [✓ FRAME] GRID

Border Style

Border Width

Units Inches

Margins Left 0 Right 0

Top ,25 Bottom ,25

Captions LEFT RIGHT TOP BOTTOM

Width 6,42 [✓ FIXED] VARYING

Height 6,42 [✓ FIXED] VARYING

Alignment FLUSH LEFT [✓ CENTERED] FLUSH RIGHT horizontally

FLUSH TOP FLUSH BOTTOM [✓ FLOATING] vertically

Frame Properties

```
DIP.FrameProps: TYPE = LONG POINTER TO DIP.FramePropsRecord;
DIP.ReadonlyFrameProps: TYPE = LONG POINTER TO READONLY DIP.FramePropsRecord;

DIP.FramePropsRecord: TYPE = RECORD[
  borderStyle: DIP.BorderStyle,
  borderThickness: CARDINAL,
  frameDims: DIP.FrameDims,  --TYPE = RECORD [w, h: CARDINAL]
  fixedWidth, fixedHeight: BOOL,
  span: DIP.Span,
  verticalAlignment: DIP.VerticalAlignment,
  horizontalAlignment: DIP.HorizontalAlignment,
  topMarginHeight, bottomMarginHeight,
  leftMarginWidth, rightMarginWidth: CARDINAL,
  spare1: LONG CARDINAL];

DIP.BorderStyle: TYPE = MACHINE DEPENDENT {invisible(0),
  solid, dashed, broken, dotted, double, firstAvailable, lastAvailable(255)};

DIP.HorizontalAlignment: TYPE = MACHINE DEPENDENT {
  left(0), centered, right, floating, firstAvailable, lastAvailable(255)};

DIP.Span: TYPE = MACHINE DEPENDENT {partialColumn(0),
  fullColumn, partialPage, fullPage, firstAvailable, lastAvailable(255)};

DIP.VerticalAlignment: TYPE = MACHINE DEPENDENT {
  top(0), bottom, floating, firstAvailable, lastAvailable(255)};
```

Note: All dimensions are in 1/72 inch.

Modes

The mode of a document determines the way the information is displayed to the user (e.g. structure showing, cover sheet, etc.). You can retrieve and modify the document's mode as an assistance to the user.

```
DI.GetModeProps: PROC[doc: DI.Doc, modeProps: DIP.ModeProps];

DI.SetModeProps: PROC[
  doc: DI.Doc,
  modeProps: DIP.ReadonlyModeProps,
  selections: DIP.ModeSelections];

DIP.ModeProps: TYPE = LONG POINTER TO DIP.ModePropsRecord;
DIP.ReadonlyModeProps: TYPE = LONG POINTER TO READONLY DIP.ModePropsRecord;

DIP.ModePropsRecord: TYPE = RECORD[
  structureShowing, nonPrintingShowing,
  coverSheetShowing, promptFields: BOOL,
  spare1: LONG CARDINAL];

DIP.ModeSelections: TYPE =
  PACKED ARRAY DIP.ModeElements OF DIP.BooleanFalseDefault;

DIP.ModeElements: TYPE = {
  structureShowing, nonPrintingShowing, coverSheetShowing, promptFields,
  spare1, spare2, spare3, spare4, spare5, spare6, spare7, spare8};

DIP.BooleanFalseDefault: TYPE = BOOL ← FALSE;
```

Constants and Defaults

DocInterchangePropsDefs provides a set of constants that you can use to initialize property records with "neutral" values. In most cases, each value is the same as what would be set by the corresponding `Get*PropsDefaults` operation.

```
DIP.nullFrameProps: DIP.FramePropsRecord = [...];
DIP.nullPageProps: DIP.PagePropsRecord = [...];
DIP.nullColumnWidth: DIP.ColumnWidthRecord = [...];
DIP.nullFieldProps: DIP.FieldPropsRecord = [...];
DIP.nullFontProps: DIP.FontPropsRecord = [...];
DIP.nullFontDescription: DIP.FontDescription = [...];
DIP.nullRun: DIP.Run = [...];
DIP.nullParaProps: DIP.ParaPropsRecord = [...];
DIP.nullBasicProps: DIP.BasicPropsRecord = [...];
DIP.nullTabStop: DIP.TabStop = [...];
DIP.nullModeProps: DIP.ModePropsRecord = [...];

DIP.classic: Family = century;
DIP.modern: Family = frutiger;

DIP.GetFramePropsDefaults: PROC[props: DIP.FrameProps];
DIP.GetPagePropsDefaults: PROC[props: DIP.PageProps];
DIP.GetFieldPropsDefaults: PROC[props: DIP.FieldProps];
DIP.GetFontPropsDefaults: PROC[props: DIP.FontProps];
DIP.GetParaPropsDefaults: PROC[props: DIP.ParaProps];
DIP.GetModePropsDefaults: PROC[props: DIP.ModeProps];
```

Additional Append Comments

- The append procedures often allow font, paragraph, frame, or page properties to be set; however, if you default these arguments, then the properties of the preceding object will apply to the new object being appended.

- The client must manage the storage required for arguments passed into the append procedures (this does not apply to handles that DocInterchange returns to the client). When an append procedure returns, the client may release the storage allocated for any arguments. It is important that the client release non-NIL handles obtained from append procedures even if they were not used.

- The ERROR DocInterchangeDefs.Error can be raised by any operation. If this error is raised by an Append* procedure, the object will not be appended.

```
DI.Error: ERROR[why: DI.ErrorCode];

DI.ErrorCode: TYPE = MACHINE DEPENDENT {
  containerFull(0), documentFull, readonlyDoc, outOfDiskSpace,
  outOfVM, objectIllegalInContainer, badParameter,
  unimplemented, firstAvailable, lastAvailable(255)}
```

Finishing Creation

After you have called StartCreation and maybe appended some information to the document, you should either finish the creation process or abort the uncompleted document. Clients can also invoke a call-back procedure before finishing the document creation process; this call-back has the power to abort the document's creation.

```
DI.FinishCreation: PROC[
  docPtr: LONG POINTER TO DI.Doc,
  checkAbortProc: DI.CheckAbortProc ← NIL,
  checkAbortClientData: LONG POINTER ← NIL]
  RETURNS[
    docFile: NSFile.Handle,
    session: NSFile.Session,
    status: DI.FinishCreationStatus];

DI.AbortCreation: PROC[docPtr: LONG POINTER TO DI.Doc];

DI.FinishCreationStatus: TYPE = MACHINE DEPENDENT {
  ok(0), aborted, okButNotEnoughDiskSpaceToPaginate, okBuNotEnoughVMToPaginate,
  okButUnknownPaginateProblem, firstAvailable, lastAvailable(255)};

DI.CheckAbortProc: TYPE = PROC[clientData: LONG POINTER]
  RETURNS[abort: BOOL];
```

Finishing Creation (cont'd)

The NSFile handle, `docFile`, that is returned by `FinishCreation` is a temporary NSFile and will be purged at the next system boot unless it is made permanent. To make the file permanent, the file should be moved to the desktop and the desktop should be given a reference to the file.

```
NSFile.Move: PROC[
  file: NSFile.Handle, destination: NSFile.Handle,
  attributes: NSFile.AttributesList ← NSFile.nullAttributeList,
  session: NSFile.Session ← NSFile.nullSession];
```

```
StarDesktop.AddReferenceToDesktop: PROC[
  reference: NSFile.Reference,
  place: Window.Place ← StarDesktop.nextPlace];
```

You will see an example of this in upcoming slides.

Document Enumeration

DocInterchangeDefs not only provides procedures for creating documents; it also allows you to enumerate the objects within an existing document. Before you can enumerate, you must call DocInterchangeDefs.Open to get a handle to the document and prepare the file for enumeration. Open allows you to specify a session argument for background processing. The password argument is not currently supported.

```
DI.Open: PROC[
  docFileRef: NSFile.Reference,
  session: NSFile.Session ← NSFile.nullSession,
  password: XString.Reader ← NIL]
  RETURNS[doc: DI.Doc, status: DI.OpenStatus];
```

```
DI.OpenStatus: TYPE = MACHINE DEPENDENT {
  ok(0), malFormed, incompatible, notLocal, outOfDiskSpace, outOfVM,
  busy,invalidPassword, firstAvailable, lastAvailable(255)};
```

Document Enumeration (cont'd)

Once you have opened the document, you can call DocInterchangeDefs.Enumerate; this requires a set of procedures that it can call for each object in the document. If a procedure is left defaulted to NIL, then objects of that type are ignored. The clientData parameter in the call to Enumerate is used to pass data between the client defined enumeration procedures.

```
DI.Enumerate: PROC[
  textContainer: DI.TextContainer,
  procs: DI.EnumProcs,
  clientData: LONG POINTER ← NIL]
  RETURNS[dataSkipped: BOOL];
```

```
DI.EnumProcs: TYPE = LONG POINTER TO DI.EnumProcsRecord;
```

```
DI.EnumProcsRecord: TYPE = RECORD[
  anchoredFrameProc: DI.AnchoredFrameProc ← NIL,
  columnBreakProc: DI.ColumnBreakProc ← NIL,
  fieldProc: DI.FieldProc ← NIL,
  newParagraphProc: DI.NewParagraphProc ← NIL,
  pageBreakProc: DI.PageBreakProc ← NIL,
  pfcProc: DI.PFCProc ← NIL,
  textProc: DI.TextProc ← NIL,
  tileProc: DI.TileProc ← NIL,
  spare1Proc: DI.SpareProc ← NIL, spare2Proc: DI.SpareProc ← NIL,
  spare3Proc: DI.SpareProc ← NIL, spare4Proc: DI.SpareProc ← NIL];
```

Document Enumeration (cont'd)

When the client supplied call-back procedures are invoked, they have all the properties for the particular object passed in. These procedures can then use this information for pattern matching, building another document, etc. The client cannot modify the document or any of the arguments passed in to the enumerate procedures; the enumerated information is read-only and valid only for the duration of the call.

When any of the call-back procedures returns `stop = TRUE`, the enumeration stops.

```
DI.NewParagraphProc: TYPE = PROC[
  clientData: LONG POINTER,
  fontProps: DIP.ReadonlyFontProps,
  paraProps: DIP.ReadonlyParaProps]
  RETURNS[stop: BOOL ← FALSE];

DI.PageBreakProc: TYPE = PROC[
  clientData: LONG POINTER,
  fontProps: DIP.ReadonlyFontProps]
  RETURNS[stop: BOOL ← FALSE];

DI.TextProc: TYPE = PROC[
  clientData: LONG POINTER,
  fontProps: DIP.ReadonlyFontProps,
  text: XString.Reader, textEndContext: XString.Context]
  RETURNS[stop: BOOL ← FALSE];
```

Document Enumeration (cont'd)

Some of the enumerate procedures return text containing objects (e.g. fields and PFCs), which can also be enumerated. NIL values are returned for objects that were never allocated (such as headings or footings). Since the client does not own the storage for these text containers, it is not necessary to release them when finished.

```
DI.FieldProc: TYPE = PROC[
  clientData: LONG POINTER,
  fontProps: DIP.ReadonlyFontProps,
  fieldProps: DIP.ReadonlyFieldProps,
  field: DI.Field]
  RETURNS[stop: BOOL ← FALSE];

DI.PFCProc: TYPE = PROC[
  clientData: LONG POINTER,
  fontProps: DIP.ReadonlyFontProps,
  pageProps: DIP.ReadonlyPageProps,
  leftHeading, rightHeading: DI.Heading,
  leftFooting, rightFooting: DI.Footing]
  RETURNS[stop: BOOL ← FALSE];

DI.AnchoredFrameProc: TYPE = PROC[
  clientData: LONG POINTER,
  type: DI.AnchoredFrameType,
  anchorFontProps: DIP.ReadonlyFontProps,
  anchoredFrame: DI.Instance,
  anchoredFrameProps: DIP.ReadonlyFrameProps,
  content: DI.Instance,
  topCaption, bottomCaption, leftCaption,
  rightCaption: DI.Caption]
  RETURNS[stop: BOOL ← FALSE];
```

Document Enumeration (cont'd)

When `Enumerate` returns, the client will generally close the document to release any storage associated with the document handle and to allow other clients to access the file. `Close` sets `docPtr↑` to `NIL`.

```
DI.Close: PROC[docPtr: LONG POINTER TO DI.Doc];
```

Document Example

This sample program enumerates a selected document and copies its fields into a newly created document. When the enumeration is complete, the new document is saved and placed on the desktop and the original is closed. Since fields and PFC characters may contain additional information, the program must recursively copy their content as well.

```
-- Copyright (C) Xerox Corporation 1986. All rights reserved.

DIRECTORY
  Attention,
  BWSZone,
  Heap,
  DocInterchangeDefs,
  MenuData,
  NSFile,
  Selection,
  StarDesktop,
  Window,
  XString;

CopyDoc: PROGRAM
  IMPORTS Attention, BWSZone, DocInterchangeDefs, Heap, MenuData, NSFile,
Selection, StarDesktop, XString = {

  FileProblem: SIGNAL = CODE;

  -- used for passing a text container (via clientData) to call-back procs
  ContainerPtr: TYPE = LONG POINTER TO DocInterchangeDefs.TextContainer;

  -- call-back procedures called when enumerating a document
  enumProcs: DocInterchangeDefs.EnumProcsRecord ← [
    fieldProc: FieldProc,
    newParagraphProc: NewParagraphProc,
    pageBreakProc: PageBreakProc,
    pfcProc: PFCProc,
    textProc: TextProc];

  -- simply add the copy document command to the attention menu
  Init: PROC = {
    copyDoc: XString.ReaderBody ← XString.FromSTRING["Copy Document"L];
    Attention.AddMenuItem[
      MenuData.CreateItem[
         zone: BWSZone.permanent,
         name: @copyDoc,
         proc: Copy] ];
    };
```

```
-- enumerate through the selected document and copy the contents to a new document
Copy: MenuData.MenuProc = {
  ref: NSFile.Reference ← OpenSelectedFile[!FileProblem => GOTO Exit];
  newFile: NSFile.Handle;
  oldDoc, newDoc: DocInterchangeDefs.Doc;
  status: DocInterchangeDefs.OpenStatus;
  finishStatus: DocInterchangeDefs.FinishCreationStatus;
  textContainer: DocInterchangeDefs.TextContainer;

  [oldDoc, status] ← DocInterchangeDefs.Open[docFileRef:ref];
  IF status # ok THEN GOTO Exit;

  -- create new document and begin copying the old one
  newDoc ← DocInterchangeDefs.StartCreation[].doc;
  textContainer ← [doc[h: newDoc]];
  [] ← DocInterchangeDefs.Enumerate[
    textContainer: [doc[h: oldDoc]],
    procs: @enumProcs,
    clientData: @textContainer];

  DocInterchangeDefs.Close[docPtr: @oldDoc];
  [newFile, ,finishStatus] ← DocInterchangeDefs.FinishCreation[
    docPtr: @newDoc, checkAbortProc: DummyAbortProc];

  IF finishStatus # aborted THEN {
    desktop: NSFile.Handle ← NSFile.OpenByReference[
      StarDesktop.GetCurrentDesktopFile[]];
    NSFile.Move[file: newFile, destination: desktop];
    StarDesktop.AddReferenceToDesktop[reference: NSFile.GetReference[newFile]];
    NSFile.Close[newFile]}
  ELSE DocInterchangeDefs.AbortCreation[docPtr: @newDoc];
  EXITS Exit => NULL;
  };                        close [desktop]

DummyAbortProc: DocInterchangeDefs.CheckAbortProc = {RETURN[FALSE]};

-- return a reference to the selected file.
-- If no file is selected print a message and raise the error FileProblem
OpenSelectedFile: PROC RETURNS[ref: NSFile.Reference] = {
  element: Selection.Value ← Selection.Convert[target: file];
  refPtr: LONG POINTER TO NSFile.Reference ← element.value;
  IF refPtr = NIL THEN SIGNAL FileProblem;
  RETURN[refPtr↑];
  };
```

```
-- copy the field passed in to the textcontainer specified by clientdata.
-- Since fields may contain information, recursively enumerate the contents of the field
FieldProc: DocInterchangeDefs.FieldProc = {
  textContainer: DocInterchangeDefs.TextContainer ←
    LOOPHOLE[clientData, ContainerPtr]↑;
  newField: DocInterchangeDefs.Field ← DocInterchangeDefs.AppendField[
    to: textContainer,
    fieldProps: fieldProps,
    fontProps: fontProps !DocInterchangeDefs.Error => GOTO bad];
  -- now enumerate any items within the field
  newTextContainer: DocInterchangeDefs.TextContainer ← [field[h: newField]];
  [] ← DocInterchangeDefs.Enumerate[
    textContainer: [field[h: field]],
    procs: @enumProcs,
    clientData: @newTextContainer];
  DocInterchangeDefs.ReleaseField[@newField];
  EXITS bad => NULL;
  };

NewParagraphProc: DocInterchangeDefs.NewParagraphProc = {
  textContainer: DocInterchangeDefs.TextContainer ←
    LOOPHOLE[clientData, ContainerPtr]↑;
  [] ← DocInterchangeDefs.AppendNewParagraph[
    to: textContainer,
    fontProps: fontProps,
    paraProps: paraProps !DocInterchangeDefs.Error => CONTINUE];
  };

-- will need to use a with statement to determine if doc or field
PageBreakProc: DocInterchangeDefs.PageBreakProc = {
  textContainer: DocInterchangeDefs.TextContainer ←
    LOOPHOLE[clientData, ContainerPtr]↑;
  WITH z: textContainer SELECT FROM
    doc => [] ← DocInterchangeDefs.AppendPageBreak[
      to: z.h,
      fontProps: fontProps !DocInterchangeDefs.Error => CONTINUE];
    ENDCASE;
  };

-- append the text passed in to the new text container
TextProc: DocInterchangeDefs.TextProc = {
  textContainer: DocInterchangeDefs.TextContainer ←
    LOOPHOLE[clientData, ContainerPtr]↑;
  DocInterchangeDefs.AppendText[
    to: textContainer,
    text: text,
    textEndContext: textEndContext,
    fontProps: fontProps !DocInterchangeDefs.Error => CONTINUE];
  };
```

```
PFCProc: DocInterchangeDefs.PFCProc = {
 textContainer: DocInterchangeDefs.TextContainer ←
  LOOPHOLE[clientData, ContainerPtr]↑;
 newLeftHeading, newRightHeading: DocInterchangeDefs.Heading;
 newLeftFooting, newRightFooting: DocInterchangeDefs.Footing;
 WITH z: textContainer SELECT FROM
  doc => {
   [newLeftHeading, newRightHeading, newLeftFooting, newRightFooting] ←
    DocInterchangeDefs.AppendPFC[
     to: z.h, pageProps: pageProps,
     wantHeadingHandles: (leftHeading # NIL) OR (rightHeading # NIL),
     wantFootingHandles: (leftFooting # NIL) OR (rightFooting # NIL),
     fontProps: fontProps ! DocInterchangeDefs.Error => GOTO bad];
   IF leftHeading # NIL THEN {
    newTextContainer: DocInterchangeDefs.TextContainer ←
     [heading[h: newLeftHeading]];
    [] ← DocInterchangeDefs.Enumerate[
     textContainer: [heading[h:leftHeading]],
     procs: @enumProcs,
     clientData: @newTextContainer];
    DocInterchangeDefs.ReleaseHeading[@newLeftHeading]};
   IF rightHeading # NIL THEN {
    newTextContainer: DocInterchangeDefs.TextContainer ←
     [heading[h: newRightHeading]];
    [] ← DocInterchangeDefs.Enumerate[
     textContainer: [heading[h:rightHeading]],
     procs: @enumProcs,
     clientData: @newTextContainer];
    DocInterchangeDefs.ReleaseHeading[@newRightHeading]};
   IF leftFooting # NIL THEN {
    newTextContainer: DocInterchangeDefs.TextContainer ←
     [footing[h: newLeftFooting]];
    [] ← DocInterchangeDefs.Enumerate[
     textContainer: [footing[h:leftFooting]],
     procs: @enumProcs,
     clientData: @newTextContainer];
    DocInterchangeDefs.ReleaseFooting[@newLeftFooting]};
   IF rightFooting # NIL THEN {
    newTextContainer: DocInterchangeDefs.TextContainer ←
     [footing[h: newRightFooting]];
    [] ← DocInterchangeDefs.Enumerate[
     textContainer: [footing[h:rightFooting]],
     procs: @enumProcs,
     clientData: @newTextContainer];
    DocInterchangeDefs.ReleaseFooting[@newRightFooting]}  };
  ENDCASE
 EXITS bad => NULL
 };

-- mainline code
Init[];
}...
```

ViewPoint Tables

Outline

I. ViewPoint Tables
   A. Table Overview
   B. Fill-in Rules
   C. TableInterchangeDefs
      1. Creating new table
      2. Modifying existing tables
      3. Column and row properties
   D. Miscellaneous

ViewPoint Tables

Tables simplify and organize information for the reader. In addition to presenting statistical and tabular information, tables can also be used for multiple column data, in which rows of information must stay together across columns. The number of rows and columns, column width and margins, and the widths and styles of ruling lines can all be specified.

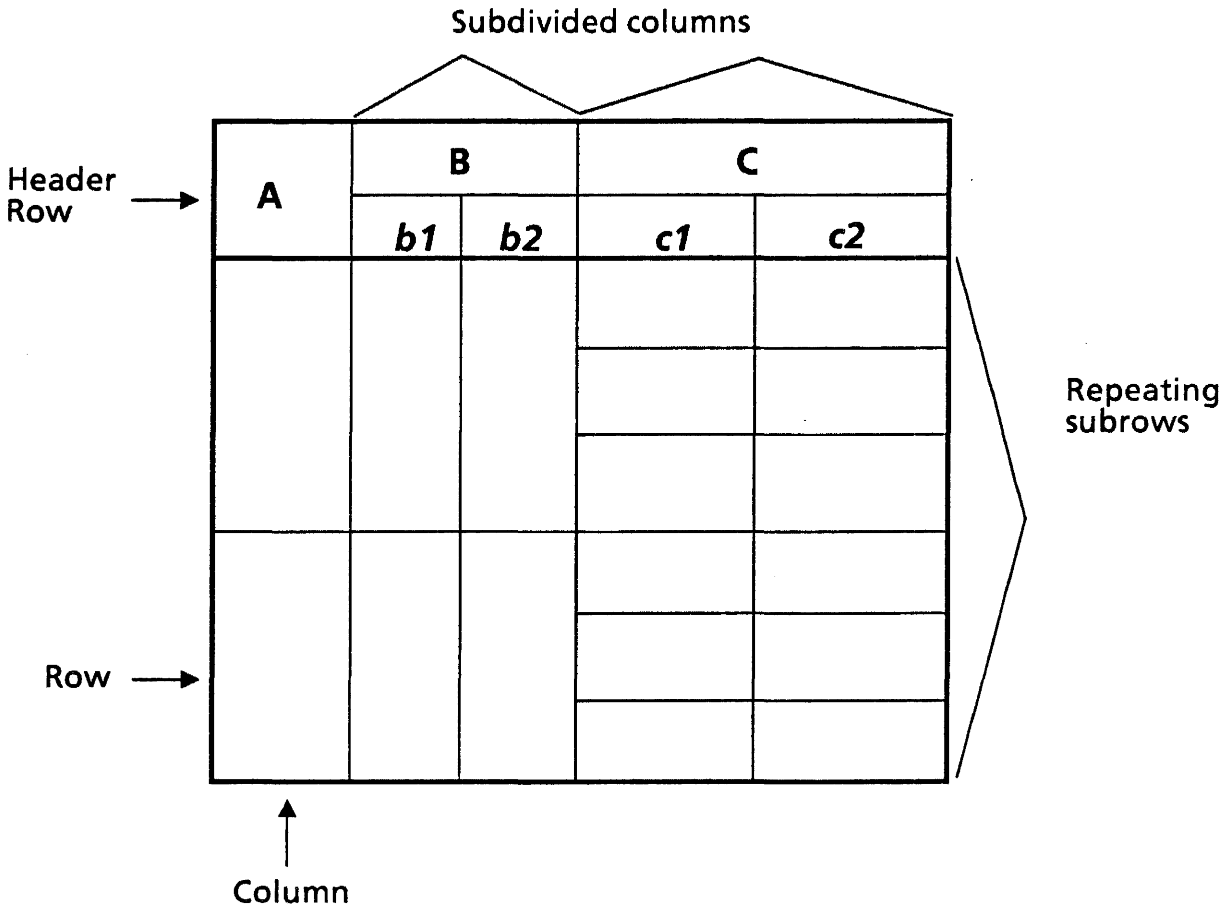
Name	College Graduate?
Frank	Yes
Mark	Yes

Example of a table

ViewPoint Table Structure

A table consists of individual cells that are organized into rows and columns. Cells are rectangular field-like areas that can be accessed with <NEXT> and have field settings. Cells can contain numbers and text. The row at the top of the table, called the header row, is used for labeling the columns of the table.

A table resides in an anchored frame, which reserves space for the table in a document. A table can consist of a variety of rows, columns, repeating rows, divided columns, etc.



Sub-columns

A table column can be divided into subcolumns either through the column property sheet or the document auxiliary menu. In this example, a single column was divided into five subcolumns. Each subcolumn has its own property sheet and can be further divided if desired.

Test Scores				
Student	Test 1	Test 2	Final Exam	Wtd. Average
Alison	87	79	91	87.00
Byron	66	89	84	80.75
Carlson	45	40	98	70.25
Darvon	90	96	32	62.50

Sub-rows

Subrows can be enabled within a column by using the column property sheet. Subrows can be added only to a divided column.

Male Parent	Children
Lloyd Bridges	Jeff Andrew
	Beau Timothy
	Brooke Lynn
Harmon Katz	Sherri
	Douglas

In this table, the "Children" column has been made a divided column with only 1 subcolumn. Subrows were then added to this divided column.

Fill-in Rules

Fill-in Rules make it possible to take information contained in one or more fields in a table and use that information to fill in other fields. The following are examples of fill-in rule usage:

- Take information contained in a table and place it in another table.
- Take various sets of figures in a table and perform computations on them.

Fill-in Rules Example

In this example, the table calculates the wages to be paid to some employees. The *AmountPaid* column was filled in when the *Update Fields* command (from the document's auxiliary menu) was invoked.

Employee	Hourly Wage	Hours Worked	Amount Paid
Jimmy Sue	6.50	40	260.00
Betty Lou	4.25	16	68.00
Bobby Jo	7.90	47	398.95
Peggy Ann	28.00	68	2296.00

Table Name: `Wages2`
Column 2 Name: `HourlyWage` -- Column type = AMOUNT
Column 3 Name: `HoursWorked` -- Column type = AMOUNT
Column 4 Name: `AmountPaid` -- Column type = AMOUNT
Fill-in Rule for
AmountPaid Column:

```
CHOOSE
Wages2[THIS ROW].HoursWorked <= 40  ->
     Wages2[THIS ROW].HourlyWage *
     Wages2[THIS ROW].HoursWorked;
OTHERWISE  ->
     Wages2[THIS ROW].HourlyWage * 40 +
     Wages2[THIS ROW].HourlyWage *
     (Wages2[THIS ROW].HoursWorked - 40) *
     1.5
```

TableInterchangeDefs

TableInterchangeDefs is the interface for accessing and manipulating tables in documents programmatically. With this interface and DocInterchangeDefs, you can perform the following actions:

- Create a new table
- Read the contents of a table
- Add information to a table

Your Guide to Table Structure

TableInterchangeDefs (TI) defines the data structure for VP tables. This diagram shows some of the highlights of the beast:

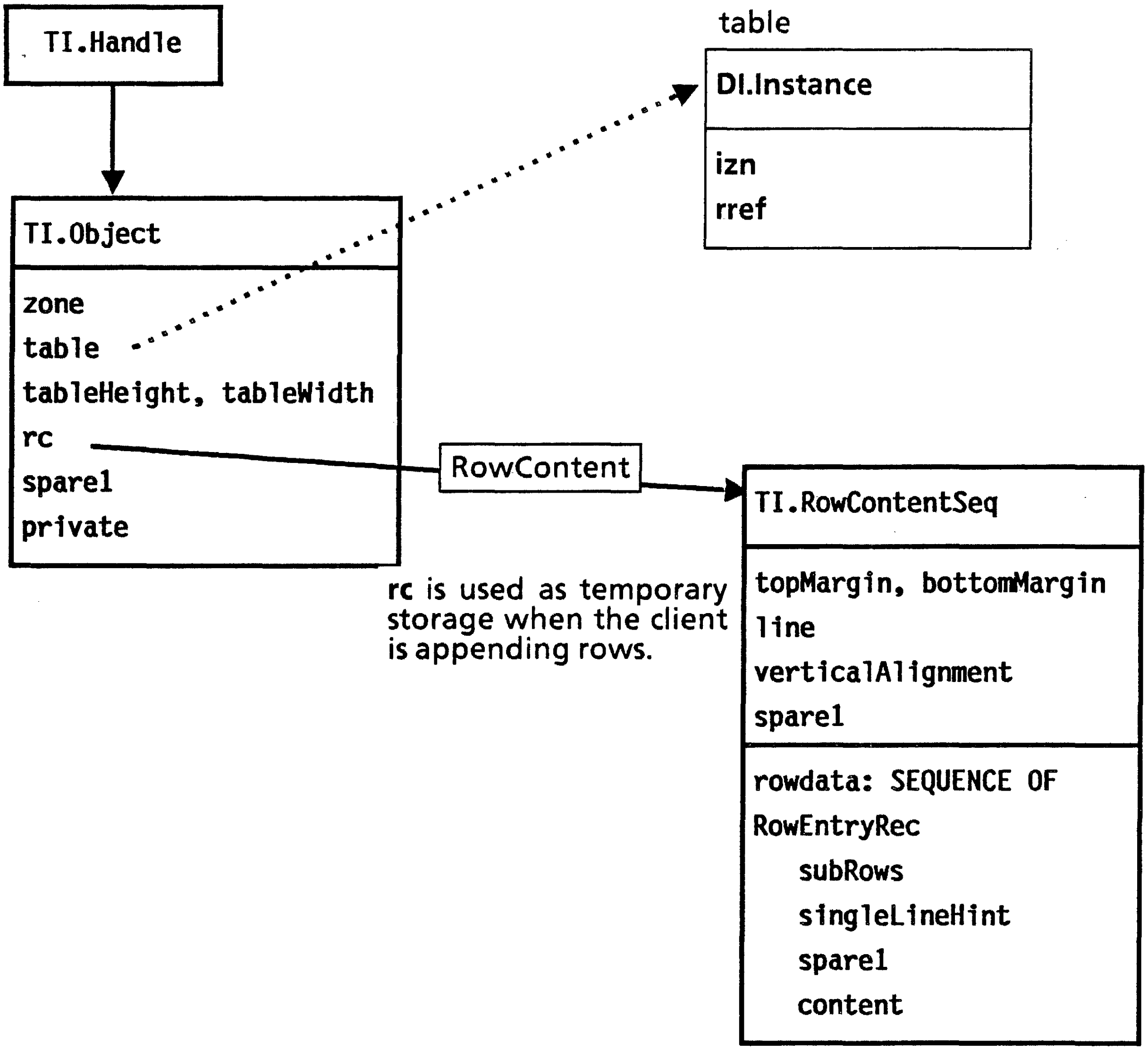


Table Creation

Call TableInterchangeDefs.StartTable to create a new table:

```
TI.StartTable: PROC[
  doc: DI.Doc,
  props: TI.TableProps,
  c: TI.ColumnInfo]
  RETURNS[h: TI.Handle];
```

doc is the document in which the table will be created. props and c describe the same characteristics as those that the user can set in the property sheets; these two parameters are defined later.

Opening an Existing Table

Call TableInterchangeDefs.StartExistingTable to obtain a handle to an existing table:

```
TI.StartExistingTable: PROC[
  table: DI.Instance,
  hi: TI.HeaderInfo ← NIL,
  rowPropsSource: NATURAL ← 0,
  deleteExistingRows: BOOL ← TRUE,
  numberOfRowsHint: NATURAL ← 0]
  RETURNS[h: TI.Handle];
```

The parameter table (the table object) is often obtained from a call to TableSelectionDefs.TableFromSelection (TSD), which returns the current selection as a table.

```
  TSD.TableFromSelection: PROC RETURNS[DI.Instance];
```

hi (header info) contains the desired properties for the table headers. When defaulted, the existing column headers are used.

rowPropsSource is an index of a row in the table; this is the row from which row default properties are taken (rows are numbered from 0 - n).

deleteExistingRows indicates whether the implementation should delete the contents of the table before adding new information.

numberOfRowsHint is a hint about the number of rows that the table might contain; this helps the system allocate the space for the table.

Table Properties

`TableInterchangeDefs.TableProps` points to a record of table properties. These properties can be set by a user via table property sheets.

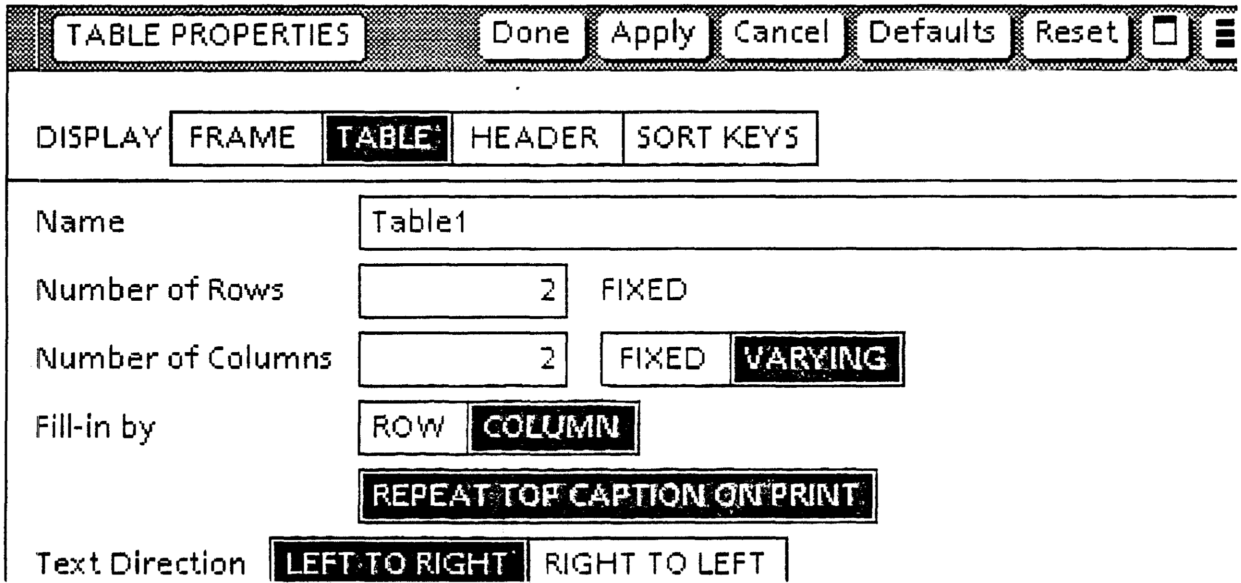
```
TI.TableProps: TYPE = LONG POINTER TO TI.TablePropsRec;

TI.TablePropsRec: TYPE = RECORD[
  name: XString.Reader,
  fillinByRow, fixedRows, fixedColumns: BOOL,
  numberOfColumns, numberOfRows: NATURAL,
  visibleHeader, repeatHeader,
  repeatTopCaption, repeatBottomCaption: BOOL,
  borderLine, dividerLine: TI.Line,
  horizontalAlignment: TI.HeaderAlignment,
  headerVerticalAlignment: TI.VerticalAlignment,
  topHeaderMargin, bottomHeaderMargin: LONG CARDINAL,
  sortKeys: TI.SortKeys,
  spare1: LONG CARDINAL];

TI.nullTableProps: TI.TablePropsRec = [...];
```

micas

Table Properties (cont'd)

```
TI.HorizontalAlignment: TYPE = MACHINE DEPENDENT {
  left(0), center(1), right(2), decimal(3)};

TI.HeaderAlignment: TYPE = TI.HorizontalAlignment[left..right];

TI.VerticalAlignment: TYPE = MACHINE DEPENDENT {
  flushtop(0), centered(1), flushbottom(2)};

TI.SortKeys: TYPE = LONG POINTER TO TI.SortKeysRec;
TI.SortKeysRec: TYPE = RECORD[
  length: CARDINAL,
  spare1: LONG CARDINAL,
  keys: SEQUENCE maxLength: CARDINAL OF TI.SortKey];

TI.SortKey: TYPE = RECORD[
  columnName: XString.Reader,
  sortOrder: XString.SortOrder,
  ascending: BOOL,
  spare1: LONG CARDINAL];

TI.nullSortKey: TI.SortKey = [...];
```

Table Properties (cont'd)



DISPLAY FRAME [✓ TABLE] HEADER SORT KEYS

Name: Table1

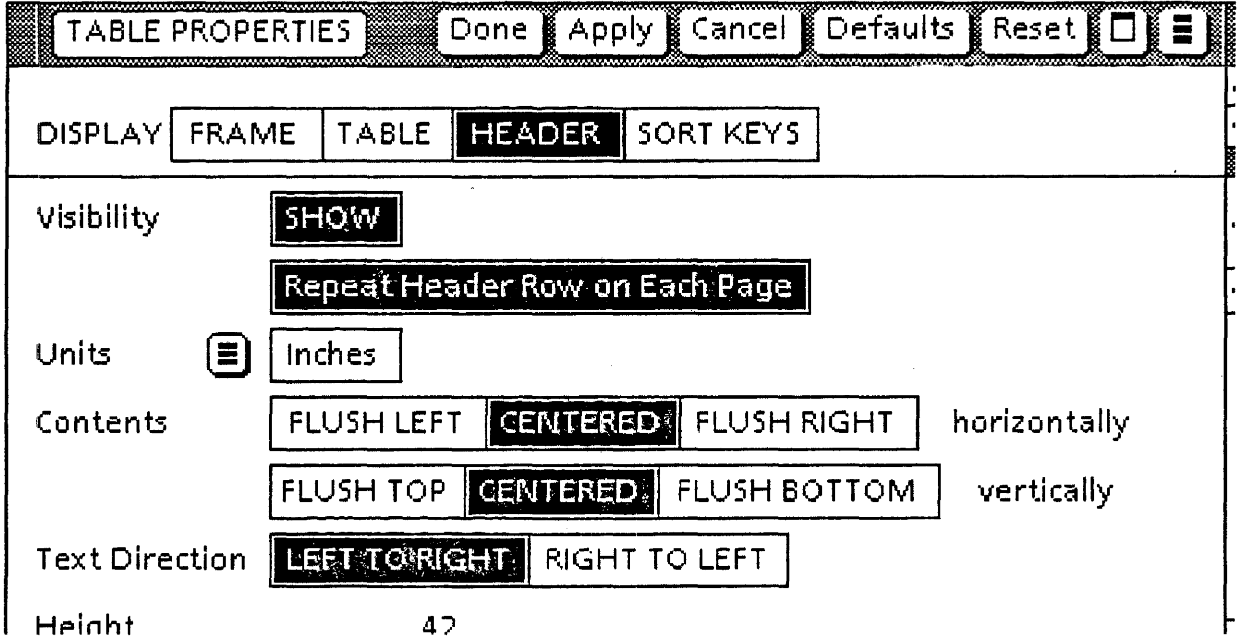
Number of Rows: 2 FIXED

Number of Columns: 2 FIXED [✓ VARYING]

Fill-in by: ROW [✓ COLUMN]

[✓ REPEAT TOP CAPTION ON PRINT]

Text Direction: [✓ LEFT TO RIGHT] RIGHT TO LEFT



DISPLAY FRAME TABLE [✓ HEADER] SORT KEYS

Visibility: [✓ SHOW]

[✓ Repeat Header Row on Each Page]

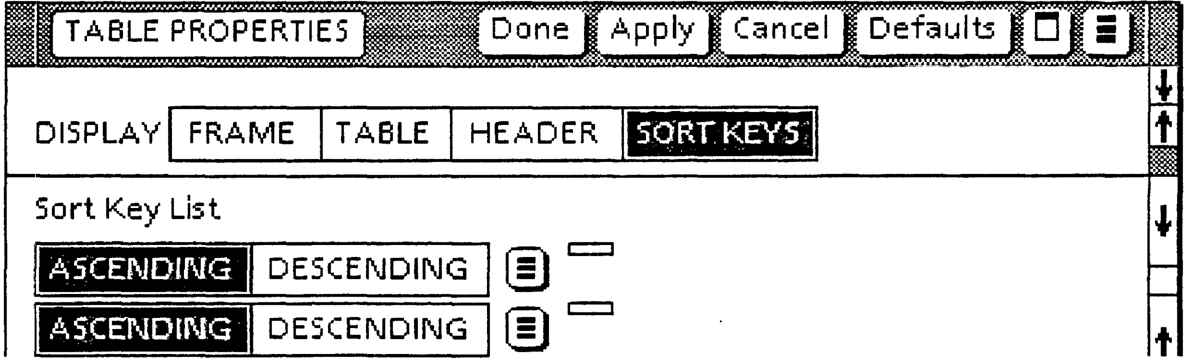
Units: Inches

Contents: FLUSH LEFT [✓ CENTERED] FLUSH RIGHT horizontally

FLUSH TOP [✓ CENTERED] FLUSH BOTTOM vertically

Text Direction: [✓ LEFT TO RIGHT] RIGHT TO LEFT

Height 42



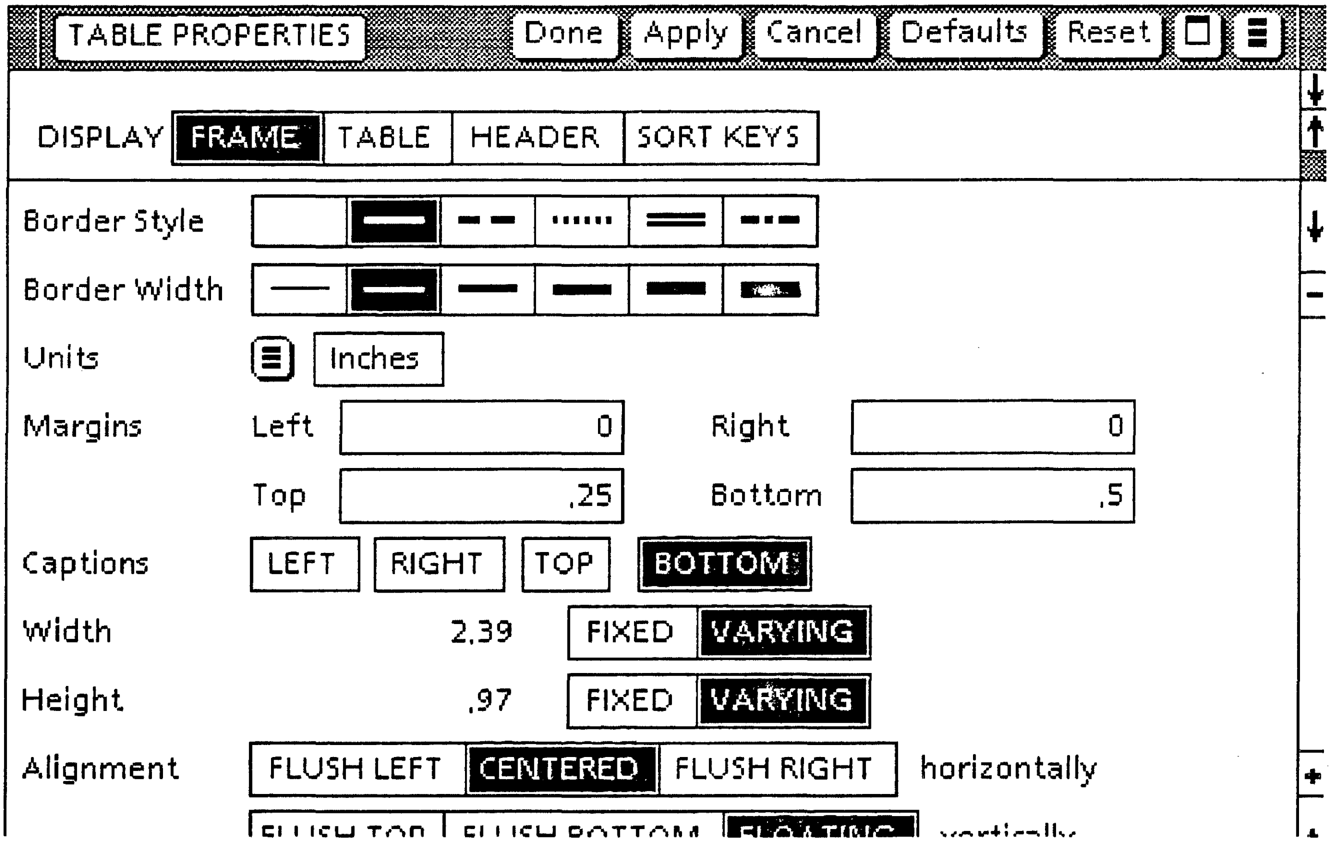
DISPLAY FRAME TABLE HEADER [✓ SORT KEYS]

Sort Key List

[✓ ASCENDING] DESCENDING

[✓ ASCENDING] DESCENDING

Table Properties (cont'd)



DISPLAY [✓ FRAME] TABLE HEADER SORT KEYS

Border Style

Border Width

Units Inches

Margins Left 0 Right 0

Top .25 Bottom .5

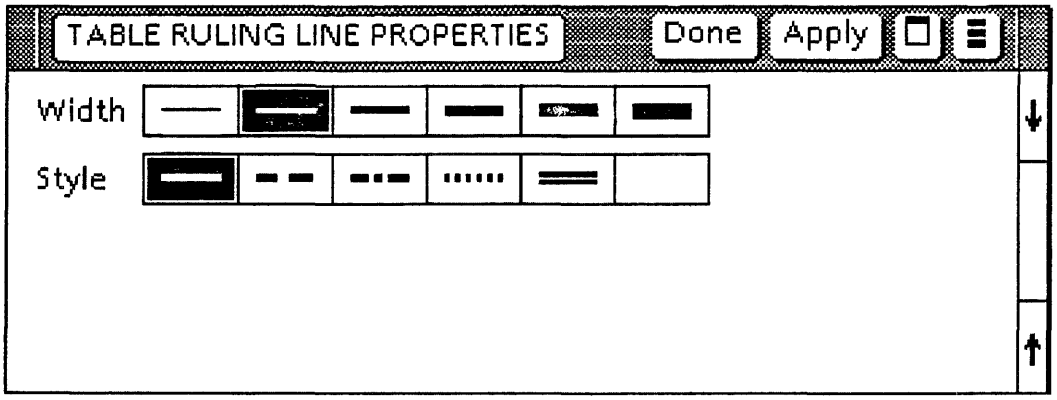
Captions LEFT RIGHT TOP [✓ BOTTOM]

Width 2.39 FIXED [✓ VARYING]

Height .97 FIXED [✓ VARYING]

Alignment FLUSH LEFT [✓ CENTERED] FLUSH RIGHT horizontally

FLUSH TOP FLUSH BOTTOM [✓ FLOATING] vertically



Width

Style

Header Properties

```
TI.HeaderInfo: TYPE = LONG POINTER TO TI.HeaderInfoSeq;

TI.HeaderInfoSeq: TYPE = RECORD[
  SEQUENCE length: CARDINAL OF TI.HeaderEntryRec];

TI.HeaderEntryRec: TYPE = RECORD[
  subHeaders: TI.HeaderInfo,
  line: TI.Line,
  singleLineHint: BOOL,
  spare1: LONG CARDINAL,
  content: TI.EntryContent];  -- EntryContent explained later

TI.nullHeaderEntry: TI.HeaderEntryRec = [...];

TI.Line: TYPE = RECORD[linestyle: TI.Linestyle, linewidth: TI.Linewidth];

TI.Linestyle: TYPE = MACHINE DEPENDENT {
  none(0), solid, dashed, dotted, double, broken, firstAvailable,
  lastAvailable(255)};

TI.Linewidth: TYPE = MACHINE DEPENDENT {
  w1(0), w2(1), w3(2), w4(3), w5(4), w6(5)};

TI.nullLine: TI.Line = [linestyle: solid, linewidth: w1];
```

Appending Information to Tables

Call TableInterchangeDefs.AppendRow to add a row and its data to a table:

```
TI.AppendRow: PROC[h: TI.Handle, rc: TI.RowContent];
```

AppendRow adds the row described by rc to the table described by h, which was probably obtained from StartTable or StartExistingTable.

The parameter rc is typically obtained from the rc field within the table object that h points to. In fact, the rc field within Object exists for that purpose: you fill in rc within Object and then call AppendRow.

This procedure can raise the error DI.Error[documentFull] if this row will not fit in the document. If this error occurs, clients are expected to CONTINUE the error and then make a call to FinishTable.

```
TI.Error: SIGNAL[type: TI.ErrorType];

TI.ErrorType: TYPE = MACHINE DEPENDENT {
  tableTooWide, tableHeaderTooTall,     -- can be raised by StartTable
  tableTooTall,     -- can be raised by AppendRow
  firstAvailable, lastAvailable(255)};
```

no AppendColumn proc

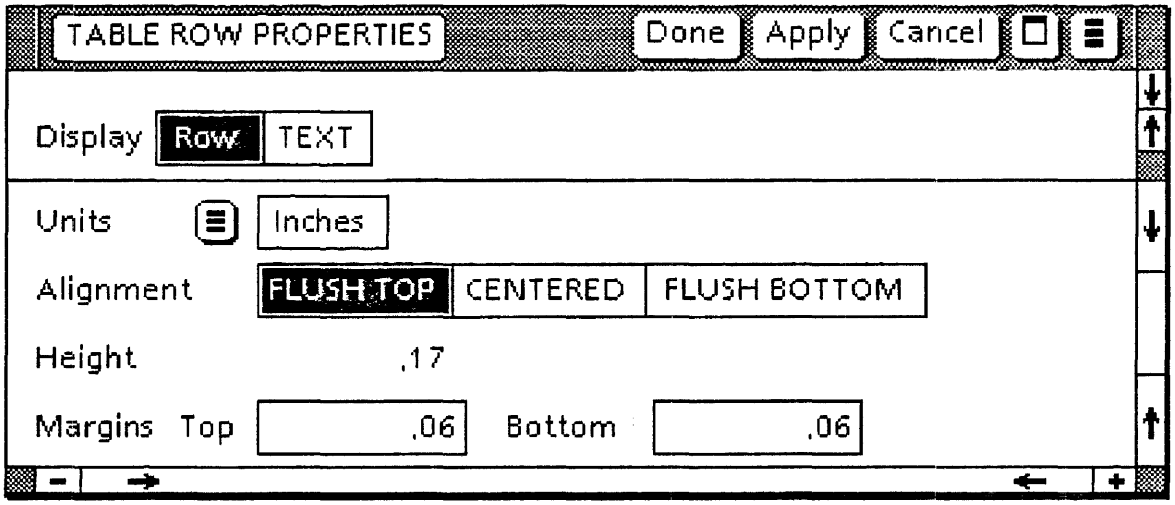
RowContent Structure

```
TI.RowContent: TYPE = LONG POINTER TO TI.RowContentSeq;

TI.RowContentSeq: TYPE = RECORD[
  topMargin, bottomMargin: LONG CARDINAL ← 0,
  line: TI.Line ← [solid, w2],
  verticalAlignment: TI.VerticalAlignment ← flushtop,
  spare1: LONG CARDINAL ← 0,
  rowdata: SEQUENCE length: CARDINAL OF TI.RowEntryRec];

TI.RowEntryRec: TYPE = RECORD[
  subRows: TI.SubRows,
  singleLineHint: BOOL,
  spare1: LONG CARDINAL,
  content: TI.EntryContent];

TI.nullRowEntry: TI.RowEntryRec = [...];
```

RowContent Structure (cont'd)

```
TI.SubRows: TYPE = LONG POINTER TO TI.SubRowsRec;

TI.SubRowsRec: TYPE = RECORD[
  length: CARDINAL,
  spare1: LONG CARDINAL ← 0,
  rows: SEQUENCE maxLength: CARDINAL OF TI.RowContent];

TI.EntryContent: TYPE = RECORD[
  SELECT mode: * FROM
    read => [   --when enumerating
      obtainTextProc: TI.ObtainTextProc,
      obtainTextData: TI.ObtainTextData],
    write => [ -- when appending a row
      fillInTextProc: TI.FillInTextProc ← NIL,
      clientData: LONG POINTER ← NIL],
    ENDCASE];

TI.ObtainTextProc: TYPE = PROC[obtainTextData: TI.ObtainTextData]
  RETURNS[text: TextInterchangeDefs.Text];

TI.ObtainTextData: TYPE[4];

TI.FillInTextProc: TYPE = PROC[  -- client fills in text when called
  text: TextInterchangeDefs.Text, clientData: LONG POINTER];
```

RowContent Structure (cont'd)



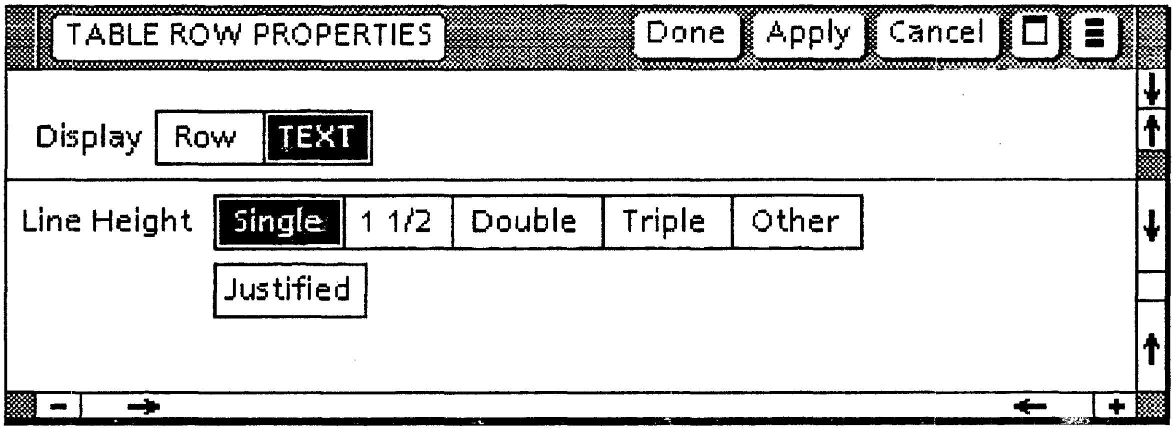
Display [✓ Row] TEXT

Units Inches

Alignment [✓ FLUSH TOP] CENTERED FLUSH BOTTOM

Height .17

Margins Top .06 Bottom .06



Display Row [✓ TEXT]

Line Height [✓ Single] 1 1/2 Double Triple Other

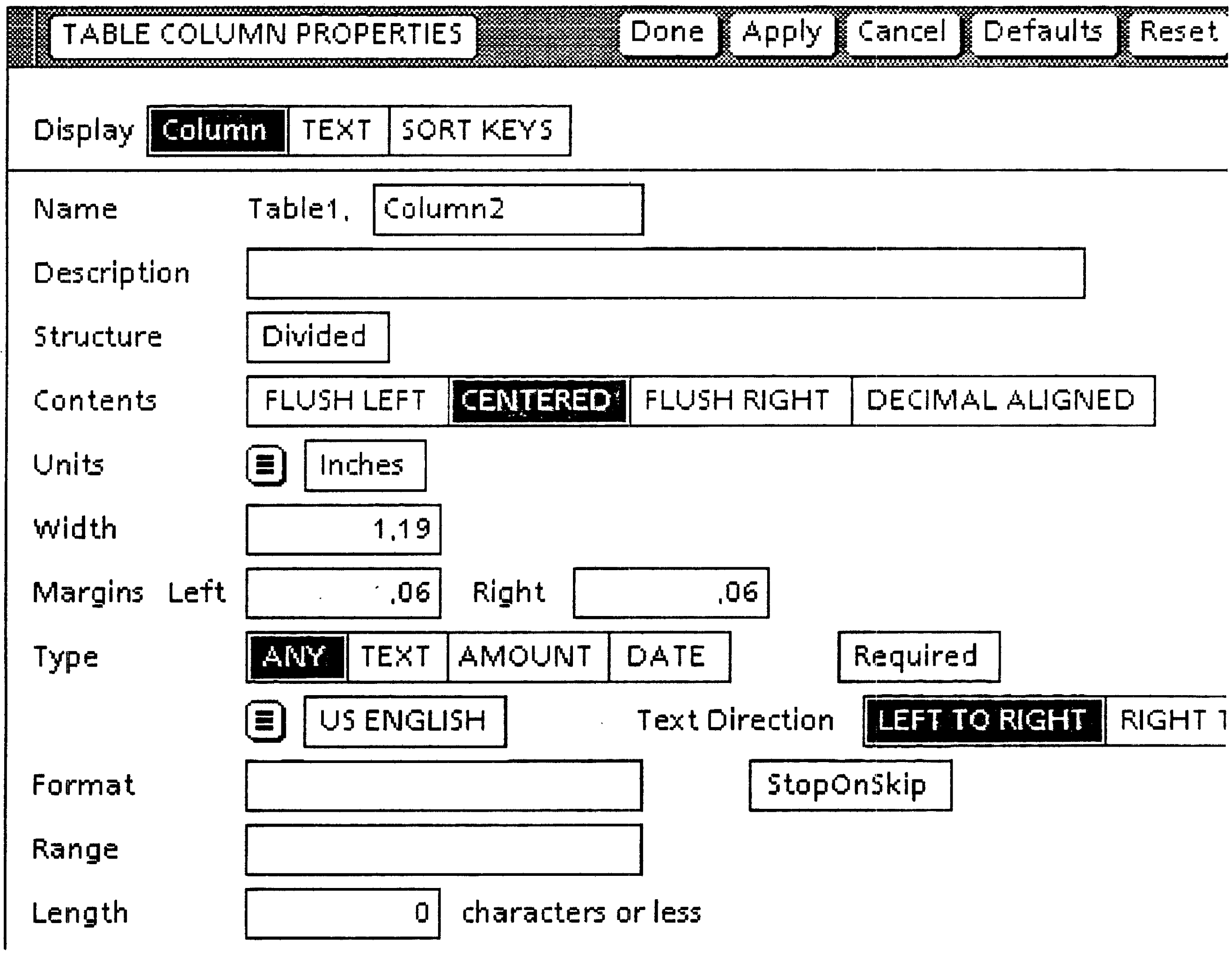
Justified

ColumnInfo

TableInterchangeDefs.ColumnInfo points to a record containing a sequence of column properties. These properties can be set by a user via table property sheets.

```
TI.ColumnInfo: TYPE = LONG POINTER TO TI.ColumnInfoSeq;
TI.ColumnInfoSeq: TYPE = RECORD[SEQUENCE length: CARDINAL OF TI.ColumnInfoRec];

TI.ColumnInfoRec: TYPE = RECORD [
  headerEntryRec: TI.HeaderEntryRec,
  name, description: XString.Reader,
  divided: BOOL,
  subcolumns: NATURAL,
  repeating: BOOL,
  subcolumnInfo: TI.ColumnInfo,
  alignment: TI.HorizontalAlignment,
  tabOffset, -- Micas! (different from DIP.TabStop)
  width, leftMargin, rightMargin: LONG CARDINAL,
  type: DIP.FieldChoiceType,
  ...];

TI.nullColumnInfo: TI.ColumnInfoRec = [...];
```

ColumnInfo (cont'd)



Display [✓ Column] TEXT SORT KEYS

Name Table1, Column2

Description

Structure Divided

Contents FLUSH LEFT [✓ CENTERED] FLUSH RIGHT DECIMAL ALIGNED

Units Inches

Width 1.19

Margins Left .06 Right .06

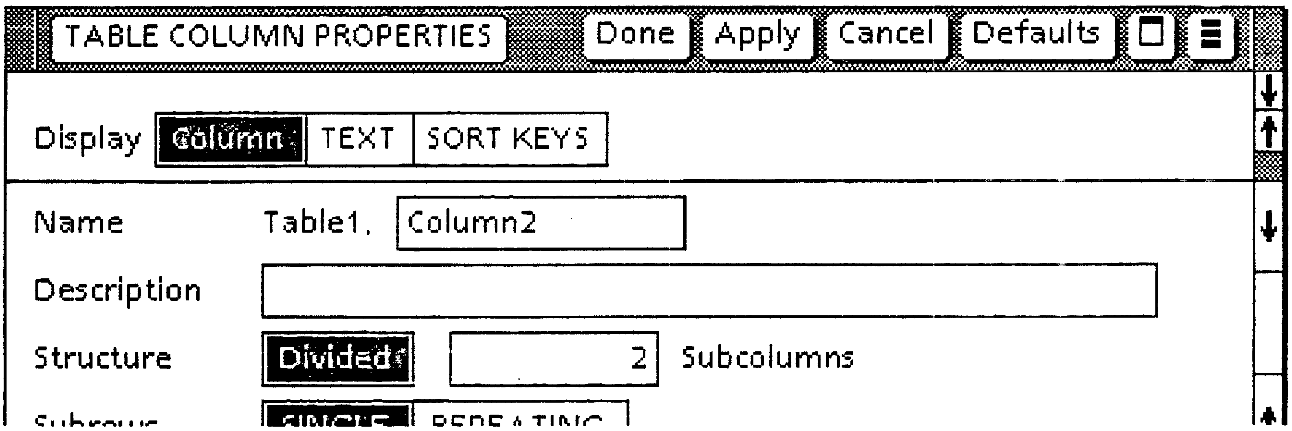
Type [✓ ANY] TEXT AMOUNT DATE Required

US ENGLISH Text Direction [✓ LEFT TO RIGHT] RIGHT T

Format StopOnSkip

Range

Length 0 characters or less



Display [✓ Column] TEXT SORT KEYS

Name Table1, Column2

Description

Structure [✓ Divided] 2 Subcolumns

Subrows [✓ SINGLE] REPEATING

Finishing a Table

You should call TableInterchangeDefs.FinishTable when you are through editing a table.

```
TI.FinishTable: PROC[h: TI.Handle] RETURNS[
  table: DI.Instance,
  tableWidth, tableHeight: LONG CARDINAL]; -- in micas
```

Pass table as the content argument to DocInterchangeDefs.AppendAnchoredFrame. FinishTable deletes h.zone.

Reading a Table

You can read a table's content with TableInterchangeDefs.Enumerate.

```
TI.EnumerateTable: PROC[
  table:DI.Instance,
  procs: TI.EnumProcs,
  clientData: LONG POINTER ← NIL];
```

```
TI.EnumProcs: TYPE = LONG POINTER TO TI.EnumProcsRec;
```

```
TI.EnumProcsRec: TYPE = RECORD[
  tableProc: TI.TableProc ← NIL,          -- called once
  columnsProc: TI.ColumnsProc ← NIL,      -- called once
  rowProc: TI.RowProc ← NIL];             -- called for each row in table
```

Enumerate calls **tableProc** and **columnsProc** to get the table and column properties, respectively. **Enumerate** then calls **rowProc** for each row in the table.

Enumeration Call-back Procedures

```
TI.TableProc: TYPE = PROC[
  clientData: LONG POINTER,
  props: TI.TableProps]
  RETURNS[stop: BOOL ← FALSE];

TI.ColumnsProc: TYPE = PROC[
  clientData: LONG POINTER,
  columns: TI.ColumnInfo]
  RETURNS[stop: BOOL ← FALSE];

TI.RowProc: TYPE = PROC[
  clientData: LONG POINTER,
  content: TI.RowContent]
  RETURNS[stop: BOOL ← FALSE];
```

Enumeration Call-back Procedures

There are a few different units of measurements in these interfaces and there will be times that you will need to convert between, for example, 1/72s and micas. The interface `UnitConversion` exists for this reason. UnitConversion has some predefined units and 2 convert procs that will convert integers or reals to whatever units you plug in to the procedures.

```
UnitConversion.Units: TYPE = MACHINE DEPENDENT {
  inch(0), mm, cm, mica, point, pixel, pica, didotPoint, cicero,
  seventySecondOfAnInch, last(15)};
  -- point is a printer dot, pixel is a screen dot, pica = 12 points

UnitConversion.ConvertInteger: PROC[
  n: LONG INTEGER,
  inputUnit, outputUnits: UnitConversion.Units]
  RETURNS[LONG INTEGER];

UnitConversion.ConvertReal: PROC[
  n: XReal.Number,
  inputUnit, outputUnits: UnitConversion.Units]
  RETURNS[XReal.Number];
```

Table Example

Here is an example of a basic program that runs from the Attention Menu. It registers two commands: MakeTable, which creates a new document with a table, and AddToTable, which adds four new rows to the selected table.

```
DIRECTORY
  Attention,
  BWSZone,
  DocInterchangeDefs,
  DocInterchangePropsDefs,
  Heap,
  MenuData,
  NSFile,
  StarDesktop,
  TableInterchangeDefs,
  TableSelectionDefs,
  TextInterchangeDefs,
  XString;

  TableExample: PROGRAM
  IMPORTS Attention, BWSZone, DocInterchangeDefs, Heap, MenuData, NSFile,
StarDesktop, TableInterchangeDefs, TableSelectionDefs, TextInterchangeDefs,
XString = {

  tableWidth: CARDINAL = 1600;       -- micas
  headerMargin: CARDINAL = 35 * 9;   -- micas; margin should be 9 seventySecondsOfAnInch
  rowMargin: CARDINAL = 100;

<<Menu Proc for Create Table command. Creates new document, creates new table, appends table to
document, and then adds document to desktop.>>
  MakeDocument: MenuData.MenuProc = {
    rows, columns: CARDINAL ← 3;  -- arbitrary
    doc: DocInterchangeDefs.Doc ← DocInterchangeDefs.StartCreation[].doc;
    table: DocInterchangeDefs.Instance = BuildSimpleTable[doc, rows, columns];
    props: DocInterchangePropsDefs.FramePropsRecord ←
      DocInterchangePropsDefs.nullFrameProps;
    props.frameDims ← [tableWidth, tableWidth];
    [ ] ← DocInterchangeDefs.AppendAnchoredFrame[
      to: doc,
      type: table,
      anchoredFrameProps: @props,
      content: table];
    AddFileToDeskTop[doc];
    };
  << Create table inside doc with specified number of rows and columns. The content will be the string
"abc.">>
  BuildSimpleTable: PROC[doc: DocInterchangeDefs.Doc, rows, columns: CARDINAL]
   RETURNS[table: DocInterchangeDefs.Instance ← DocInterchangeDefs.instanceNil] =
    {
    h: TableInterchangeDefs.Handle;
    contentString: XString.ReaderBody ← XString.FromSTRING["abc"L];
    c: TableInterchangeDefs.ColumnInfo ← BWSZone.shortLifetime.NEW[
      TableInterchangeDefs.ColumnInfoSeq[columns]];
    props: TableInterchangeDefs.TablePropsRec ← [
      name: NIL,
      fillinByRow: TRUE,
      fixedRows: FALSE,
      fixedColumns: TRUE,
      numberOfColumns: columns,
      numberOfRows: rows,
      visibleHeader: TRUE,
      repeatHeader: TRUE,
      repeatTopCaption: TRUE,
      repeatBottomCaption: TRUE,
      borderLine: [none, w1],
      dividerLine: [solid, w4],
      horizontalAlignment: center,
      headerVerticalAlignment: centered,
      topHeaderMargin: headerMargin,
      bottomHeaderMargin: headerMargin,
      sortKeys: NIL,
      spare1: 0];
    FOR i: CARDINAL IN [0..columns) DO
      c[i] ← TableInterchangeDefs.nullColumnInfo;
      c[i].width ← tableWidth;
    ENDLOOP;
    -- start creating table
    h ← TableInterchangeDefs.StartTable[doc: doc, props: @props, c: c];
    BWSZone.shortLifetime.FREE[@c];
    -- set row props and content
    h.rc.topMargin ← rowMargin;
    h.rc.bottomMargin ← rowMargin;
    FOR i: CARDINAL IN [0..rows) DO
      FOR j: CARDINAL IN [0..columns) DO
        h.rc[j] ← [
          subRows: NIL,
          singleLineHint: FALSE,
          spare1: 0,
          content: [write[fillInTextProc: FillInText,
                          clientData: @contentString]] ];
        ENDLOOP;
      TableInterchangeDefs.AppendRow[h, h.rc];
      ENDLOOP;
    RETURN[TableInterchangeDefs.FinishTable[h].table];
    };
```

```
FillInText: TableInterchangeDefs.FillInTextProc = {
<<PROC [text: TextInterchangeDefs.Text, clientData: LONG POINTER];>>
  r: XString.Reader ← NARROW[clientData, XString.Reader];
  TextInterchangeDefs.AppendTextToText[
    to: text, text: r, textEndContext: XString.unknownContext];
  };

<<Menu Proc for Add To Table command. Just adds four new blank rows to selected table.>>
AddToTable: MenuData.MenuProc = {
  h: TableInterchangeDefs.Handle ← NIL;
  table: DocInterchangeDefs.Instance = TableSelectionDefs.TableFromSelection[];
  -- if current selection is not a table, then return. Otherwise, add new rows to it.
  -- If doc is not editable, then return.
  IF table = DocInterchangeDefs.instanceNil THEN RETURN
  ELSE {
    h ← TableInterchangeDefs.StartExistingTable[
      table: table, deleteExistingRows: FALSE    -- catch error if doc is not editable
      ! TableInterchangeDefs.TableError => GOTO Exit];
    THROUGH [0..4) DO
      TableInterchangeDefs.AppendRow[h, h.rc];
    ENDLOOP;
    };
  [] ← TableInterchangeDefs.FinishTable[h];
  EXITS Exit => NULL;
  };

AddFileToDeskTop: PROC[doc: DocInterchangeDefs.Doc] = {
  docFile: NSFile.Handle ← DocInterchangeDefs.FinishCreation[@doc].docFile;
  refDoc: NSFile.Reference = NSFile.GetReference[docFile];
  refDt: NSFile.Reference = StarDesktop.GetCurrentDesktopFile[];
  fileDt: NSFile.Handle = NSFile.OpenByReference[refDt];
  NSFile.Move[docFile, fileDt];
  NSFile.Close[fileDt];
  NSFile.Close[docFile];
  StarDesktop.AddReferenceToDesktop[refDoc];
  };

Init: PROC = {
  makeTable: XString.ReaderBody ← XString.FromSTRING["MakeTable"L];
  addToTable: XString.ReaderBody ← XString.FromSTRING["AddToTable"L];
  Attention.AddMenuItem[MenuData.CreateItem[
            zone: BWSZone.permanent,
            name: @makeTable,
            proc: MakeDocument]];
  Attention.AddMenuItem[MenuData.CreateItem[
            zone: BWSZone.permanent,
            name: @addToTable,
            proc: AddToTable]];
  };

Init[];

}...
```

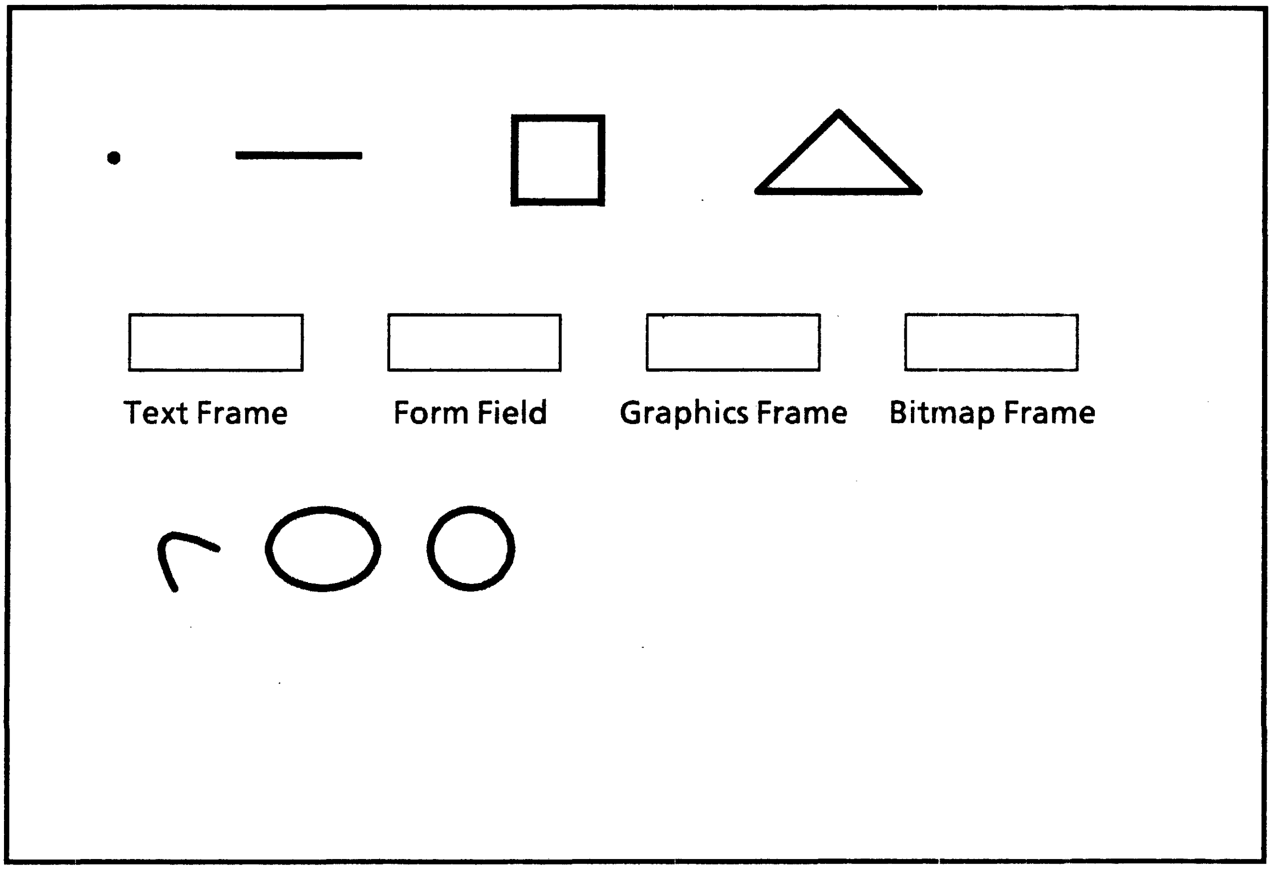
Day 4: GraphicsInterchangeDefs

Outline

I. GraphicsInterchangeDefs
   A. ViewPoint Graphics Overview
   B. Adding Graphics to a Document
      1. Starting & Finishing Graphic Frames
      2. Appending objects
         a. Box within the frame
         b. Frame properties
         c. Geometric objects
         d. Text objects
         e. Other objects
   C. Appending to text and captions
   D. Enumerating Graphic Frames
   E. Enumerating text

ViewPoint Graphics

ViewPoint graphics must appear in a graphics frame. The following is a graphics frame with all possible graphics objects (except an image frame and CUSP button):



What's GraphicsInterchangeDefs For?

GraphicsInterchangeDefs is the main interface for creating and enumerating graphics programmatically in ViewPoint. It is used in conjunction with **DocInterchangeDefs** to both add and enumerate graphics in VP documents. It uses types defined in **DocInterchangePropsDefs** and other interfaces.

In the following slides, **GI** may be substituted for GraphicsInterchangeDefs, **DI** may be substituted for DocInterchangeDefs, and **DIP** for DocInterchangePropsDefs.

Adding Graphics to a Document

Graphics can be added to a document only during the process of a `DocInterchangeDefs.StartCreation`. Graphics cannot be appended to an existing document.

Use the following steps to append graphics to a new document. Each step is covered in more detail on the following pages:

1. Call `DI.StartCreation` to get a document handle (`doc`).
2. Call `GI.StartGraphics[doc]` to get an anchored frame handle (`h`).
3. Call `GI.Add*[h]` to add graphics to that anchored frame.
4. If you want to add a container to the anchored frame:
   A. Call `GI.StartGraphicsFrame`, `GI.StartCluster`, or `GI.StartButton` to get a handle for the container (`gfh`, `ch`, or `bfh`).
   B. Call `GI.Add*[gfh`, `ch`, or `bfh]` to add graphics to the container.
   C. Call `GI.FinishGraphicsFrame`, `GI.FinishCluster`, or `GI.FinishButton`.
5. Call `GI.FinishGraphics[h]` to complete the anchored frame and get an object of type `DI.Instance` (`graphics`).
6. Call `DI.AppendAnchoredFrame[graphics]`.
7. Call `DI.FinishCreation[@doc]`.

Starting a Graphics Frame

Call `GraphicsInterchangeDefs.StartGraphics` to create a graphics frame:

```
GI.StartGraphics: PROC[doc: DI.Doc] RETURNS[h: GI.Handle];
```

```
GI.Handle: TYPE = LONG POINTER TO GI.Object;
GI.Object: TYPE;
```

The returned handle points to an opaque type. The handle is used by other procedures to add objects to the graphics frame.

Starting a Nested Frame

Among the objects that can be added to a graphics frame are nested frames. A `Handle` for each of these frames allows the client to add objects to the nested frames later. This `Handle` is of the same type returned by `StartGraphics`. The return parameters are the handle to the nested frame and the handles to any captions the caller wanted.

1. Call `GraphicsInterchangeDefs.StartGraphicsFrame` to create a nested graphics frame:

```
GI.StartGraphicsFrame: PROC[
  h: GI.Handle,
  box: GI.Box,
  frameProps: GI.ReadonlyFrameProps,
  name, description: XString.Reader ← NIL, -- extra props, like tables
  spareProps: LONG POINTER ← NIL,
  wantTopCaptionHandle, wantBottomCaptionHandle,
  wantLeftCaptionHandle, wantRightCaptionHandle: BOOLEAN ← FALSE]
  RETURNS[
    gfh: GI.Handle,
    topCaption, bottomCaption,
    leftCaption, rightCaption: DI.Caption];
```

`h` is the handle to the anchored graphics frame in which this nested frame will reside.

`box` is the size and frame-relative location of the nested frame. Units are specified in micas.

Frame Properties

All frames have frame properties. These properties define the border, margins, caption capabilities, and expansion properties of a frame.

```
GI.FrameProps: TYPE = LONG POINTER TO GI.FramePropsRec;
GI.ReadonlyFrameProps: TYPE = LONG POINTER TO READONLY GI.FramePropsRec;
```

```
GI.FramePropsRec: TYPE = RECORD[
  brush: GI.Brush,
  fixedShape: BOOL,
  margins: ARRAY GI.Side OF LONG CARDINAL,
  captionContent: ARRAY GI.Side OF DI.Caption,
  spare1: LONG CARDINAL];
```

`captionContent` is ignored in all `Add*` operations and is only meaningful during enumeration.

```
GI.Brush: TYPE = RECORD[
  wthBrush: LONG CARDINAL,        -- width of the brush in micas
  styleBrush: GI.StyleBrush];
```

```
GI.StyleBrush: TYPE = MACHINE DEPENDENT {
  invisible(0), solid(1), dashed(2), dotted(3), double(4), broken(5), (15)};
```

```
GI.Side: TYPE = {top, bottom, left, right};
```

Starting a Nested Frame (cont'd)

2. Call GraphicsInterchangeDefs.StartCluster to create a set of cluster objects:

```
GI.StartCluster: PROC[h: GI.Handle, box: GI.Box]
  RETURNS[ch: GI.Handle];
```

h is the handle to the anchored graphics frame in which this cluster will reside.

box is the size and frame-relative location of the nested frame. Units are specified in micas.

ch is the handle to the cluster. Subsequent calls to add objects to the cluster will use ch.

Starting a Nested Frame (cont'd)

3. Call GraphicsInterchangeDefs.StartButton to create a CUSP button in a graphics frame:

```
GI.StartButton: PROC[
  h: GI.Handle,
  box: GI.Box,     -- if buttonProps is NIL then a unique name is generated
  buttonProps: GI.ReadonlyButtonProps,
  frameProps: GI.ReadonlyFrameProps,
  wantProgramHandle, wantTopCaptionHandle,
  wantBottomCaptionHandle, wantLeftCaptionHandle,
  wantRightCaptionHandle: BOOLEAN ← FALSE]
  RETURNS[
    bfh: GI.Handle,
    buttonProgram: GI.ButtonProgram,
    topCaption, bottomCaption,
    leftCaption, rightCaption: DI.Caption];
```

h is the handle to the anchored graphics frame in which this button will reside.

box is the size and frame-relative location of the nested frame. Units are specified in micas.

```
DI.ReadonlyButtonProps: TYPE = LONG POINTER TO READONLY DI.ButtonPropsRec;

DI.ButtonPropsRec: TYPE = RECORD[
  name: XString.Reader,
  spare1: LONG CARDINAL];
```

Note: Graphics objects may be inserted inside a button by calling the Add* procedures and passing the button handle (bfh).

Finishing Containers

When you have finished adding objects to a container, you must call the appropriate procedure to notify the graphics implementation that you are not going to add any more objects.

If your container was started with:	...then finish it with:
StartGraphics	FinishGraphics
StartGraphicsFrame	FinishGraphicsFrame
StartCluster	FinishCluster
StartButton	FinishButton

```
GI.FinishGraphics: PROC[h: GI.Handle]
  RETURNS[graphics: DI.Instance]; -- return Instance to DI.AppendAnchoredFrame

GI.FinishGraphicsFrame: PROC[gfh: GI.Handle];

GI.FinishCluster: PROC[ch: GI.Handle];

GI.FinishButton: PROC[bfh: GI.Handle];
```

Adding Objects to a Graphics Frame

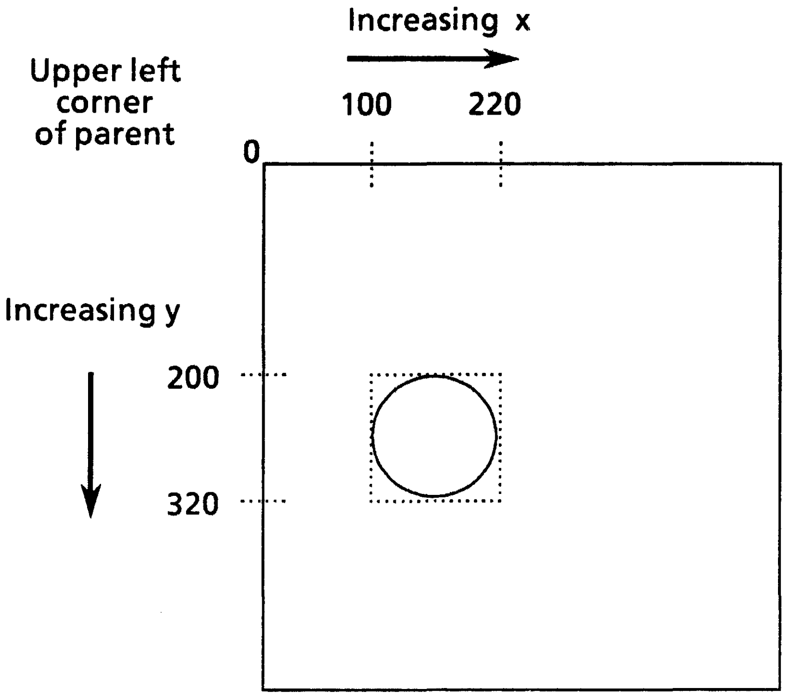
All of the following procedures add an object to a graphics frame (anchored or nested), button, or cluster, as defined by the handle passed in. A **box** parameter is also passed to each procedure that defines where the object should appear in the frame, button, or cluster.

```
GI.AddPoint: PROC[
  h: GI.Handle, box: GI.Box, pointProps: GI.ReadonlyPointProps];
GI.AddLine: PROC[h: GI.Handle, box: GI.Box, lineProps: GI.ReadonlyLineProps];
GI.AddCurve: PROC[
  h: GI.Handle, box: GI.Box, curveProps: GI.ReadonlyCurveProps];
GI.AddEccentricCurve: PROC[
 h: GI.Handle, box: GI.Box, eccentricCurveProps: GI.ReadonlyEccentricCurveProps];
GI.AddEllipse: PROC[
  h: GI.Handle, box: GI.Box, ellipseProps: GI.ReadonlyEllipseProps];
GI.AddRectangle: PROC[
  h: GI.Handle, box: GI.Box, rectangleProps: GI.ReadonlyRectangleProps];
GI.AddTriangle: PROC[
  h: GI.Handle, box: GI.Box, triangleProps: GI.ReadonlyTriangleProps];
GI.AddTextFrame: PROC[
  h: GI.Handle, box: GI.Box, frameProps: GI.ReadonlyFrameProps, ...];
GI.AddFormField: PROC[
  h: GI.Handle, box: GI.Box, fieldProps: GI.ReadonlyFieldProps, ...];
GI.AddBitmap: PROC[
  h: GI.Handle, box: GI.Box, bitmapProps: GI.ReadonlyBitmapProps...];
GI.AddImage: PROC[
  h: GI.Handle, box: GI.Box, imageProps: GI.ReadonlyImageProps, ...];
GI.AddOther: PROC[h: GI.Handle, box: GI.Box, instance: DI.Instance];
```

Box

All **GraphicsInterchangeDefs.Add*** procedures take a **GraphicsInterchangeDefs.Box** as a parameter. This parameter specifies the location and size of the object to be appended. Note that most **Add*** procedures specify no dimensions for the object being added; in those cases, the dimensions are determined by **box**. Dimensions are specified in micas (roughly 35 micas/point).

```
GI.Box: TYPE = RECORD[   -- A GI.Box is similar to a Window.Box but a Window.box is
  place: GI.Place, dims: GI.Dims];        -- specified in points not micas

GI.Place: TYPE = RECORD[x, y: INTEGER];
GI.Dims: TYPE = RECORD[w, h: INTEGER];
```



This circle (ellipse) being added to a graphics frame has a box of [[x:100, y:200],[w:120, h:120]].

Note: it is illegal to specify negative width or height for a box. An object's **place** should always indicate its upper left corner.

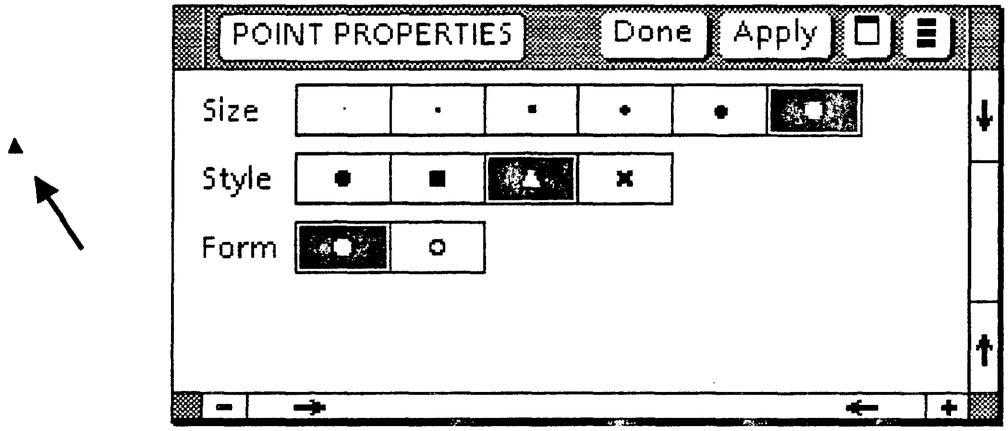
Adding a Point to a Graphics Frame

A point is defined by its `pointProps` and its `box` within the frame. The fields `w` and `h` are ignored in the `box` specification.

```
GI.AddPoint: PROC[
  h: GI.Handle, box: GI.Box, pointProps: GI.ReadonlyPointProps];
```

```
GI.PointProps: TYPE = LONG POINTER TO GI.PointPropsRec;
GI.PointPropsRec: TYPE = RECORD[
  wthbrush: CARDINAL,                 -- in micas (roughly 35 micas/point)
  pointStyle: GI.PointStyle ← round,-- shape of point
  pointFill: GI.PointFill ← solid,
  spare1: 0];
GI.nullPointProps: GI.PointPropsRec = [...];

GI.PointStyle: TYPE = MACHINE DEPENDENT {solid(0), hollow(1), (255)};
GI.PointFill: TYPE = MACHINE DEPENDENT {
  round(0), square(1), triangle(2), cross(3), (255)};
```

wthbrush specifies the size in micas. The legal set of values for **wthbrush** is {35, 71, 106, 141, 176, 212}. Any other values used may be harmful to the health of the document.



A

Adding a Line to a Graphics Frame

A line is defined by its `lineProps` and its `box` within the frame.

```
GI.AddLine: PROC[
  h: GI.Handle, box: GI.Box, lineProps: GI.ReadonlyLineProps];
```

```
GI.LineProps: TYPE = LONG POINTER TO GI.LinePropsRec;
LinePropsRec: TYPE = RECORD[
  brush: GI.Brush,
  lineEndNW: GI.LineEnd,     -- shape of line end
  lineEndSE: GI.LineEnd,
  lineEndHeadNW: GI.LineEndHead, -- shape of arrow head, if line end is an arrow
  lineEndHeadSE: GI.LineEndHead,
  direction: GI.LineDirection,
  fixedAngle: BOOL,
  spare1: LONG CARDINAL];
```

```
GI.LineEnd: TYPE = MACHINE DEPENDENT {
  flush(0), square(1), round(2), arrow(3), (7)};
GI.LineEndHead: TYPE = MACHINE DEPENDENT {none(0), h1(1), h2(20, h3(3), (15)};
GI.LineDirection: TYPE = MACHINE DEPENDENT {WE(0), NS(1), NwSe(2), SwNe(3)};
```

`lineEndNW` and `lineEndSE` describe the shape of the ends of a line. `lineEndNW` describes the end that is in the West, North, or NorthWest; West has precedence over North. `lineEndSE` describes the end that is in the East, South, or SouthEast, where East has precedence over South (See Slide 4-17). If `lineEnd* = arrow`, then `lineEndHead*` describes the type of arrow. For example, if `lineEndNW = arrow`, then `lineHeadNW = h1`, `h2`, or `h3`; otherwise, `lineEndHead*` is ignored.

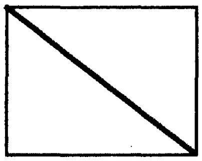
Adding a Line to a Graphics Frame (cont'd)

`direction` serves as either a constraint or a necessary parameter, depending on the type of line being added:

1. If you want to add a horizontal line, then the height of the box <u>must</u> be zero. If you want this line to have the constraint of always being horizontal, then set `direction = WE`; else, set it to **NwSe** or **SwNe** (doesn't matter which).

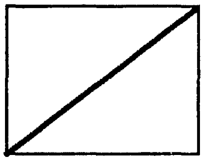
2. If you want to add a vertical line, then the width of the box <u>must</u> be zero. If you want this line to have the constraint of always being vertical, then set `direction = NS`; else, set it to **NwSe** or **SwNe** (doesn't matter which).

3. If you want a line other than horizontal or vertical, it will be a variation on one of the following:



Case A

or

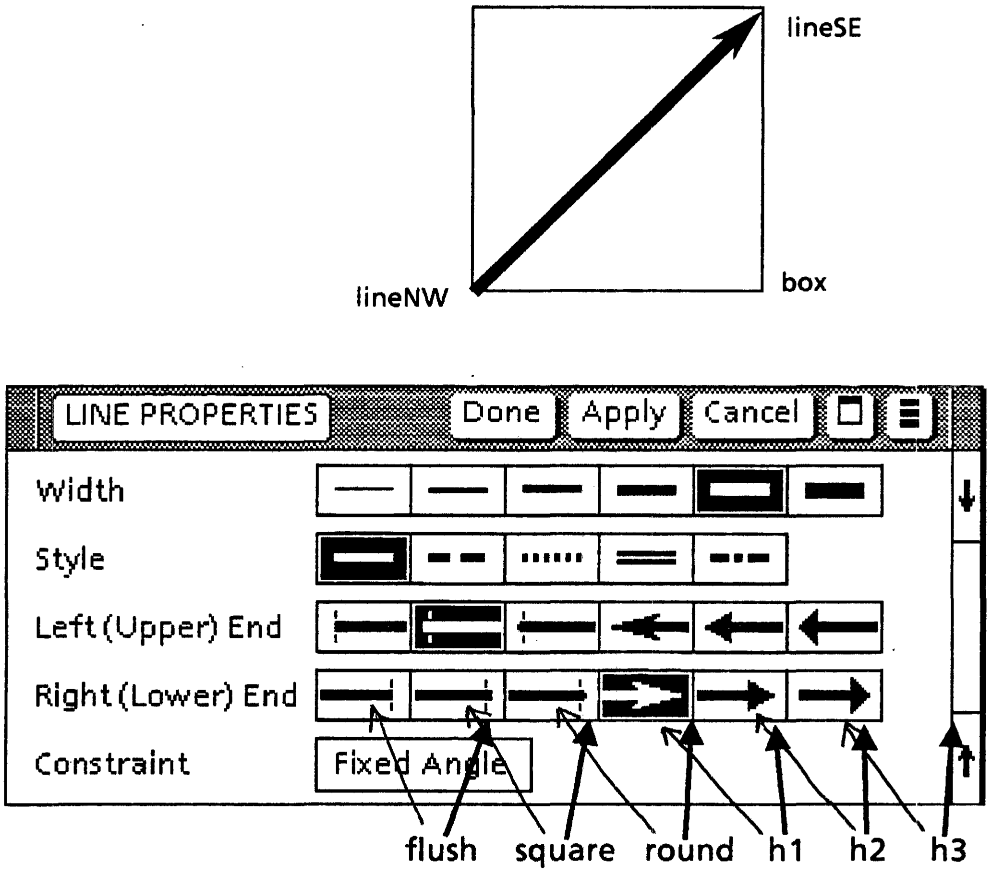


Case B

In Case A, set `direction = NwSe`. In Case B, set `direction = SwNe`.

Any other combination of `box` dimensions and `direction` will result in potentially hazardous situations.

Adding a Line to a Graphics Frame (cont'd)



A line where lineNW is in the SW corner of its box. direction is SwNe.
The property sheet for this line is also shown.

For the above...

```
myLinePropsRec: GI.LinePropsRec ← [
  brush: [176, solid], lineEndNW: square, lineEndSE: arrow,
  lineEndHeadNW: none, lineEndHeadSE: h1, direction: SwNe,
  fixedAngle: FALSE, spare1: 0];
```

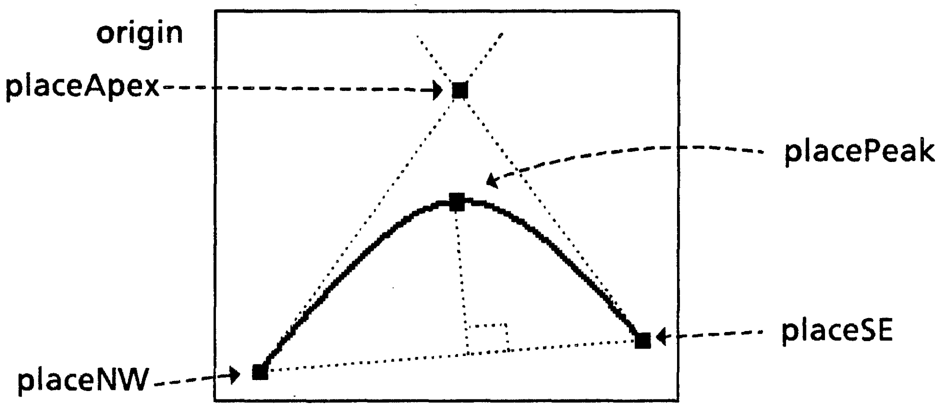
Adding a Curve to a Graphics Frame

A curve is defined by its `curveProps` and its `box` within the frame.

```
GI.AddCurve: PROC[
  h: GI.Handle, box: GI.Box, curveProps: GI.ReadonlyCurveProps];
```

```
GI.CurveProps: TYPE = LONG POINTER TO GI.CurvePropsRec;
GI.CurvePropsRec: TYPE = RECORD[
  brush: GI.Brush,
  lineEndNW: GI.LineEnd,
  lineEndSE: GI.LineEnd,
  lineEndHeadNW: GI.LineEndHead,
  lineEndHeadSE: GI.LineEndHead,
  direction: GI.LineDirection,          -- direction ignored, set to WE
  placeNW, placeApex, placeSE, placePeak: GI.Place,    -- RECORD [x, y: INTEGER]
  fixedAngle: BOOL,
  spare1: LONG CARDINAL];
```

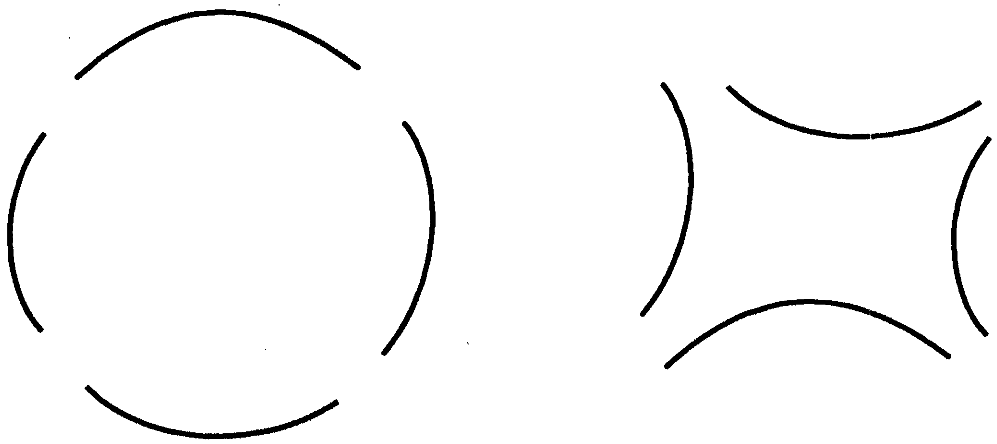
`place*` defines the curve by specifying the endpoints, apex, and peak. `place*` is relative to the origin of curve's `box`.



The peak is the highest point on the curve; the apex is the intersection of the two tangents drawn from placeNW and placeSE.

Adding a Curve to a Graphics Frame (cont'd)

Curves paint clockwise, where the NW point of a curve must be painted before the SE point. Can you identify the NW and SE points on the following curves?



Adding an Eccentric Curve to a Graphics Frame

An eccentric curve is defined by its curveProps and its box within the frame. The curve is specified by its endpoints, apex, and eccentricity. eccentricity is a fraction whose value is treated as:

```
eccentricity / LAST[CARDINAL]
```

thereby allowing the highest precision for eccentricities from 0 - 1.

```
GI.AddEccentricCurve: PROC[
  h: GI.Handle,
  box: GI.Box,
  eccentricCurveProps: GI.ReadonlyEccentricCurveProps];
```

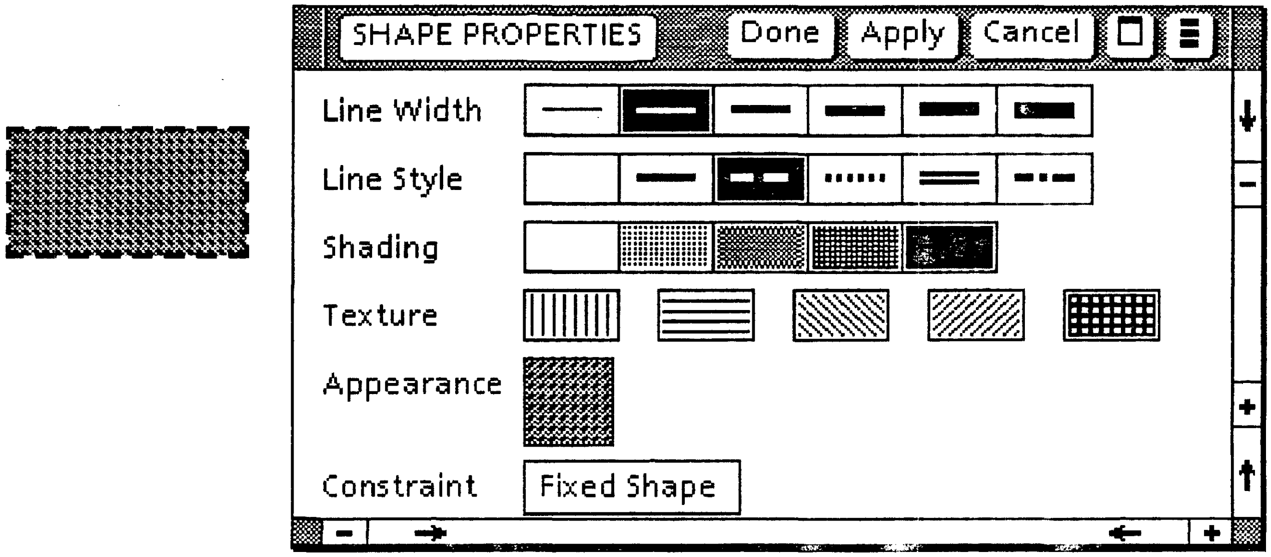
```
GI.EccentricCurveProps: TYPE = LONG POINTER TO GI.EccentricCurvePropsRec;
GI.EccentricCurvePropsRec: TYPE = RECORD[
  brush: GI.Brush,
  lineEndNW: GI.LineEnd,
  lineEndSE: GI.LineEnd,
  lineEndHeadNW: GI.LineEndHead,
  lineEndHeadSE: GI.LineEndHead,
  direction: GI.LineDirection,
  placeNW, placeApex, placeSE: GI.Place,  -- relative to origin of curve's box
  eccentricity: CARDINAL,
  fixedAngle: BOOL,
  spare1: LONG CARDINAL];
```

Adding a Rectangle to a Graphics Frame

A rectangle is defined by its `rectangleProps` and its `box` within the frame. Squares are a subset of rectangles. The dimensions of the rectangle correspond to the `box` size.

```
GI.AddRectangle: PROC[h: GI.Handle,
  box: GI.Box, rectangleProps: GI.ReadonlyRectangleProps];
```

```
GI.RectangleProps: TYPE = LONG POINTER TO GI.RectanglePropsRec;
GI.RectanglePropsRec: TYPE = RECORD[
  brush: GI.Brush,
  shading: GI.Shading,
  fixedShape: BOOL,
  spare1: LONG CARDINAL];
```



A rectangle and its properties. These properties are the same for any polygon.

Shape Properties

All polygons have similar properties that include a `brush` and `shading`.

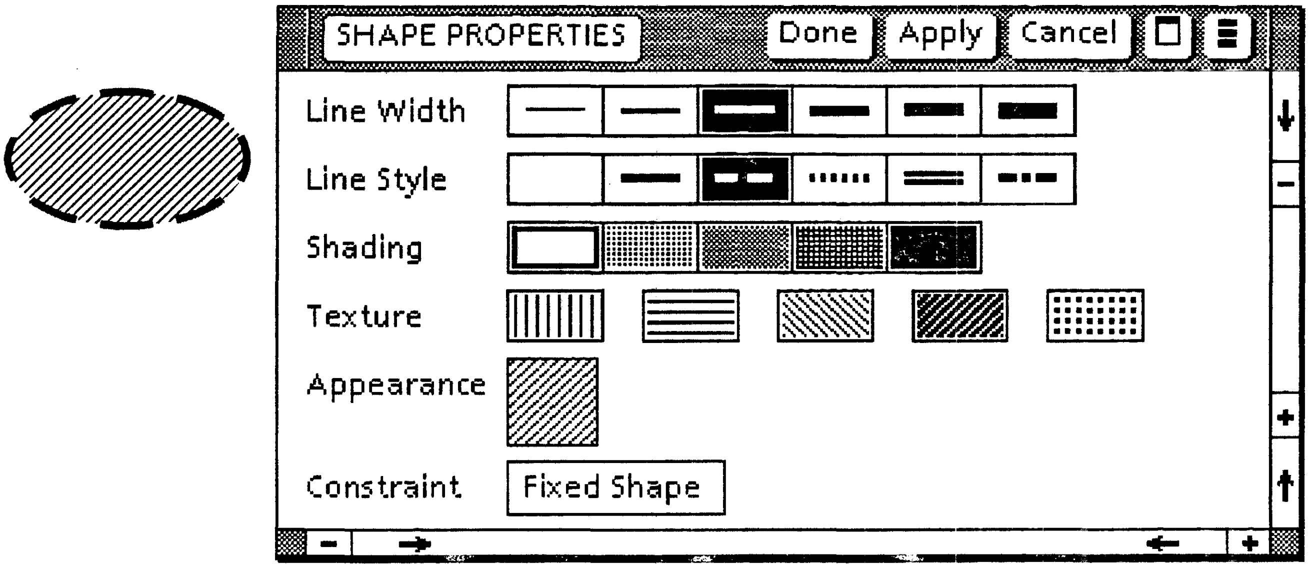
```
GI.Brush: TYPE = RECORD[
  wthBrush: CARDINAL,          -- in micas
  styleBrush: GI.StyleBrush]; -- invisible, solid, dashed, dotted, double, broken

GI.Shading: TYPE = RECORD[gray: GI.Gray, textures: GI.Textures];

GI.Gray: TYPE = MACHINE DEPENDENT {
  none(0), gray25(1), gray50(2), gray75(3), black(4), (15)};

GI.Textures: TYPE = PACKED ARRAY GI.Texture OF BOOLEAN;

GI.Texture: TYPE = MACHINE DEPENDENT {
  vertical(0), horizontal(1), nwse(2), swne(3), polkadot(4), (11)};
```

Adding an Ellipse to a Graphics Frame

An elipse is defined by its `ellipseProps` and its `box` within the frame. Circles are a subset of ellipses.

```
GI.AddEllipse: PROC[
  h: GI.Handle, box: GI.Box,
  ellipseProps: GI.ReadonlyEllipseProps];
```

```
GI.EllipseProps: TYPE = LONG POINTER TO GI.EllipsePropsRec;
GI.EllipsePropsRec: TYPE = RECORD[
  brush: GI.Brush,
  shading: GI.Shading,
  fixedShape: BOOL,
  spare1: LONG CARDINAL];
```



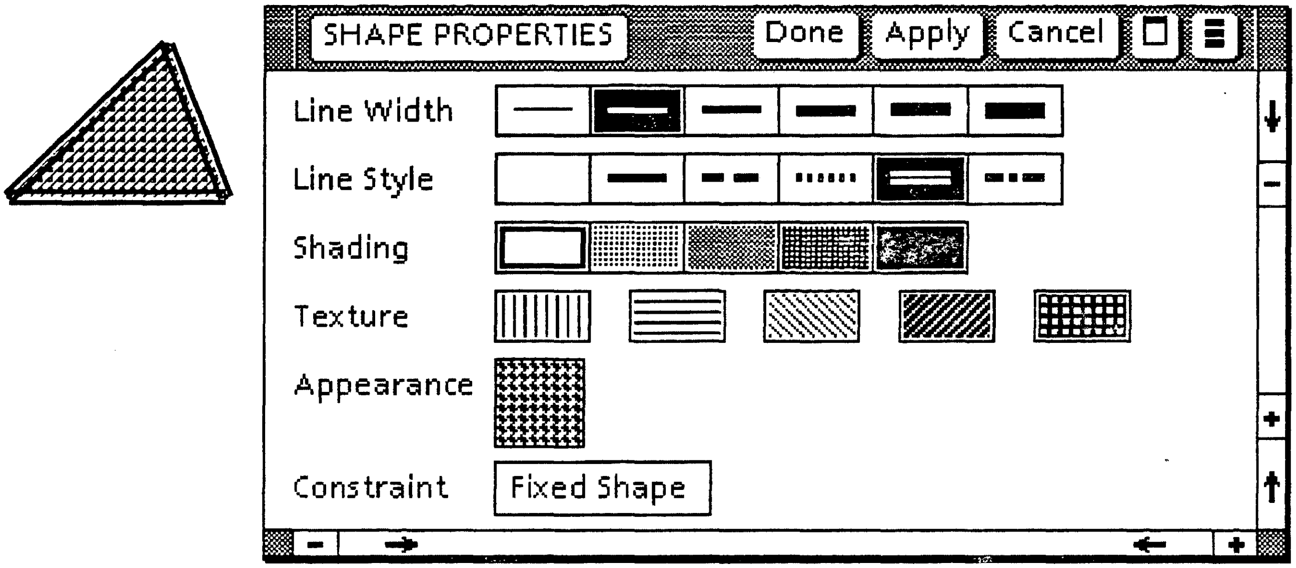
An ellipse and its properties. Double border line style is not supported for ellipses.

Adding a Triangle to a Graphics Frame

A triangle is defined by its `triangleProps` and its `box` within the frame. The triangle's dimensions are determined by their `place*`, although the amount displayed is determined by `box`.

```
GI.AddTriangle: PROC[
  h: GI.Handle, box: GI.Box,
  triangleProps: GI.ReadonlyTriangleProps];
```

```
GI.TriangleProps: TYPE = LONG POINTER TO GI.TrianglePropsRec;
GI.TrianglePropsRec: TYPE = RECORD[
  brush: GI.Brush,
  shading: GI.Shading,
  place1, place2, place3: GI.Place, -- corners of triangle relative to upper left of triangle's box
  fixedShape: BOOL,
  spare1: LONG CARDINAL];
```



Line Width

Line Style

Shading

Texture

Appearance

Constraint Fixed Shape

A triangle and its properties.

Adding a Text Frame to a Graphics Frame

A text frame is defined by its `frameProps` and its `box` within the frame.

```
GI.AddTextFrame: PROC[
  h: GI.Handle, box: GI.Box,
  frameProps: GI.ReadonlyFrameProps,
  textFrameProps: GI.ReadonlyTextFrameProps,
  wantTextHandle, wantTopCaptionHandle, wantBottomCaptionHandle,
  wantLeftCaptionHandle, wantRightCaptionHandle: BOOLEAN ← FALSE]
  RETURNS[
    text: TextInterchangeDefs.Text, topCaption,
    bottomCaption, leftCaption, rightCaption: DI.Caption];
```

```
GI.FrameProps: TYPE = LONG POINTER TO GI.FramePropsRecord;
GI.FramePropsRec: TYPE = RECORD[
  brush: GI.Brush,
  fixedShape: BOOL,
  margins: ARRAY GI.Side OF CARDINAL,     -- micas
  captionContent:ARRAY Side OF DID.Caption,
  spare1: LONG CARDINAL];
```

```
GI.TextFrameProps: TYPE = LONG POINTER TO GI.TextFramePropsRec;
GI.TextFramePropsRec: TYPE = RECORD[
  expandRight, expandBottom, transparent: BOOL,
  tFrameProps: TextInterchangeDefs.TFramePropsRec,
  spare1: LONG CARDINAL];
```

Specify `want*Handle` for those objects to which you want to add text.

Adding a Form Field to a Graphics Frame

A form field is defined by its fieldProps and its box within the frame.

```
GI.AddFormField: PROC[
  h: GI.Handle, box: GI.Box, fieldProps: DIP.ReadonlyFieldProps,
  frameProps: GI.ReadonlyFrameProps,
  paraProps: DIP.ReadonlyParaProps ← NIL,
  fontProps: DIP.ReadonlyFontProps ← NIL,
  expandRight, expandBottom: BOOL ← FALSE,
  wantFieldHandle, wantTopCaptionHandle, wantBottomCaptionHandle,
  wantLeftCaptionHandle, wantRightCaptionHandle: BOOLEAN ← FALSE]
  RETURNS[
    field: DI.Field, topCaption,
    bottomCaption, leftCaption, rightCaption: DI.Caption];
```

```
DIP.FieldProps: TYPE = LONG POINTER TO DIP.FieldPropsRecord;
DIP.FieldPropsRecord: TYPE = RECORD[
  language: MultiNational.Language,
  length: CARDINAL,
  required: BOOL,
  skipIf: DIP.SkipIfChoiceType,
  stopOnSkip: BOOL,
  type: DIP.FieldChoiceType,
  ...];
```

As in AddTextFrame, you specify want*Handle for those objects to which you want to add text.

Adding a Bitmap to a Graphics Frame

A bitmap, as defined by its `bitmapProps`, can be added to a graphics frame.

```
GI.AddBitmap: PROC[
  h: GI.Handle, box: GI.Box,
  bitmapProps: GI.ReadonlyBitmapProps,
  frameProps: GI.ReadonlyFrameProps,
  wantTopCaptionHandle, wantBottomCaptionHandle,
  wantLeftCaptionHandle, wantRightCaptionHandle: BOOLEAN ← FALSE]
  RETURNS[
    topCaption, bottomCaption,
    leftCaption, rightCaption: DI.Caption];
```

```
GI.BitmapProps: TYPE = LONG POINTER TO GI.BitmapPropsRec;
BitmapPropsRec: TYPE = RECORD[
  opaque: BOOLEAN,
  xOffset, yOffset: LONG INTEGER,
  printFile: XString.ReaderBody, -- name of print file if desired
  displaySource: GI.BmDisplay,
  scalingProps: GI.BitmapScalingProps,
  spare1: LONG CARDINAL];
```

Bitmap Display and Scaling Props

```
GI.BmDisplay: TYPE = RECORD [  -- bitmap source is either internal
 SELECT type: * FROM           -- (the bits are copied into the document) or a desktop file.
   internal => [bm: LONG POINTER TO GI.BitmapData],
   file => [name: XString.ReaderBody],
   ENDCASE];

GI.BitmapData: TYPE = RECORD[
 signature: INTEGER ← GI.bmSignature, -- do not use any other value
 xScale: Interpress.Rational,
 yScale: Interpress.Rational,
 xDim: CARDINAL, yDim: CARDINAL,     -- # of bits wide and tall
 bpl: CARDINAL,                      -- Bits Per Line = ((xDim + 15) / 16) * 16
 pages: NSSegment.PageCount,
 bits: PACKED ARRAY [0..0) OF Environment.Byte];

GI.BitmapScalingProps: TYPE = RECORD[
 SELECT type: * FROM
   printerResolution => [resolution: CARDINAL],
   fixed => [
     horizontalAlignment: {center, right, left},
     verticalAlignment: {center, bottom, top},
     scalingPercentage: CARDINAL [0..1024)],
     automatic => [shape: {similar, fillUp}],
   other => [
     spare1: PACKED ARRAY [2..15) OF [0..1], spare2: CARDINAL],
   ENDCASE];
```

Adding an Image Frame to a Graphics Frame

An image frame is defined by its imageProps and its box within the frame. An image frame deals with Xerox' 9700 product and is beyond the scope of this class.

```
GI.AddImage: PROC[
  h: GI.Handle, box: GI.Box,
  imageProps: GI.ReadonlyImageProps,
  frameProps: GI.ReadonlyFrameProps,
  wantTopCaptionHandle, wantBottomCaptionHandle,
  wantLeftCaptionHandle, wantRightCaptionHandle: BOOLEAN ← FALSE]
  RETURNS[
    topCaption, bottomCaption,
    leftCaption, rightCaption: DI.Caption];
```

Adding Other Objects to a Graphics Frame

Other objects are defined by their `instance` and `box` within the frame. One example of usage would be adding VP Charts to a graphics frame.

```
GI.AddOther: PROC[
  h: GI.Handle, box: GI.Box, instance: DI.Instance];
```

Constants and Defaults

You can get "neutral" property values by specifying one of the following constants:

```
GI.nullBitmapProps: GI.BitmapPropsRec = [...];
GI.nullBmDisplay: GI.BmDisplay = [...];
GI.nullBitmapScalingProps: GI.BitmapScalingProps = [...];
GI.nullButtonProps: GI.ButtonPropsRec = [...];
GI.nullCurveProps: GI.CurvePropsRec = [...];
GI.nullEccentricCurveProps: GI.EccentricCurvePropsRec = [...];
GI.nullEllipseProps: GI.EllipsePropsRec = [...];
GI.nullFrameProps: GI.FramePropsRec = [...];
GI.nullImageProps: GI.ImagePropsRec = [...];
GI.nullLineProps: GI.LinePropsRec = [...];
GI.nullRectangleProps: GI.RectanglePropsRec = [...];
GI.nullTextFrameProps: GI.TextFramePropsRec = [...];
GI.nullTriangleProps: GI.TrianglePropsRec = [...];
```

. Adding Text to a Graphics Frame

Text can be added to form fields, text frames, and captions of all frames (i.e. bitmap, image, text, button, and nested) and form fields. Note the return parameters for the following procedures:

```
GI.StartGraphicsFrame: PROC[...]
  RETURNS[
    gfh: GI.Handle,
    topCaption, bottomCaption,
    leftCaption, rightCaption: DI.Caption];

GI.AddFormField: PROC[...]
  RETURNS[
    field: DI.Field,
    topCaption, bottomCaption,
    leftCaption, rightCaption: DI.Caption];

GI.AddTextFrame: PROC[...]
  RETURNS[
    text:TextInterchangeDefs.Text,
    topCaption, bottomCaption, leftCaption,
    rightCaption: DI.Caption];
```

Adding Text to a Graphics Frame (cont'd)

Once you have a DocInterchangeDefs.Caption or DI.Field, you can use that Caption or Field as a DocInterchangeDefs.TextContainer and thus use the DocInterchangeDefs.Append* procedures to add various objects to it.

```
DI.TextContainer: TYPE = RECORD[
  var: SELECT type: * FROM
    caption => [h: DI.Caption],
    doc => [h: DI.Doc],
    field => [h: DI.Field],
    heading => [h: DI.Heading],
    footing => [h: DI.Footing],
    ...
    ENDCASE];
```

When you have finished appending to the caption or field, call DocInterchangeDefs.ReleaseCaption or DI.ReleaseField to release any resources associated with the non-NIL handle.

```
DI.ReleaseCaption: PROC[captionPtr: LONG POINTER TO DI.Caption];

DI.ReleaseField: PROC[fieldPtr: LONG POINTER TO DI.Field];
```

Adding Text to a Graphics Frame (cont'd)

Once you have a `TextInterchangeDefs.Text` (TID), you can add objects to that `Text` with calls to `TextInterchangeDefs.Append*ToText`:

```
TID.AppendCharToText: PROC[
  to: TID.Text,
  char: XChar.Character,
  fontProps:DIP.ReadonlyFontProps ← NIL,
  nToAppend: CARDINAL ← 1];

TID.AppendFieldToText: PROC[
  to: TID.Text,
  fieldProps: DIP.ReadonlyFieldProps,
   fontProps: DIP.ReadonlyFontProps ← NIL]
  RETURNS[field: DI.Field];

TID.AppendNewParagraphToText: PROC[
  to: TID.Text,
  paraProps: DIP.ReadonlyParaProps ← NIL,
  fontProps: DIP.ReadonlyFontProps ← NIL,
  nToAppend: CARDINAL ← 1];

TID.AppendTextToText: PROC[
  to: TID.Text,
  text: XString.Reader,
  textEndContext: XString.Context,
  fontProps: DIP.ReadonlyFontProps ← NIL];
```

Releasing Text

When you've finished appending to `Text`, call `TextInterchangeDefs.ReleaseText` to release any resources associated with the handle.

```
TID.ReleaseText: PROC[textPtr: LONG POINTER TO TID.Text];
```

Adding Text to a Button Program

You can add text to the button program returned by the call to GraphicsInterchangeDefs.StartButton.

```
GI.AppendCharToButtonProgram: PROC[
  to: GI.ButtonProgram,
  char: XChar.Character,
  fontProps: DIP.ReadonlyFontProps ← NIL,
  nToAppend: CARDINAL ← 1];

GI.AppendNewParagraphToButtonProgram: PROC[
  to: GI.ButtonProgram,
  paraProps: DIP.ReadonlyParaProps ← NIL,
  fontProps: DIP.ReadonlyFontProps ← NIL,
  nToAppend: CARDINAL ← 1];

GI.AppendTextToButtonProgram: PROC[
  to: GI.ButtonProgram,
  text: XString.Reader,
  textEndContext: XString.Context,
  fontProps: DIP.ReadonlyFontProps ← NIL];

GI.ButtonProgram: TYPE = LONG POINTER TO GI.ButtonProgramObject;
GI.ButtonProgramObject: TYPE;
```

Enumerating a Graphics Frame

When enumerating a document with DocInterchangeDefs.Enumerate, you may want to look at the objects that are in the graphic frames of the document. You can enumerate those objects with a call to GraphicsInterchangeDefs.Enumerate.

```
GI.Enumerate: PROC[
  doc: DI.Doc,
  graphicsContainer: DI.Instance,
  procs: GI.EnumProcs,
  clientData: LONG POINTER ← NIL]
  RETURNS[dataSkipped: BOOLEAN];
```

The return parameter dataSkipped is not currently implemented and is always set to FALSE.

Enumerating a Graphics Frame Example

Typically, you would call **GI.Enumerate** from the **AnchoredFrameProc** that was passed to **DID.Enumerate**.

```
AnchoredFrameProc: DI.AnchoredFrameProc = {
<< clientData: LONG POINTER, type: DI.AnchoredFrameType, ...,
   anchoredFrame: DI.Instance,...>>
  enumProcs: GI.EnumProcsRecord ← [    -- Some are defaulted to NIL
    bitmapProc: MyBitmapProc,
    clusterProc: MyClusterProc,
    curveProc: MyCurveProc,
    ellipseProc: MyEllipseProc,
    lineProc: MyLineProc,
    pointProc: MyPointProc,
    rectangleProc: MyRectangleProc,
    triangleProc: MyTriangleProc];
  myDoc: DID.Doc ← LOOPHOLE[clientData];
  ...
  IF type = graphics THEN {
    [ ] ← GI.Enumerate[
      doc: myDoc,
      graphicsContainer: anchoredFrame,
      procs: @enumProcs];
  ...
  };
 ...
 };
```

EnumProcs

GraphicsInterchangeDefs.EnumProcs are passed when enumeration is performed. A separate EnumProc is provided for each type of object that can appear in a graphics frame. If a procedure is defaulted to NIL, then nothing will happen when an object of the type covered by that procedure is encountered. The Enumerator always releases containers after calling each EnumProc.

```
GI.EnumProcs: TYPE = LONG POINTER TO GI.EnumProcsRecord;
GI.EnumProcsRecord: TYPE = RECORD[
  bitmapProc: GI.BitmapProc ← NIL,
  buttonProc: GI.ButtonProc ← NIL,
  clusterProc: GI.ClusterProc ← NIL,
  curveProc: GI.CurveProc ← NIL,
  ellipseProc: GI.EllipseProc ← NIL,
  formFieldProc: GI.FormFieldProc ← NIL,
  frameProc: GI.FrameProc ← NIL,
  imageProc: GI.ImageProc ← NIL,
  lineProc: GI.LineProc ← NIL,
  otherProc: GI.OtherProc ← NIL,
  pointProc: GI.PointProc ← NIL,
  rectangleProc: GI.RectangleProc ← NIL,
  textFrameProc: GI.TextFrameProc ← NIL,
  triangleProc: GI.TriangleProc ← NIL];
```

LineProc, CurveProc, and PointProc

These EnumProcs are similar in that they are all geometric objects. For each geometric object, a set of properties describing that object is passed into the appropriate procedure. box describes the position and size of the object within the frame.

```
GI.LineProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  lineProps: GI.ReadonlyLineProps]
  RETURNS[stop: BOOLEAN ← FALSE];

GI.CurveProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  curveProps: GI.ReadonlyCurveProps]
  RETURNS[stop: BOOLEAN ← FALSE];

GI.PointProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  pointProps: GI.ReadonlyPointProps]
  RETURNS[stop: BOOLEAN ← FALSE];
```

EllipseProc, RectangleProc, and TriangleProc

Each of these procedures represents a shape that can be shaded.

```
GI.EllipseProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  ellipseProps: GI.ReadonlyEllipseProps]
  RETURNS[stop: BOOLEAN ← FALSE];

GI.RectangleProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  rectangleProps: GI.ReadonlyRectangleProps]
  RETURNS[stop: BOOLEAN ← FALSE];

GI.TriangleProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  triangleProps: GI.ReadonlyTriangleProps]
  RETURNS[stop: BOOLEAN ← FALSE];
```

ButtonProc, ClusterProc, FrameProc

These EnumProcs are similar in that they all deal with containers within a graphics frame.

```
GI.ButtonProc: TYPE = PROC[
  clientData: LONG POINTER,
  graphicsContainer: DI.Instance,
  box: GI.Box,
  buttonProps: GI.ReadonlyButtonProps,
  frameProps: GI.ReadonlyFrameProps,
  buttonProgram: GI.ButtonProgram]
  RETURNS[stop: BOOLEAN ← FALSE];

GI.ClusterProc: TYPE = PROC[
  clientData: LONG POINTER,
  graphicsContainer: DI.Instance,
  box: GI.Box]
  RETURNS[stop: BOOLEAN ← FALSE];

GI.FrameProc: TYPE = PROC[
  clientData: LONG POINTER,
  graphicsContainer: DI.Instance,
  box: GI.Box,
  frameProps: GI. ReadonlyFrameProps,
  name, description: XString.Reader,
  spareProps: LONG POINTER]
  RETURNS[stop: BOOLEAN ← FALSE];
```

BitmapProc, FormFieldProc, and TextFrameProc

These EnumProcs comprise the remaining procedures.

```
GI.BitmapProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  bitmapProps: GI.ReadonlyBitmapProps,
  frameProps: GI.ReadonlyFrameProps]
  RETURNS[stop: BOOLEAN ← FALSE];

GI.FormFieldProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  fieldProps: DIP.ReadonlyFieldProps,
  frameProps: GI.ReadonlyFrameProps,
  paraProps: DIP.ReadonlyParaProps,
  fontProps: DIP.ReadonlyFontProps,
  expandRight, expandBottom: BOOL, content: DI.Field]
  RETURNS[stop: BOOLEAN ← FALSE];

GI.TextFrameProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  frameProps: GI.ReadonlyFrameProps,
  textFrameProps: GI.ReadonlyTextFrameProps,
  content: TextInterchangeDefs.Text]
  RETURNS[stop: BOOLEAN ← FALSE];
```

ImageProcs and OtherProcs

```
GI.ImageProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  imageProps: GI.ReadonlyImageProps,
  frameProps: GI.ReadonlyFrameProps]
  RETURNS[stop: BOOLEAN ← FALSE];

GI.OtherProc: TYPE = PROC[
  clientData: LONG POINTER,
  box: GI.Box,
  instance: DI.Instance,
  objectType: GI.OtherObjectType]
  RETURNS[stop: BOOLEAN ← FALSE];

GI.OtherObjectType: TYPE = MACHINE DEPENDENT {
  illusFrame(0),  -- not implemented
  barchart, linechart, piechart, pieslice, table, equation,
  firstAvailable, lastAvailable(255)};
```

Enumerating Text in Graphics Frames

`TextInterchangeDefs` (TID) allows you to enumerate text in a graphics frame. You supply three call-back procedures to handle fields, new paragraphs, and text.

```
TID.EnumerateText: PROC[
  text: TID.Text,
  procs: TID.TextEnumProcs,
  clientData: LONG POINTER ← NIL]
  RETURNS[dataSkipped: BOOLEAN];

TID.TextEnumProcs: TYPE = LONG POINTER TO TID.TextEnumProcsRecord;

TID.TextEnumProcsRecord: TYPE = RECORD[
  fieldProc: DI.FieldProc ← NIL,
  newParagraphProc: DI.NewParagraphProc ← NIL,
  textProc: DI.TextProc ← NIL,
  tileProc: DI.TileProc ← NIL];
```

Enumerating a Button Program

You can extract the contents of a Button program obtained by calling GraphicsInterchangeDefs.EnumerateButtonProgram.

```
GI.EnumerateButtonProgram: PROC[
  buttonProgram: GI.ButtonProgram,
  procs: GI.ButtonProgramEnumProcs,
  clientData: LONG POINTER ← NIL]
  RETURNS[dataSkipped: BOOLEAN];

GI.ButtonProgramEnumProcs: TYPE =
  LONG POINTER TO GI.ButtonProgramEnumProcsRecord;

GI.ButtonProgramEnumProcsRecord: TYPE = RECORD[
  newParagraphProc: DI.NewParagraphProc ← NIL,
  textProc: DI.TextProc ← NIL];
```

Retrieving Additional Anchored Frame Properties

There are additional properties that you can store and retrieve from an anchored frame.

```
GI.ExtraAnchoredFramePropsElements: TYPE = {
  name, description, spareProps};

GI.ExtraAnchoredFramePropsSelections: TYPE = PACKED ARRAY
  GI.ExtraAnchoredFramePropsElements OF
  DIP.BooleanFalseDefault;

GI.GetExtraAnchoredFrameProps: PROC[
  doc: DI.Doc,
  anchoredFrame: DI.Instance,
  spareProps: LONG POINTER ← NIL,
  zone: UNCOUNTED ZONE],-- zone is for future use; return values are read-only
  RETURNS[name, description: XString.ReaderBody];

GI.SetExtraAnchoredFrameProps: PROC[
  doc: DI.Doc,
  anchoredFrame: DI.Instance,
  name, description: XString.Reader,
  spareProps: LONG POINTER ← NIL,
  selections: GI.ExtraAnchoredFramePropsSelections];
```

The Big Example

Here's how you would use **GraphicsInterchangeDefs** to copy graphics frames from one document to another. Basically you enumerate the objects in a graphics frame in the original document and append corresponding objects to a graphics frame in the new document. This example recognizes only graphics frames in the original document; everything else is ignored. Document and graphics handles are declared as global variables for simplicity.

```
OPEN DI:DocInterchangeDefs, GI:GraphicsInterchangeDefs;
originalDoc, newDoc: DI.Doc;
graphicsHandle: GI.Handle;

CopyGraphicsFrames: PROC[someRef: NSFile.Reference] = {
  enumProcs: DI.EnumProcsRecord ← [anchoredFrameProc: GraphicProc];
  newFile: NSFile.Handle ← NSFile.nullHandle;
  desktop: NSFile.Handle ← NSFile.OpenByReference[
    StarDesktop.GetCurrentDesktopFile[]];
  originalDoc ← DI.Open[docFileRef: someRef].doc;      -- should check for errors
  newDoc ← DI.StartCreation[].doc;
  [] ← DI.Enumerate[textContainer: [doc[h: originalDoc]], procs: @enumProcs];
  DI.Close[docPtr: @originalDoc];
  [newFile,] ← DI.FinishCreation[@newDoc];   -- should check for errors

  -- place new file on desktop
  NSFile.Move[file: newFile, destination: desktop];
  StarDesktop.AddReferenceToDesktop[reference: NSFile.GetReference[newFile]];
  NSFile.Close[newFile];
  };

GraphicProc: DI.AnchoredFrameProc = {
  graphicsEnumProcs: GI.EnumProcsRecord ← [
    clusterProc: MakeCluster,
    curveProc: MakeCurve,
    ellipseProc: MakeEllipse,...];
  graphicsInstance:DI.Instance;
  IF type # graphics THEN RETURN;
  graphicsHandle ← GI.StartGraphics[doc:newDoc];
  [] ← GI.Enumerate[doc: originalDoc, graphicsContainer: anchoredFrame,
    procs: @graphicsEnumProcs];
  graphicsInstance ← GI.FinishGraphics[graphicsHandle];
  [] ← DI.AppendAnchoredFrame[to: newDoc, type: graphics,
    anchoredFrameProps: anchoredFrameProps, content: graphicsInstance];
  };
```

```
MakeCluster: GI.ClusterProc = {
  graphicsEnumProcs: GI.EnumProcsRecord ← [
    clusterProc: MakeCluster,
    curveProc: MakeCurve,
    ellipseProc: MakeEllipse,...];
  [] ← GI.Enumerate[doc: originalDoc, graphicsContainer: graphicsContainer,
    procs: @graphicsEnumProcs];
  };

MakeCurve: GI.CurveProc = {
  GI.AddCurve[h: graphicsHandle, box: box, curveProps: curveProps];
  };

MakeEllipse: GI.EllipseProc = {
  GI.AddEllipse[h: graphicsHandle, box: box, ellipseProps: ellipseProps];
  };
```

Day 5: Performance Issues

Outline

I. Overview - The Mesa Machine and Pilot

II. Code

- A. Localizing Code
    - 1. Reducing Call-outs
    - 2. INLINEs
- B. Minimizing Code Bytes
    - 1. Links
    - 2. Tools
- C. Avoiding Recomputation

III. Data

- A. Localizing Data
- B. Minimizing Data
- C. Heaps

IV. Cost of NSFile Operations

V. Examples

Overview

This section will help you produce code that will run faster and use less memory. We will discuss the Mesa machine and Pilot, and how you can use this knowledge to write more efficient code.

- Page swapping is a major cause of poor program performance. If you reduce the number of swaps required for common operations, the performance benefits can be great.
- Generally, most program computation occurs in small "hot" portions of code. We will discuss how to locate hot code and make it run as fast as possible.
- The MDS is a scarce resource! By eliminating global variables, you can conserve the MDS and produce reentrant code.

Keep in mind that the following suggestions all have trade offs (i.e. you might increase the speed of a program at the expense of code size).

Swapping

Pilot swaps collections of pages called *swap units*; when anything contained inside a swap unit is needed in real memory, Pilot swaps in all of the pages of the swap unit. When Pilot needs to reclaim real memory, it swaps out individual pages.

The Binder packages each module in a config into a single swap unit. Thus, Pilot swaps in an entire module even if only one procedure was accessed.

Because of this behavior, you should limit intermodule references whenever possible. Shared code should be placed in a single "hot" module so that it will remain in real memory independent of the user's actions. Code specific to a user action can be placed in a separate module that is only swapped in when that action is invoked and swapped out when other actions occur.

Hot Code

Many programs spend much of their processing time in a relatively small number of procedures. By locating these procedures and making them more efficient, you can often dramatically speed up your code.

There are several performance tools that can be used to monitor the frequency in which procedures in a particular program are called. Using this information, we can isolate the hot code and spend more time increasing its speed. Similarly, the Lister utility allows you to list the machine code for procedures; thus, the code sizes for various algorithms can be compared.

Localizing Code

A VP application may have several menu commands that are implemented by several different modules. If these commands are contained within the corresponding module (plug-ins), then when the application is opened, the system must take at least two page faults for each module. One page faults occurs for the global frame of each separate module and another for the actual code.

```
--FooImpl.mesa
```

```
MenuData.CreateItem[...proc: FooDefs.Command1...];
MenuData.CreateItem[...proc: FooDefs.Command2...];
...
```

```
--FooImplA.mesa
```

```
Command1: PUBLIC MenuData.MenuProc = {...};
```

```
--FooImplB.mesa
```

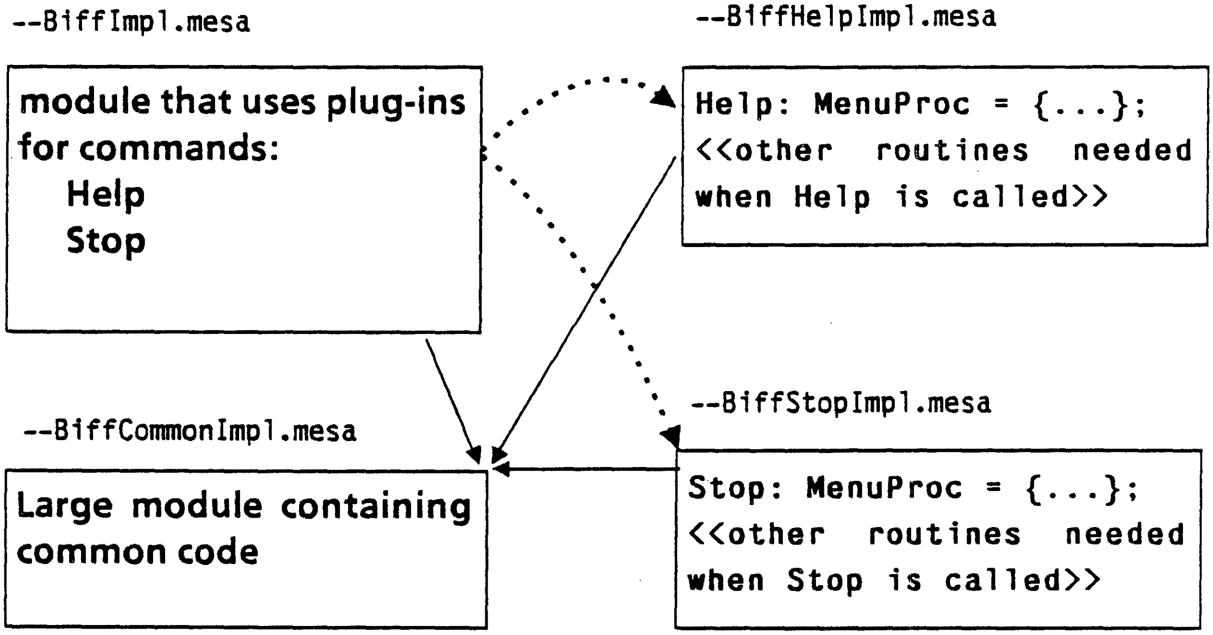
```
Command2: PUBLIC MenuData.MenuProc = {...};
```

...

Localizing Code (cont'd)

When you must use plug-in procedure values, you should place procedures with different functions in separate modules. In this fashion, you can reduce the amount of code that must be swapped in for a particular event.

The idea here is to place code used for independent functions in relatively small modules to minimize the effect of swapping. Shared code and hot code should be placed in large modules that will, hopefully, remain resident in memory independent of the user's actions.



--BiffImpl.mesa

module that uses plug-ins for commands:
Help
Stop

--BiffHelpImpl.mesa

```
Help: MenuProc = {...};
<<other routines needed
when Help is called>>
```

--BiffCommonImpl.mesa

Large module containing common code

--BiffStopImpl.mesa

```
Stop: MenuProc = {...};
<<other routines needed
when Stop is called>>
```

Direct Procedure Calls

You should call directly to the source implementation, instead of calling other modules that eventually make the same call. This practice often occurs when you attempt to avoid modifying the client code after an implementation has been replaced.

In the example below, FooClientImpl should call `FooPrivDefs.RealGetNum` directly.

```
-- FooClientImpl.mesa
num: CARDINAL ← FooDefs.GetNum[...];

-- FooImpl.mesa
-- gutted implementation (executable code is now in another procedure)
GetNum: PUBLIC PROC[...] RETURNS[CARDINAL] = {
  RETURN[FooPrivDefs.RealGetNum[...]];
  };

--FooPrivImpl.mesa
RealGetNum: PUBLIC PROC[...] RETURNS[num: CARDINAL] = {
  <<get the number>>
  RETURN[num];
  };
```

Inline Procedures

Using inline procedures correctly can potentially save two swaps per call; however, indiscriminate use of inlines can degrade compiler performance, cause tables to overflow, and generate vast amounts of code. Use inlines when you have:

- a procedure that is defined in a PROGRAM module and called only once in that same module. This allows you to use procedures for program structure without the overhead of the procedure call.

- a procedure that is small (no more than 6 byte codes) that has no local variables, no named return values and no side-effects. Such a procedure can be called an arbitrary number of times.

- a procedure that no longer directly implements a function can be turned into an INLINE to efficiently redirect the client to the new implementation.

During development, you should comment out the keyword "INLINE" until the program runs correctly, since an inline procedure cannot be debugged.

Minimizing Code

Minimizing code is important since larger modules mean larger swap units, hence longer swapping times. Often a module is only a few words over a page boundary; by reducing the size of the code slightly, you can make the module fit.

In addition to minimizing code, you should try reducing the size of Global Frames (GF) and the number of intermodule references (Links). Both of these structures must be swapped in, in addition to the code. You can used the Lister to show the size of the module's Global frame and the number of links.

Package Code

"Packaging" code places hot and cold code into separate swap units. The Packager is a post processor that lets you keep the logical structure of your modules, while dividing up the physical code. The packaging process itself is beyond the scope of this course.

Comment Unused Code

Comment out unused procedures. Commenting not only reduces the amount of generated code, but also eliminates any links referenced only by the commented code.

```
-- commenting eliminates code, and the link for FooDefs
<< OldProc: PROC[...] RETURNS[...] = {
  FooDefs.ProcOnlyCalledHere[...];
  ...
  }; >>
```

Eliminate Redundant Code

Eliminate code that deals with situations that never occur or that is redundant. You should document your assumptions carefully to account for future changes.

```
GetContext: PROCEDURE [body: Window.Handle] RETURNS [mydata: Defs.Data] = {
  mydata ← Context.Find[context, body];
  IF mydata = NIL THEN ERROR; --just in case.
  RETURN [mydata];
  };

Entry point: 1,   Frame size index:  0
    1245:   SLD2
    1260:   LLD0
    1246:   LGB      26
    1250:   LLD2
    1251:   EFCB     26
    1253:   PLD0
    1254:   IOR
    1255:   JNZ3        (1260)
    1256:   KFCB
    1261:   RET
Instructions: 10, Bytes: 14
```

VS.

```
-- the code should be fully debugged
GetContext: PROCEDURE [body: Window.Handle] RETURNS [mydata: Defs.Data] = {
  mydata ← Context.Find[context, body];
  };

Entry point: 1,   Frame size index:  0
    1245:   SLD0
    1246:   LGB      26
    1250:   LLD0
    1251:   EFCB     26
    1253:   PLDB      2
    1255:   RET
Instructions: 6, Bytes: 10
```

The code to test the **NIL** case is only needed before the program is fully debugged; thus, it can be removed.

Minimizing Code With INLINEs

Let's take the GetContext procedure one step further and make it inline.

```
Expert: MenuData.MenuProc = {
  body: Window.Handle ← StarWindowShell.GetBody[[window]];
  data: Defs.Data ← GetContext[body];
  data.level ← expert;
  };
```

```
GetContext: PROCEDURE [body: Window.Handle] RETURNS [Defs.Data] = INLINE {
  RETURN[Context.Find[context, body]];
  };  -- generates no procedure body
```

-- with INLINEs, the call to GetContext generates this code

```
    1105: LG2
    1106: LLD2
    1107: EFC2
    1110: SLD4
```

Instructions: 4, Bytes: 4

VS.

```
GetContext: PROCEDURE [body: Window.Handle] RETURNS [Defs.Data] = {
  RETURN[Context.Find[context, body]];
  };  -- generates code for procedure body
```

-- without INLINEs, the same call to GetContext generates this code

```
    1106: LFC     1244
    1111: SLD4
```

Instructions: 2, Bytes: 4

So by using INLINEs in this situation, we have added two instructions, kept the code the same size, and eliminated a procedure call.

Pilot's Inline Interface

In addition to making your own procedures INLINE, there is a separate **Inline** interface in Pilot. You should become aware of the procedures it contains since they are optimized by being in microcode. You should use the Inline interface when:

Copying large amounts of data to or from VM.

```
  Inline.LongCOPY[...];
```

Performing LONG arithmetic.

```
  Inline.LongMult[...];
  Inline.LongDiv[...];
```

Operating on bits (i.e. shifts, rotates, etc.)

```
  Inline.BITSHIFT[...];
  Inline.BITROTATE[...];
```

Reduce the Number of Links

You should attempt to reduce the number of links (references to external procedures) that each module has.

When initialization code references many external procedures that are not referenced by any other code, consider moving the init code to another module to minimize the size of the links array.

When you split modules, try to place procedures that use similar sets of links together. For example, all the code that references the `StarWindowShell` interface would be placed in a single module.

Do not qualify procedures contained in the same modules (i.e. don't write `Foo.Proc` when in the module that implements `Proc`).

Optimize SELECT statements

Avoid using select statements to perform simple mathematical mappings.

```
Fruit: TYPE = {apple, orange, grape, tomato, cherry};
i: CARDINAL ← SELECT x FROM
  apple => 1,
  orange => 2,
  grape => 3,
  tomato => 4,
  ENDCASE => 5;
```

vs.

```
i: CARDINAL ← Fruit.x.ORD + 1;
```

Note - your mathematical expression may not be very meaningful to other programmers; thus, you should always carefully document your assumptions.

Special Mesa Machine Optimizations

By understanding more about how the Mesa machine works, you can take advantage of special machine instructions.

Loops should start with zero when possible because Mesa has a special machine instruction for testing `[0..n)` where `n` is positive. Thus, `IN [0..4)` generates fewer bytes of code than `IN [-2..2)`.

The first sixteen words of a Mesa RECORD are the quickest to access; thus you should place the most frequently accessed data in the first sixteen words of all records.

Access to fields in unpacked records requires less code than for packed records since the fields are word-aligned. In addition, packed records must be unpacked before accessing any of their fields.

Listing Assembly Language Code

The Lister utility (documented in the *Xerox User's Guide*) allows you to display the assembly language code that is generated by the compiler. Thus, you can compare different algorithms and see which one produces less/ faster code. The Lister subsystem that you should use is "code."

You can generate the assembly language code for a given ".bcd" file by typing "`lister code[foo.bcd]`" into the Executive. The Lister will then generate a file called "`foo.cl`" that contains the code.

Locating Hot Procedures

As a programmer, you have some idea about what procedures will be used most frequently. To find out exactly which procedures in a module are invoked and how often, you can use the ProcList tool. The ProcList tool lets you "spy" on a particular module during some period of time (e.g. while playing a particular game).

From the information below, we can determine that **MyNotifyProc** and **GetContext** are hot procedures. In addition, **InRange** and **Word** are called the same number of times, so they might be combined.

```
RandomImpl.ENTRY VECTOR
RandomImpl.Initialize 1
RandomImpl.InRange 246
RandomImpl.MAIN 1
RandomImpl.Word 246
TipoMsgImpl.ENTRY VECTOR
TipoMsgImpl.GetMessageHandle 2
TipoTipImpl.ENTRY VECTOR
TipoTipImpl.MyNotifyProc 355
TipoTipImpl.SetUpTipTable 1
TipoTipImpl.Start 1
TipoTipImpl.Stop 1
TipoTipImpl.Trans 63
TipoToolImpl.AdvanceLines 188
TipoToolImpl.CheckUserAction 63
TipoToolImpl.DisplayAllLetters 2
TipoToolImpl.DisplayBox 254
TipoToolImpl.DisplayColumn 66
TipoToolImpl.DisplayGrid 2
TipoToolImpl.DisplayLetter 66
TipoToolImpl.DisplayVertLines 2
TipoToolImpl.ENTRY VECTOR
TipoToolImpl.GenericProc 1
TipoToolImpl.GetContext 424
TipoToolImpl.GetRandomLetter 58
TipoToolImpl.InitializeGame 1
TipoToolImpl.InvertPosition 154
TipoToolImpl.MakeShell 1
TipoToolImpl.MyRedisplayProc 2
TipoToolImpl.PictureProc 1
TipoToolImpl.PrintStats 1
TipoToolImpl.Start 1
TipoToolImpl.StringFromKey 66
```

Locating Hot Procedures (cont'd)

The procedures on the previous slide were grouped logically into modules. The Packager would generate swap units that reflect the frequency that each procedure is called. You could package the code yourself using the data generated by ProcList. Only a few changes are needed to greatly reduce the number of intermodule calls.

A major disadvantage of moving procedures is that your program begins to lose its readablity!

```
-- init code swapped out after use
RandomImpl.Initialize 1
TipoToolImpl.GenericProc 1
TipoToolImpl.MakeShell 1
TipoToolImpl.PictureProc 1
TipoToolImpl.Start 1
TipoToolImpl.PrintStats 1
TipoToolImpl.InitializeGame 1
TipoTipImpl.SetUpTipTable 1

-- Hot code should stay resident
RandomImpl.InRange 246
RandomImpl.Word 246
TipoTipImpl.MyNotifyProc 355
TipoTipImpl.Trans 63
TipoToolImpl.AdvanceLines 188
TipoToolImpl.GetContext 424
TipoToolImpl.InvertPosition 154
TipoToolImpl.CheckUserAction 63
TipoToolImpl.DisplayBox 254
TipoToolImpl.DisplayColumn 66
TipoToolImpl.DisplayLetter 66
TipoToolImpl.StringFromKey 66
TipoToolImpl.GetRandomLetter 58

-- major display code called at beginning and end of program
TipoMsgImpl.GetMessageHandle 2
TipoToolImpl.DisplayAllLetters 2
TipoToolImpl.DisplayGrid 2
TipoToolImpl.DisplayVertLines 2
TipoToolImpl.MyRedisplayProc 2
TipoTipImpl.Stop 1
```

Avoiding Recomputation

You should avoid dereferencing multiple pointers whenever possible. As you begin using more complex data structures, it is better to define a local variable then to dereference similar expressions. OPEN statements will hide the problem but not make it go away.

```
-- generates more and slower code (the OPEN simply hides the dereferencing)
OPEN a↑.b↑.c[d]↑.e[f];
count ← count + 1;
char ← charac;
prop ← property;
```

Generates this code:

```
a↑.b↑.c[d]↑.e[f].count ← a↑.b↑.c[d]↑.e[f].count + 1;
a↑.b↑.c[d]↑.e[f].char ← charac;
a↑.b↑.c[d]↑.e[f].prop ← property;
```

VS.

```
-- must allocate a local variable
p: LONG POINTER TO MyType ← @a↑.b↑.c[d]↑.e[f];
p.count ← p.count + 1;
p.char ← charac;
p.prop ← property;
```

Preallocate Data Nodes

When you allocate nodes from the system heap, each node that you allocate may come from a different disk page. To avoid this, you should create a private heap from which to allocate nodes. This is especially important for structures such as linked lists that may need to be searched. You must weigh the cost of creating a private heap vs. the cost of potential page faults in the system heap.

You should consider using arrays rather than linked lists. Even though arrays have compute time deficiencies, you will avoid the fragmentation that occurs when using linked allocation.

Minimize Data

- Data should be allocated from one big node rather than several small nodes to save on node overhead. Very large nodes can be allocated using the Pilot Space interface (i.e. ARRAYs vs. linked nodes).

- Minimize global frame size. There is currently a limit of 64k of global frame space for all processes. ViewPoint software uses a great deal of this allocation already, so you should make every attempt to minimize your impact on this resource.

- Minimizing local variables can result in smaller local frames, hence faster procedure calls. Remember that local frames are allocated from the MDS too; thus, recursive procedures should not have large local frames.

Reduce Global Data

- You can allocate all global variables associated with your application from a heap; this data can later be retrieved using procedures from the Context interface. Examples of this technique can be found in all of the exercises for this course.

- Data that is associated with all instances of an application (atoms, heaps, etc.), can also be allocated from a heap. Thus, the only global variable that your application needs is a long pointer to this record. The disadvantage of allocating the data from a heap is that you must dereference a pointer to access it.

Global Data Example

```
-------------- <<Waste of MDS>>
AtomList: TYPE = {open, takeCopy, takeMove, canYouTake, props};
-- Global variables
z: UNCOUNTED ZONE ← BWSZone.Permanent[];
atoms: ARRAY AtomList OF Atom.ATOM ← ALL[Atom.null];
bigWriter: XString.WriterBody ← XString.NewWriterBody[255, z];
name: XString.ReaderBody ← XString.FromSTRING["Name for menu"L];
count: CARDINAL ← 0;

Compiler...
lines: 20, code: 75, links: 3, frame: 35, time: 28
>>>>> uses 35 words of global frame! <<<<<

------------- <<Conserve MDS>>
AtomList: TYPE = {open, takeCopy, takeMove, canYouTake, props};
Globals: TYPE = LONG POINTER TO GlobalData;
GlobalData: TYPE = RECORD [
  z: UNCOUNTED ZONE,
  atoms: ARRAY AtomList OF Atom.ATOM,
  bigWriter: XString.WriterBody,
  name: XString.ReaderBody,
  count: CARDINAL];

-- Global variables
g: Globals ← NIL;

-- initialize
Init: PROC = {
    z: UNCOUNTED ZONE ← BWSZone.Permanent[];
    rb: XString.ReaderBody ← XString.FromSTRING["Name for menu"L];

    g ← z.NEW[GlobalData ← [
      z: z,
      atoms: ALL[Atom.null],
      bigWriter: XString.NewWriterBody[255, z],
      name: XString.CopyToNewReaderBody[@rb, z],
      count: 0]];
  };

Compiler ...
lines: 34, code: 124, links: 4, frame: 6, time: 30
>>> Provides same data structures, but only uses 6 words. <<<
```

[illegible]

XString Usage

Creating and destroying **XString.WriterBodys** can be expensive. If you are using the WriterBody for formatting output, then it is better to allocate a single large WriterBody and reuse it.

You should never use **XString.FromSTRING** for global strings; these are better allocated from a heap. In general, all strings should be stored and accessed via XMessage.

Heaps

Heaps are used for storing data for all or part of the duration of a program. Indiscriminate heap usage can result in unused heap pages, expansion or fragmentation of the heap. By examining the behavior of your program, you can make decisions that will minimize these undesirable effects.

Growing a sequence is an expensive heap operation because of the time spent copying information and the need to allocate additional space for the new sequence. The cost of allocating the new sequence is the more serious problem since it may involve growing the heap and it creates a lot of unused heap space. If the heap is also used for other structures, then the heap mechanism may not be able to coalesce freed sequences.

Too many heaps (distributed heaps) can cause problems as well. Since heaps are located in VM, they must be swapped into real memory before they can be used. You may end up swapping each time you make a reference to a different heap.

NSFiling

Since NSFiling involves the disk, it is inherently slow. Because of this, you should minimize the number of calls to NSFile. When you must perform lengthy filing operations (i.e. remote file transfers), you may want to do so in the background.

Operation	Avg Time (sec)
ChangeAttributes (numeric)	.224
ChangeAttributes (string values)	.245
Create directory (size = 0)	.761
Create directory (size = 10)	.987
Create directory (size = 100)	1.288
Delete directory (size = 0)	.538
Delete directory (size = 10)	.677
Delete directory (size = 100)	.940
Create temp file (size = 0)	.357
Create temp file (size = 10)	.450
Create temp file (size = 100)	.729
Delete temp file (size = 0)	.293
Delete temp file (size = 10)	.367
Delete temp file (size = 100)	.613
Listing files (number = 10)	.266
Listing files (number = 50)	1.251

Some recent benchmark times for common NSFile operations

NSFiling (cont'd)

Opening an NSFile can take between 22 and 500 milliseconds, depending on the size of the file. Since this takes so long, it is better to leave files open until they are no longer needed. Similarly, mapping a file to VM can take up to 300 milliseconds. Since mapping is so expensive, you may want to try to map all portions of a file that will be needed and let Pilot worry about managing the space.

However, mapping a large file for an extended period of time will impact other applications.

Concurrency

It would be nice to be able to retrieve a large folder from a server and still be able to edit some other file. However, the current implementation performs file copies in the foreground. You can obtain a significant perceived performance improvement by forking such operations when possible.

There are some obstacles with concurrent filing operations. First, the Attention interface will only post messages from a foreground process. However, from a forked process, you must use the BackgroundProcess interface to post messages to the user. Second, all filing requests within a session are executed consecutively, not concurrently. In order to have concurrent filing operations, you must establish a separate session for each file operation (discussed later).

BackgroundProcess

The BackgroundProcess (BP) interface provides user feedback and control facilities to clients that want to run in a process other than the Notifier. A process registers itself by calling BackgroundProcess.ManageMe. ManageMe requires a call-back procedure to perform the actual work, and it is within this call-back that messages may be posted. If the call-back procedure is prepared to catch the ABORTED SIGNAL, then abortable should be set to TRUE.

```
BP.ManageMe: ManageProc;

BP.ManageProc: TYPE = PROCEDURE [
  name: XString.Reader,
  callBackProc: BP.CallBackProc,
  window: Window.Handle ← NIL,
  icon: Containee.DataHandle ← NIL,
  context: LONG POINTER ← NIL,
  abortable: BOOLEAN ← FALSE]
  RETURNS[finalStatus: BP.FinalStatus];

BP.CallBackProc: TYPE = PROCEDURE [context: LONG POINTER]
  RETURNS[finalStatus: BP.FinalStatus];

BP.FinalStatus: TYPE = MACHINE DEPENDENT {importantFailure(0),
  failure, quietSuccess, success, aborted, firstFree, last(15)};
```

Establishing Separate NSFile Sessions

Most NSFile operations require a `NSFile.Session` handle, which is normally defaulted. Within this default session, all file requests are handled consecutively. If you want to perform concurrent NSFile operations you must establish a separate session handle. You must first acquire an `Auth.IdentityHandle` by calling `Atom.GetProp`. By calling GetProp with atoms "CurrentUser" and "IdentityHandle", you will receive as the `Pair.value` the `Auth.IdentityHandle` for the currently logged-in user.

```
Atom.GetProp: PROCEDURE[onto, prop: Atom.ATOM]
  RETURNS[pair: Atom.RefPair];

Atom.RefPair: TYPE = LONG POINTER TO READONLY Atom.Pair;

Atom.Pair: TYPE = RECORD[prop: Atom.ATOM, value: Atom.RefAny];

Atom.RefAny: TYPE = LONG POINTER;
```

Once you have the `Auth.IdentityHandle`, you can establish a new session by calling `NSFile.Logon`.

```
NSFile.Logon: PROCEDURE[identity: Auth.Identity]
  RETURNS[NSFile.Session];
```

Example:

userAtom ← Atom.MakeAtom["CurrentUser"L];
idAtom ← Atom.MakeAtom["IdentityHandle"L];
user ← Atom.GetProp[userAtom, idAtom]^.

Concurrency Example

```
-- CopyFileImpl.mesa

DIRECTORY
  Atom,
  Attention,
  Auth,
  BackgroundProcess,
  CopyDefs,
  Courier,
  MenuData,
  NSFile,
  Process,
  Selection,
  StarDesktop,
  StarWindowShell,
  Window,
  XMessage
  XString;

CopyFileImpl: PROGRAM
  IMPORTS Atom, Attention, BackgroundProcess, Courier, CopyDefs, NSFile, Process,
Selection, StarDesktop, StarWindowShell, XMessage, XString
  EXPORTS CopyDefs = {OPEN Defs: CopyDefs;


  -- get reference to destination. if no dest selected, copy source to desktop
  Confirm: PUBLIC MenuData.MenuProc = {
    body: Window.Handle ← StarWindowShell.GetBody[[window]];
    data: Defs.Data ← Defs.GetContext[body];
    data.dest ← GetSelectedFile[!Defs.FileProblem =>
      {data.dest ← StarDesktop.GetCurrentDesktopFile[];
       CONTINUE}];
    Process.Detach[FORK DoBackgroundCopy[data]];
    };

  -- Stores a reference to the selected file and prompts for dest
  Copy: PUBLIC MenuData.MenuProc = {
    body: Window.Handle ← StarWindowShell.GetBody[[window]];
    data: Defs.Data ← GetContext[body];
    mh: XMessage.Handle = Defs.GetHandle[];
    msg: XString.ReaderBody ← XMessage.Get[mh, Defs.keys.clickConfirm];
    data.source ← Defs.GetSelectedFile[!FileProblem => GOTO Exit];
    Attention.Post[@msg];
    EXITS
      Exit => NULL;
    };
-- Call the CopyFile procedure to copy the source file to the destination. Establish a separate session
-- for the copy so that other NSFile operations can occur concurrently. After the file is copied,
-- all handles are closed, the session is closed, and a message is printed
DoBackgroundCopy: PROCEDURE[data: Defs.Data] = {
  -- the call-back proc for ManageMe must be internal
  CopyFile: BackgroundProcess.CallBackProc = {
    good: XString.ReaderBody ← XString.FromSTRING["File Copied"L];
    currentUser: Atom.ATOM ← Atom.MakeAtom["CurrentUser"L];
    identityHandle: Atom.ATOM ← Atom.MakeAtom["IdentityHandle"L];
    identity: Auth.IdentityHandle ← Atom.GetProp[
      currentUser, identityHandle]↑.value;
    session: NSFile.Session ← NSFile.Logon[identity];
    source: NSFile.Handle ← NSFile.OpenByReference[
      reference: data.source, session: session];
    dest: NSFile.Handle ← NSFile.OpenByReference[
      reference: data.dest, session: session];
    dummy: NSFile.Handle ← NSFile.Copy[
      file: source, destination: dest, session: session
      !NSFile.Error => {NSFile.Logoff[session: session]; GOTO Exit}];
    NSFile.Close[dummy];
    NSFile.Close[source];
    NSFile.Close[dest];
    NSFile.Logoff[session: session];
    Attention.Post[@good];
    RETURN[success];
   EXITS Exit => {
     mh: XMessage.Handle = Defs.GetHandle[];
     bad: XString.ReaderBody ← XMessage.Get[mh, Defs.keys.notCopied];
     Attention.Post[@bad]; <<close files>>;
     RETURN[failure]};
   };

  Process.SetPriority[Process.priorityBackground];
  [] ← BackgroundProcess.ManageMe[name: @name, callBackProc: CopyFile];
  };

-- convert the selection to a file; if there is a problem print a message and
-- raise the SIGNAL Defs.FileProblem
GetSelectedFile: PUBLIC PROC
RETURNS[ref: NSFile.Reference ← NSFile.nullReference] = {
  element: Selection.Value ← Selection.Convert[target: file];
  refPtr: LONG POINTER TO NSFile.Reference ← element.value;
  handle: NSFile.Handle ← NSFile.nullHandle;
  IF refPtr = NIL THEN {
    mh: XMessage.Handle = Defs.GetHandle[];
    msg: XString.ReaderBody ← XMessage.Get[mh, Defs.keys.badFile];
    Attention.Post[@msg];
    SIGNAL Defs.FileProblem};
  ref ← refPtr↑;
  Selection.Free[@element];
  };
}..
```

Summary

All of the techniques that we have discussed will improve the performance of your programs. However, in some cases the changes may not be noticeable because the time savings is in milliseconds. You may notice the change with larger programs or programs that perform a lot of swapping.

Many of the methods discussed make your programs harder to read and debug. In general, you should strive to first make your program work correctly, and then to make it run faster. You must also consider that other programmers may have to support your program in the future, so readability is important.

Programming Lab Hints

The Training Lab is located in room C401, next door. There are 20 machines available, so there should be one for everyone. The machines are already set up to contain all of the software you will need for the week. You will spend the afternoons completing one or more programming exercises. Before you start, there are a few things that you should know about the machines and the assignments:

1) The afternoon labs are "Free Form". That is, you may come and go as you please, taking breaks and lunch as you wish. We do expect, however, that you do put some effort into working on the day's assignment.

2) In order for the instructors to understand how well the information is getting through, we would like each of you to show one of the instructors a running version of your assignment. (We may choose to test a few key things to see how robust your implementation actually is.)

3) There is an information card on each keyboard assigning you a logon name and password as well as other information about your machine. You do not need to be logged in to work in XDE or VP, although you do need to be logged in to perform any operations over the net (e.g. Printing something). If you have your own logon name on the Xerox net, you may use it if you like.

4) The lab machines have an established SearchPath of directories. These include a working directory (AVPWD), a BWS Interface directory (VPDefs), etc. To avoid problems, you should not alter this SearchPath.

5) All code that you write will be contained in "template" modules; these modules will be specified in the exercise handouts. These templates contain "gutted" procedures with extensive comments. So after reading the exercise handout, you should locate the proper template module(s), read the comments, and then begin writing your code. You should not modify code in any other program modules; they only need to be compiled and bound. When compiling, you may recieve warnings indicating that a procedures or data type was declared but not used; this indicates that you did not use some symbol that is in our solution. If the symbol is part of a user feedback function, you may wish to disregard the warning, but you should give careful attention to other symbols.

   After you get your template module to compile, you should do one of two things:

   1) edit the **.config** file associated with the program and change all instances of **fooXXXImpl** to **fooXXXImplTemplate**, or

   2) change the name (both internally and externally) of the **fooXXXImplTemplate** to **fooXXXImpl**.

6) You may (and should) choose your boot switches for Viewpoint depending on what your application is testing. If you do not rely on any VP applications (especially the Document Editor) then you should boot VP with the "**ONdy\365**" switches; these can be set in the Command Central Option Sheet. The capital "N" means don't run any applications that are not on the Run-line of Command Central. One advantage of the "N" switch is that you don't need to specify a password when logging in, you simply select the Start command in the Logon Option Sheet. Another major advantage is that booting will take only about 4 minutes instead of about 12. Although, when you run the applications that call into DocInterchangeDefs, GraphicsInterchangeDefs, etc, then you will need to run the Document Editor. (It is the Document Editor application that supplies the implementations for these Interfaces.) In this case you should boot VP with the "**Ody\365**" switches.

7) XDE Documentation is located in the back of the room. There are copies of the Mesa Language Manual, Mesa Programmer's Manual, Pilot Programmer's Manual, XDE User's Guide, ViewPoint Programmer's Manual, and Services Programmer's Manual. You are encouraged to use these throughout the course, although they are to remain here for future students and classes.

8) You may wish to personalize the *User.cm* on your workstation by changing the *HardCopy PrintedBy:* option, or the default ***Brush***, the ***Logon Name***, etc. It is okay to do so, but you do not have to.

9) At the end of each day, send a mail message to us mentioning what you liked or disliked about the day's work, any typos you noticed, bugs in the programs (ours not yours), or suggestions you may have. (We are constantly modifying the material based on students' suggestions.) The message may be as brief or lengthy as you like. We have found, though, that students who wait until Day 5 to summarize their thoughts in one message, tend to forget some of the thoughts that they had earlier in the week.

10) Before you leave for an extended period of time (especially overnight), be sure to run some sort of DMT on the screen in order to protect them. There are many to choose from (e.g. DMT, BrushDMT, Poly, SpaceOut, KineticFractal, etc.).

11) Most Importantly: **ASK QUESTIONS!** We are here to help you. Your lab instructors will be happy to answer any questions concerning the assignments or the lecture material, so don't be shy. Please report any bugs in the programs, or mistakes in the slides.

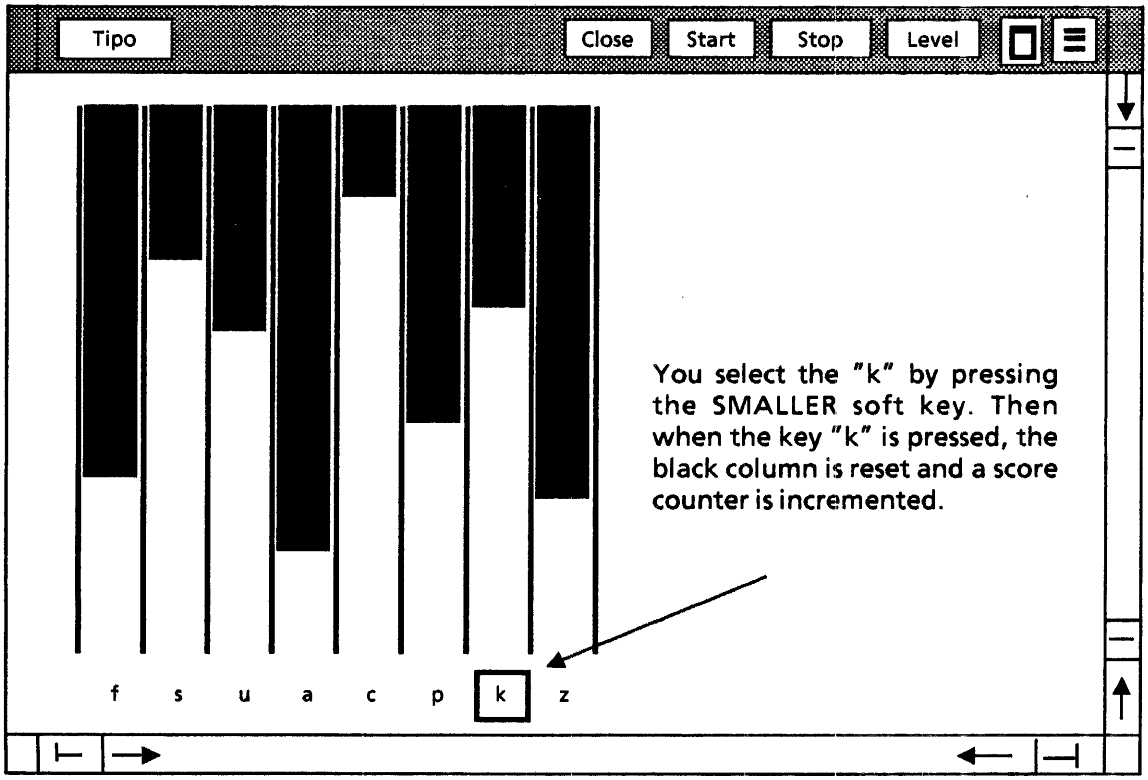
E-Mail address: Mackay:OSBU North:Xerox

Lab Exercise: TIP

Tipo

Explanation of Game: The purpose of this game is to improve your typing skills and have fun (not necessarily in that order). The tool contains eight columns, with random letters at the bottom of each column. One of these letters will be highlighted. A black line in each column grows toward the bottom of the column. You must select the correct column and type the letter at the bottom of that column before the black line reaches the bottom. If you succeed, your score will increase, you will hear a high pitched beep, a new letter will be placed at the bottom of the column, and the blick line is reset to the top. If the line reaches the bottom before you type the letter, you will hear a low pitched beep, the black line will reset and a new letter will be placed at the bottom of the column. You get 50 chances altogether; the game will display your performance at the end of the game or when you select the **Stop** command.

User Interface: You select the column that you want to type in by using the soft keys at the top of the keyboard (the ones marked CENTER, BOLD, ITALICS, UNDERLINE, SUPERSCRIPT, SUBSCRIPT, SMALLER, & DEFAULTS). The letter below the selected column will video invert. Once you've selected the column in this fashion, you type the corresponding letter on the keyboard. A low-pitched beep sounds if a black line reaches the bottom of the column before you type the correct letter. You can select one of three levels of difficulty: beginner, intermediate, or expert which alters the speed at which the line grows (i.e. there is less time to hit the appropriate key)..



You select the "k" by pressing the SMALLER soft key. Then when the key "k" is pressed, the black column is reset and a score counter is incremented.

Assignment: The tool uses several TIP features. First, Tipo uses its own TIP table to map the function keys to column numbers. This involves creating the table and associating it to the proper window. Second, the black lines are grown in a periodic notifier, with the amount of time between notifications dependent upon the level of difficulty. Finally, the NotifyProc that handles all user actions for the tool must recognize atoms and integers . Your assignment is to write all the TIP parts of **TipoTIPImplTemplate**. The procedures you must modify are *MyNotifyProc, Start, Stop,* and *SetUpTipTable*. Hints are provided in the program module.

Modules: Tipo.config
TipoDefs.mesa
TipoMsgImpl.mesa
TipoTIPImplTemplate.mesa
TipoToolImpl.mesa

Lab Solution: TIP

Tipo

From TipoTipImpl...

```
-- You'll have to recognize the "atom" and "int" variants from the input. The atom "advance" comes
-- from the periodic notifier: if data.numLetters hasn't exceeded Defs.MaxLetters then call
-- AdvanceLines, otherwise call Stop. If the atom "enter" is recognized, you'll set the input focus
-- and the translator for this program, and push the table Tipo.TIP; if the atom is "exit," you
-- must restore the old translator and pop Tipo.TIP. If the variant is int, that means one of the
-- function keys was pressed...you have to call InvertPosition to dehighlighted the current
-- position, set  data.currentPosition to the number passed in to MyNotifyProc, and call
-- InvertPosition to highlight the new position.
MyNotifyProc: PUBLIC TIP.NotifyProc = {
  data: Defs.Data ← Defs.GetContext[window];
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      atom =>
       SELECT z.a FROM
         advance => IF data.numLetters >= Defs.MaxLetters THEN Stop[window]
                    ELSE Defs.AdvanceLines[data, window];
         enter => {TIP.SetInputFocus[w: window, takesInput: FALSE];
                   data.oldTrans ← TIP.SetCharTranslator[
                     table: TIPStar.GetTable[blackKeys],
                     new: data.translator];
                   TIPStar.PushTable[softKeys, table]};
         exit =>  {[] ← TIP.SetCharTranslator[
                     table: TIPStar.GetTable[blackKeys],
                     new: data.oldTrans];
                   TIPStar.PopTable[softKeys, table]};
         ENDCASE;
     int => {Defs.InvertPosition[data.currentPosition, window];
             data.currentPosition ← z.i;
             Defs.InvertPosition[data.currentPosition, window]};
     ENDCASE;
   ENDLOOP;
  };
```

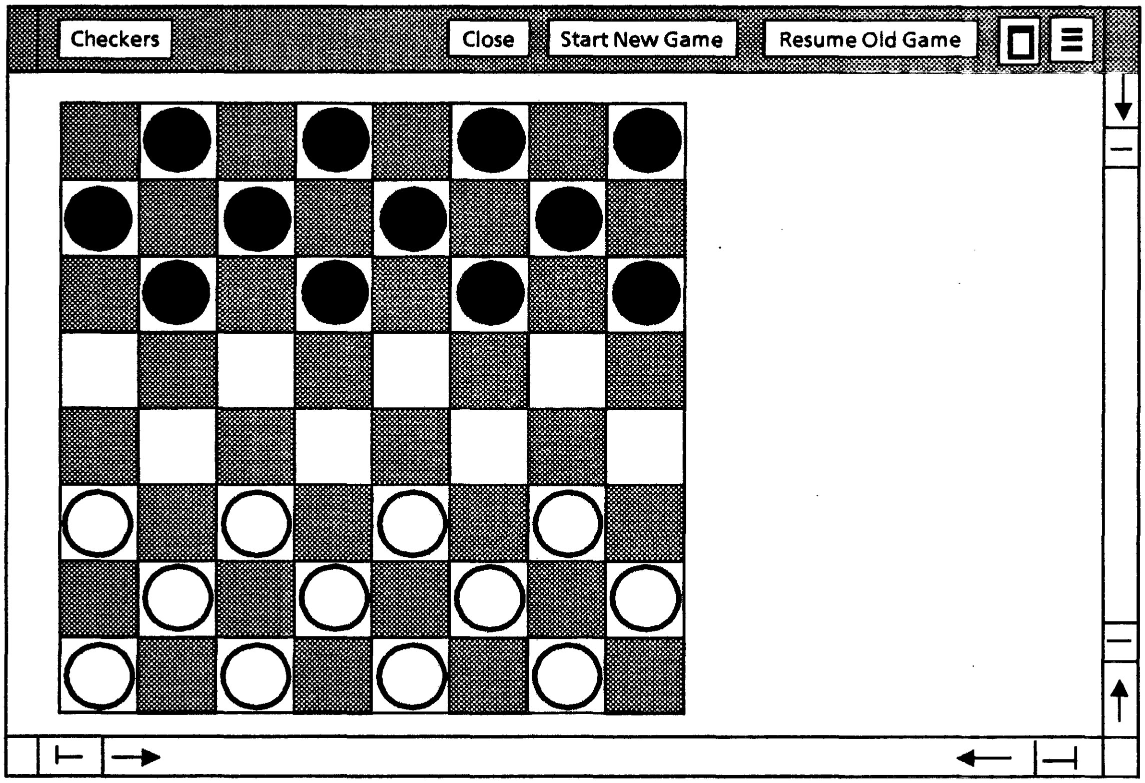
```
-- Write the TIP table for the function keys at the top of the keyboard. The result for each key
-- should be an integer corresponding to that key's position. For example, the CENTER key should
-- have a result of 0, the BOLD key should have a result of 1, etc. These integers will be used by
-- the NotifyProc as data.currentPosition. Call TIP.CreateTable to set up the TIP table.
-- Hint - there is no atom named LARGER.
SetUpTipTable: PUBLIC PROC [window: Window.Handle] = {
  fileName: XString.ReaderBody ← XString.FromSTRING["Tipo.TIP"L];
  contents: XString.ReaderBody ← XString.FromSTRING["
  SELECT TRIGGER FROM
    CENTER Down => 0;
    BOLD Down => 1;
    ITALICS Down => 2;
    UNDERLINE Down => 3;
    SUPERSCRIPT Down => 4;
    SUBSCRIPT Down => 5;
    SMALLER Down => 6;
    DEFAULTS Down => 7;
  ENDCASE...
  "L];
  table ← TIP.CreateTable[file: @fileName, contents: @contents
    ! TIP.InvalidTable => RESUME];
  IF table = NIL THEN {--post a message--};
  };

-- This procedure is called by the Start MenuData.MenuProc procedure. Set the translator variable
-- passing window as the data argument. Create a periodic  notifier and store it in the data object.
-- Reset the counters data.numHits and data.numLetters to zero and star the game.
-- The milliSeconds field should depend on the value of data.level.
Start: PUBLIC PROC [window: Window.Handle ] = {
  data: Defs.Data ← Defs.GetContext[window];
  data.translator ← [proc: Trans, data: window];
  data.numHits ← 0;
  data.numLetters ← 0;
  data.inProgress ← TRUE;
  data.periodicNotify ← TIP.CreatePeriodicNotify[
    window:window,
    results: @myResults,
    milliSeconds: SELECT data.level FROM
                    beginner => beginnerPeriod,
                    intermediate => intermediatePeriod,
                    ENDCASE => expertPeriod];
  };

-- Cancel the periodic notifier, stop the game, and Call PrintStats to print the
-- statisitics of the player's performance.
Stop: PUBLIC PROC[window: Window.Handle] = {
  data: Defs.Data ← Defs.GetContext[window];
  data.periodicNotify ← TIP.CancelPeriodicNotify[data.periodicNotify];
  data.inProgress ← FALSE;
  Defs.PrintStats[window: window];
  };
```

Lab Exercise: Selection

Checkers

Explanation of Game: In this game for peasants and little kids, you move your pieces in the direction of your opponent, hoping to reach their side of the board. If one of your pieces reaches the other side, it becomes a "king" and can henceforth move freely around the board. You can jump an opponent's piece if it's adjacent to yours and the square immediately behind it is unoccupied. A king can jump in either direction; a regular piece can only jump in the direction that will advance it towards the opponent's side of the board. A player wins when he's jumped all of the opponent's pieces.

User Interface: To move a checker, you select the desired checker and then select a new location for that checker. When you select a new square, the checker is erased from the previous square and drawn in the new square. You select a checker by pointing at it with the mouse and clicking point down. Standard checker rules apply. The tool checks the legality of moves, although it doesn't keep track of who's turn it is. A jumped checker will disappear automatically.



Assignment: The tool uses a unique selection target type. When you click the mouse over a checker, that checker becomes the current selection. Your assignment is to write the selection manager part of **CheckersSelectionImplTemplate**. This consists of the *ConvertProc*, *ActOnProc*, and *ValueCopyMoveProc*. In addition, you should write the procedure *SelectChecker*, which actually sets the selection. Hints are provided in the program module.

Modules: Checkers.config
CheckersBitmapImpl.mesa
CheckersDefs.mesa
CheckersImpl.mesa
CheckersMsgImpl.mesa
CheckersSelectionImplTemplate.mesa

Lab Solution: Selection

Checkers

```
-- Called when user points over a square that contains a checker; it uses the call back procs
-- ActOnSelection & ConvertSelection to handle marking and clearing as well as obtaining the
-- value and moving the checker. This procedure simply sets the selection.
SelectChecker: PUBLIC PROC[square: Defs.Square] = {
  Selection.Set[
    pointer: square,
    conversion: ConvertSelection,
    actOn: ActOnSelection];
  };

-- Highlights the current selection and clears the old selection when a new one is made
-- (call-back from set). This proc should handle the following action: mark, unmark, &
-- clear. Post a msg using Defs.kunknownAction for actions not handled by this proc.
ActOnSelection: Selection.ActOnProc
<<[data: ManagerData, action: Selection.Action]
RETURNS[cleared: BOOLEAN ← FALSE]>> = {
  square: Defs.Square ← NARROW[data, Defs.Square];
  SELECT action FROM
    mark => IF ~square.marked THEN
     {InvertChecker[square];
      square.marked ← TRUE};
    unmark => IF square.marked THEN
     {InvertChecker[square];
      square.marked ← FALSE};
    clear => cleared ← TRUE;
    ENDCASE => {
      mh: XMessage.Handle = Defs.GetMessageHandle[];
      msg: XString.ReaderBody ← XMessage.Get [mh, Defs.kunknownAction];
      Attention.Post[@msg]};
  };
```

```
-- support conversion to the target type square; if successful, square will be returned.
-- Enumeration is not supported. If the target is checker and there IS a checker on the
-- square, then return the square, a pointer to the opsProcs, and, in the context arg,
-- the zone passed in.
ConvertSelection: Selection.ConvertProc
<<[data: ManagerData, target: TARGET, zone: UNCOUNTED ZONE,
info: Selection.ConversionInfo] RETURNS[value: Selection.Value]>> = {
  square: Defs.Square ← NARROW[data, Defs.Square];
  WITH i: info SELECT FROM
    query =>   -- only support conversion to checker no enumeration
      FOR c: CARDINAL IN [0..LENGTH[i.query]) DO
        i.query[c].difficulty ←
          IF (i.query[c].target = checker) AND (square.piece # none) THEN easy
          ELSE impossible
        ENDLOOP;
    convert =>
      SELECT target FROM
        checker => {   -- if a checker is selected return a pointer to it
          IF square.piece = none THEN RETURN[Selection.nullValue];
          RETURN[[value: square, ops: @opsProcs, context: LOOPHOLE[zone]]]};
        ENDCASE;
    enumeration => NULL;
    ENDCASE;
  RETURN[Selection.nullValue];
  };

-- make a copy of the selected checker and delete the checker from the manager's data
-- structure. v.value is a Defs.Square so deleting checker consists of setting occupant
-- to none. Raise Selection.Error if op = copy since copy isn't be supported. Clear the
-- selection and white out the piece in the old square.
MoveChecker: Selection.ValueCopyMoveProc
<<[v: Selection.ValueHandle, op: Selection.CopyOrMove, data: LONG POINTER]>> = {
  temp: Defs.Square ← NARROW[v.value, Defs.Square];
  zone: UNCOUNTED ZONE ← LOOPHOLE[v.context];
  IF op = copy THEN ERROR Selection.Error[invalidOperation];
  v.value ← zone.NEW[Defs.SquareObject ← temp↑];
  temp.piece ← none;
  Selection.Clear[];
  Display.White[temp.window, [temp.place, Defs.squareDims]];
  };
```

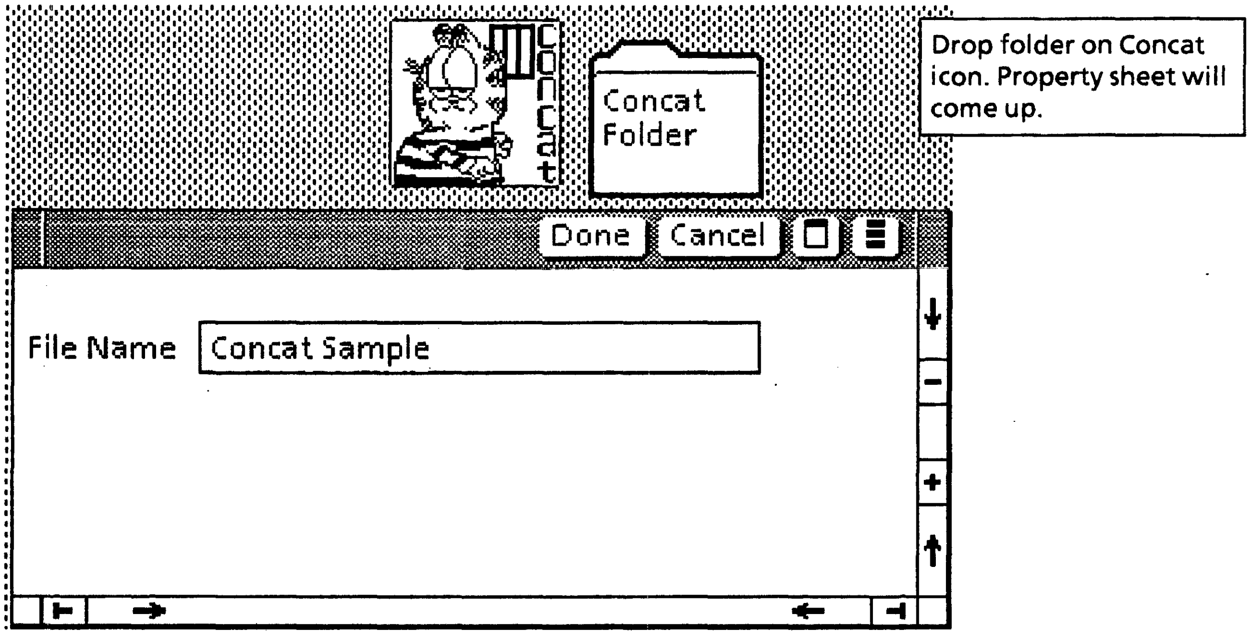
Lab Exercise: DocInterchangeDefs

Concat

Explanation of Application: Concat takes a document or a folder of documents as a selection and creates a single document that is the concatenation of the input.

User Interface: Select a VP document or a folder of VP documents and drop it on Concat. A property sheet will appear. You fill in the target file name and select **Done**. Concat will produce a new document with your specified name.

Current restrictions in this application: anchored frames are not sent to the target document; instead, a text message is sent to the document as a placeholder. Also, headings and footings are not enumerated in the input documents and thus do not appear in the target document.



Drop folder on Concat icon. Property sheet will come up.

File Name: Concat Sample

Assignment: In this assignment, you will write the code for generating a new document from the input. You will modify the module **ConcatDocImplTemplate** by completing the procedures *AppendFile*, *FieldProc*, *FirstNewParagraphProc*, *FirstPFCProc*, *InitEnumProcs*, *NewParagraphProc*, and *PFCProc*. The input documents are enumerated with calls to **DocInterchangeDefs.Enumerate**. You, as the programmer, will supply call-back procedures to handle each object that **Enumerate** finds.

Modules: Concat.config
ConcatDefs.mesa
ConcatDocImplTemplate.mesa
ConcatImpl.mesa
ConcatMsgImpl.mesa
ConcatPSheetImpl.mesa

Lab Solution: DocInterchangeDefs

Concat

```
-- Enumerate through the selected document and copy the contents to a target document. from is the
-- source file to be enumerated. You must enumerate the contents of the from file passed in.The
-- clientData argument will be set to the context data for all enumerations. Close the from doc
-- before exiting. The call to DID.StartCreation will occur in the FirstPFCProc once the
-- paragraph and PFC props have been obtained.
AppendFile: PUBLIC PROC[from :NSFile.Handle, data: Defs.Data] = {
  ref: NSFile.Reference ← NSFile.GetReference[file:from];
  oldDoc: DocInterchangeDefs.Doc;
  status: DocInterchangeDefs.OpenStatus;
  [oldDoc, status] ← DocInterchangeDefs.Open[docFileRef: ref];
  IF status # ok THEN GOTO Exit;
  [] ← DocInterchangeDefs.Enumerate[
    textContainer: [doc[h: oldDoc]], procs: @data.enumProcs, clientData: data];
  DocInterchangeDefs.Close[docPtr:@oldDoc];
  EXITS Exit => {
    rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.cantOpen.ORD];
    Attention.Post[@rb]};
  };

-- Copy the field passed in to the text container specified by clientData. Since fields may contain
-- information, enumerate the contents of the field. Before doing this, save the current
-- data.container in a local variable, set data.container to be the container for the field,
-- and then enumerate. Before exiting the procedure, restore the original data.container.
FieldProc: DocInterchangeDefs.FieldProc = {
  data: Defs.Data ← LOOPHOLE[clientData];
  newField: DocInterchangeDefs.Field ← DocInterchangeDefs.AppendField[
    to: data.container, fieldProps: fieldProps, fontProps: fontProps
    ! DocInterchangeDefs.Error => GOTO bad];

  -- Now enumerate any items within the field: save original container, put new
  -- container in data, then enumerate; afterwards, restore original container.
  originalTextContainer: DocInterchangeDefs.TextContainer ← data.container;
  data.container ← [field[h: newField]];
  [] ← DocInterchangeDefs.Enumerate[
    textContainer: [field[h: field]], procs: @data.enumProcs, clientData: data];
  DocInterchangeDefs.ReleaseField[@newField];
  data.container ← originalTextContainer;
  EXITS bad => NULL;
  };

-- Append a new paragraph to the target doc.
NewParagraphProc: DocInterchangeDefs.NewParagraphProc = {
  data: Defs.Data ← LOOPHOLE[clientData];
  [] ← DocInterchangeDefs.AppendNewParagraph[to: data.container, paraProps: paraProps
    ! DocInterchangeDefs.Error => {
        rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.cantOpen.ORD];
        Attention.Post[@rb];
        CONTINUE}];
};
```

```
-- For the 1st new paragraph of the first document to be concatenated, get its properties only;
-- i.e., don't append a new paragraph. Save these properties in the data object. The reason is
-- that DID.StartCreation appends a new paragraph & a PFC automatically to a new doc, so you
-- only want to copy the props to these default objects.
FirstNewParagraphProc: DocInterchangeDefs.NewParagraphProc = {
  data: Defs.Data ← LOOPHOLE[clientData];
  numTabs: CARDINAL;
  data.propsRecord ← paraProps↑;
  data.enumProcs.newParagraphProc ← NewParagraphProc;

  -- Copy the array of tab stops into data.propsRecord for temp storage
  numTabs ← paraProps.tabStops.LENGTH;
  IF numTabs = 0 THEN RETURN;
  data.tabStopPtr ← Defs.z.NEW[Defs.TabSeq[numTabs]];
  FOR i: CARDINAL IN[0..numTabs) DO
    data.tabStopPtr[i] ← data.propsRecord.tabStops[i];
    ENDLOOP;
  data.propsRecord.tabStops.BASE ← LOOPHOLE[data.tabStopPtr];
  data.propsRecord.tabStops.LENGTH ← numTabs;
  };

-- Start doc creation using the saved paragraph properties and the PFC props passed into this proc.
-- Set the PFC enumProc to PFCProc for any remaining PFC characters encountered and free any tabstop
-- data that you allocated.
FirstPFCProc: DocInterchangeDefs.PFCProc = {
  data: Defs.Data ← LOOPHOLE[clientData];
  startStatus: DocInterchangeDefs.StartCreationStatus;
  data.enumProcs.pfcProc ← PFCProc;
  [doc: data.doc, status: startStatus] ← DocInterchangeDefs.StartCreation[
    initialParaProps: @data.propsRecord, -- propsRecord from NewParagraphProc
    initialFontProps: fontProps,
    initialPageProps: pageProps];
  IF startStatus # ok THEN {
    rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.appendError.ORD];
    Attention.Post[@rb]};
  data.container ← [doc[h: data.doc]];
  IF data.propsRecord.tabStops.LENGTH # 0 THEN
    Defs.z.FREE[@data.propsRecord.tabStops.BASE];
  };

-- Append the PFC to the container specified by the clientData argument
PFCProc: DocInterchangeDefs.PFCProc = {
  data: Defs.Data ← LOOPHOLE[clientData];
  WITH z: data.container SELECT FROM
    doc => {
      [] ← DocInterchangeDefs.AppendPFC[to: z.h, pageProps: pageProps, fontProps: fontProps
        ! DocInterchangeDefs.Error => GOTO bad];
    };
  ENDCASE;
  EXITS bad => {
    rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.appendError.ORD];
    Attention.Post[@rb]};
};
```

```
-- ConcatDocImpl.mesa
-- Frank Yien       24-Jul-86 15:09:10
-- Mark Hahn        10-Mar-87 11:36:14  - updated for new doc interfaces
-- 27-Jul-88 18:49:40

-- Copyright (C) Xerox Corporation 1986. All rights reserved.

-- This module provides facilities for appending several documents to a new
-- document via DocInterchangeDefs.

-- PROCEDURES
-- AnchoredFrameProc:      Enum Proc - appends text "{ <frame type> }" to new doc
-- AppendFile:             Public proc that opens & enumerates a document
-- ColumnBreakProc:        Enum Proc - appends a column break
-- FieldProc:              Enum Proc - appends a field & enumerates field content
-- FirstNewParagraphProc: Enum Proc - stores the paragraph props in the data object
-- FirstPFCProc:           Enum Proc - starts document creation using saved para
--                                     - props and PFC props passed in
-- InitEnumProcs           Public proc that initializes the call back procs in data
-- NewParagraphProc:       Enum Proc - appends a new paragraph to textcontainer
-- PageBreakProc:          Enum Proc - appends a page break
-- PFCProc:                Enum Proc - appends a PFC to the textcontainer
-- SetTabStopDescriptor:  Allocates storage for tab stops
-- TextProc                Enum Proc - appends text to the new doc

DIRECTORY
  Attention USING [Post],
  ConcatDefs USING [Data, GetMessage, MessageKey, TabSeq, z],
  DocInterchangeDefs USING [AnchoredFrameProc, AppendColumnBreak, AppendField, AppendNewParagraph, AppendPageBreak, AppendPFC,
  AppendText, Close, ColumnBreakProc, Doc, Enumerate, Error, Field, FieldProc, NewParagraphProc, Open, OpenStatus, PageBreakProc,
  PFCProc, ReleaseField, StartCreation, StartCreationStatus, TextContainer, TextProc],
  NSFile USING [GetReference, Handle, Reference],
  UserTerminal USING [BlinkDisplay],
  XString USING [ReaderBody, unknownContext];

ConcatDocImpl: PROGRAM
  IMPORTS Attention, ConcatDefs, DocInterchangeDefs, NSFile, UserTerminal
  EXPORTS ConcatDefs = { OPEN Defs: ConcatDefs:

  -- Append msg in target doc saying what type of frame was encountered.
  -- We do not implement anchored frames other than appending a msg.
  AnchoredFrameProc: DocInterchangeDefs.AnchoredFrameProc = {
    data: Defs.Data ← LOOPHOLE[clientData];
    rb: XString.ReaderBody ← SELECT type FROM
      bitmap => Defs.GetMessage[Defs.MessageKey.bitmapFrame.ORD]
      cuspButton => Defs.GetMessage[Defs.MessageKey.cuspButtonFrame.ORD]
      equation => Defs.GetMessage[Defs.MessageKey.equationFrame.ORD],
      graphics => Defs.GetMessage[Defs.MessageKey.graphicsFrame.ORD],
      IMG => Defs.GetMessage[Defs.MessageKey.IMGFrame.ORD],
      table => Defs.GetMessage[Defs.MessageKey.tableFrame.ORD]
      text => Defs.GetMessage[Defs.MessageKey.textFrame.ORD],
    ENDCASE => Defs.GetMessage[Defs.MessageKey.unknownFrame.ORD]:
    DocInterchangeDefs.AppendText[to: data.container, text: @rb,
      textEndContext: XString.unknownContext, fontProps: anchorFontProps
      ! DocInterchangeDefs.Error => {
        rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.appendError.ORD];
        Attention.Post[@rb];
        CONTINUE}];
  };

  -- Enumerate through the selected document and copy the contents to a target
  -- document. from is the source file to be enumerated. You must
  -- enumerate the contents of the from file passed in.
  -- The clientData argument will be set to the
  -- context data for all enumerations. Close the from doc before exiting
  -- The call to DID.StartCreation will occur in the FirstPFCProc once the
  -- paragraph and PFC props have been obtained.
  AppendFile: PUBLIC PROC[from: NSFile.Handle, data: Defs.Data] = {
    openStatus: DocInterchangeDefs.OpenStatus;
    fromDoc: DocInterchangeDefs.Doc;
    [fromDoc, openStatus] ← DocInterchangeDefs.Open[NSFile.GetReference[file: from]];
    IF ~openStatus = ok THEN {UserTerminal.BlinkDisplay[]; RETURN;};
    [] ← DocInterchangeDefs.Enumerate[[doc[h: fromDoc]], @data.enumProcs, data];
    DocInterchangeDefs.Close[@fromDoc];
  };

  -- Copy the column break passed in to the text container specified by
  -- clientData.
  ColumnBreakProc: DocInterchangeDefs.ColumnBreakProc   {
    data: Defs.Data ← LOOPHOLE[clientData];
    WITH z: data.container SELECT FROM
      doc => [] ← DocInterchangeDefs.AppendColumnBreak[
                to:z.h, fontProps:fontProps
                ! DocInterchangeDefs.Error => {
                  rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.appendError.ORD]:
                  Attention.Post[@rb]:
                  CONTINUE}];
    ENDCASE;
  };

  -- Copy the field passed in to the text container specified by clientData.
  -- Since fields may contain information, enumerate the contents of the field
  -- Before doing this, save the current data.container in a local variable, set
  -- data.container to be the container for the field, and then enumerate. Before
  -- exiting the procedure, restore the original data.container
  FieldProc: DocInterchangeDefs.FieldProc = {
    data: Defs.Data = clientData;
```

```
    newField: DocInterchangeDefs.Field ← DocInterchangeDefs.AppendField[
      to: data.container, fieldProps: fieldProps, fontProps: fontProps]:
    data.container ← [field[h: newField]];
    [] ← DocInterchangeDefs.Enumerate[data.container, @data.enumProcs, data];
    DocInterchangeDefs.ReleaseField[@newField];
    data.container ← [doc[h: data.doc]];
  };

  -- For the 1st new paragraph of the first document to be concatenated, get its
  -- properties only; i.e., don't append a new paragraph. Save these properties
  -- in the data object. The reason is that DID.StartCreation appends a new
  -- paragraph & a PFC automatically to a new doc, so you only want to copy
  -- the props to these default objects.
  FirstNewParagraphProc: DocInterchangeDefs.NewParagraphProc = {
    data: Defs.Data = clientData;
    data.enumProcs.newParagraphProc ← NewParagraphProc;
    data.propsRecord ← paraProps↑;
<<
    IF paraProps.tabStops # NIL THEN {
      newTabs: Defs.TabSeqPtr ← zone.NEW[TabSeq[paraProps.tabStops.LENGTH]];
      FOR i: CARDINAL IN [0..paraProps.tabStops.LENGTH) DO
        newTabs[i] ← paraProps.tabStops[i];
        ENDLOOP;
      data.tabStopPtr ← @newTabs}:
>>
  };

  -- Start doc creation using the saved paragraph properties and the PFC props
  -- passed into this proc. Set the PFC enumProc to PFCProc for any remaining
  -- PFC characters encountered and free any tabstop data that you allocated.
  FirstPFCProc: DocInterchangeDefs.PFCProc = {
    startStatus: DocInterchangeDefs.StartCreationStatus:
    data: Defs.Data = clientData:
    data.enumProcs.pfcProc ← PFCProc;
    [doc: data.doc, status: startStatus] ← DocInterchangeDefs.StartCreation[
      paginateOption: simple,
      initialFontProps: fontProps,
      initialParaProps: @data.propsRecord,
      initialPageProps: pageProps];
    data.container ← [doc[h: data.doc]];
--  IF data.tabStopPtr # NIL THEN Defs.z.FREE[@data.tabStopPtr]:
    IF startStatus # ok THEN DocInterchangeDefs.Close[@data.doc]
  };

  -- Initialize the Enumeration call-back procs in the data object
  -- This proc is called when a file or folder is dropped on concat
  InitEnumProcs: PUBLIC PROC[data: Defs.Data] = {
    data.enumProcs ← [
      anchoredFrameProc: AnchoredFrameProc,
      columnBreakProc: ColumnBreakProc,
      fieldProc: FieldProc,
      newParagraphProc: FirstNewParagraphProc,
      pageBreakProc: PageBreakProc,
      pfcProc: FirstPFCProc,
      textProc: TextProc];
  };

  -- Append a new paragraph to the target doc.
  NewParagraphProc: DocInterchangeDefs.NewParagraphProc = {
    data: Defs.Data = clientData;
    DocInterchangeDefs.AppendNewParagraph[[doc[h: data.doc]], paraProps, fontProps];
  };

  PageBreakProc: DocInterchangeDefs.PageBreakProc = {
    data: Defs.Data ← LOOPHOLE[clientData];
    WITH z: data.container SELECT FROM
      doc => [] ← DocInterchangeDefs.AppendPageBreak[
               to: z.h, fontProps: fontProps
               ! DocInterchangeDefs.Error => {
                 rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.appendError.ORD]:
                 Attention.Post[@rb];
                 CONTINUE}];
    ENDCASE;
  };

  -- Append the PFC to the container specified by the clientData argument
  -- * Can't default initialPageProps in StartCreation if wantHeadingHandles or
  --   wantFootingHandles is TRUE. Steve Bartlett submitted AR 8337 for this bug.
  -- * Can't enumerate headings or footings due to error in DocInterchangeDefs
  --   procedures EnumerateHeadFooting & EnumerateFlow. Steve Bartlett submitted
  --   AR 8428 for this bug.
  PFCProc: DocInterchangeDefs.PFCProc = {
    data: Defs.Data = clientData;
    [] ← DocInterchangeDefs.AppendPFC[to: data.doc  pageProps: pageProps. fontProps: fontProps]:
  };

  -- Append the text passed in to the new text container
  TextProc: DocInterchangeDefs.TextProc = {
    data: Defs.Data ← LOOPHOLE[clientData]:
    DocInterchangeDefs.AppendText[
      to:data.container, text:text, textEndContext:textEndContext.
      fontProps:fontProps ! DocInterchangeDefs.Error => {
        rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey appendError.ORD]
        Attention.Post[@rb];
        CONTINUE}];
  };
```

Lab Exercise: TableInterchangeDefs

FMConverter

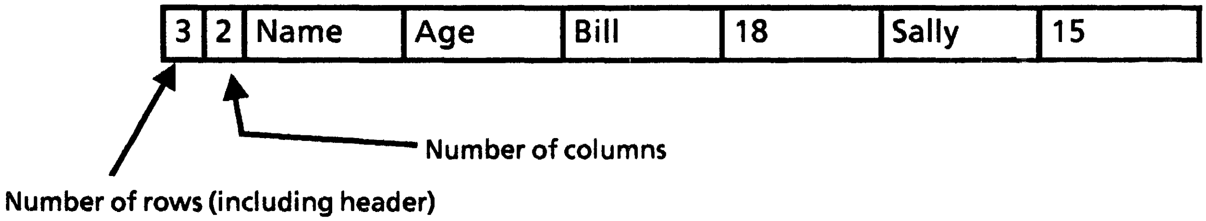
Explanation of Application: The FM Converter converts ViewPoint tables to "Static" tables and vice-versa.

User Interface: Select a VP document and drop it on the FM Converter. For each table in the document, the FM Converter will produce a separate Static file. The converter will ignore all other objects in the document. To convert back to a document from a Static file, select the Static file and drop it on the FM Converter. A new document containing the table will be produced and placed on the desktop. The Static file produced is read-only.

Assignment: In this assignment, you will write the code for a table converter. One table format will be the ViewPoint document table, and the other will be a special table format called "Static." Static is a simple table implementation that has a small subset of the VP table properties.

The Static icon is a separate application that implements its own file type. When you open a Static icon, the application draws the table represented by the data stored in the file. The data consists of the number of rows (including the header row), the number of columns, and fixed-sized blocks of text that represent table entries.

Name	Age
Bill	18
Sally	15



3	2	Name	Age	Bill	18	Sally	15

A Static table & the data file representing it

When converting from a VP table to a Static table, the FM Converter uses DocInterchangeDefs to enumerate the document in its quest for tables. For each one, it uses TableInterchangeDefs to take the table information, create a Static file, and store the table information in that file.

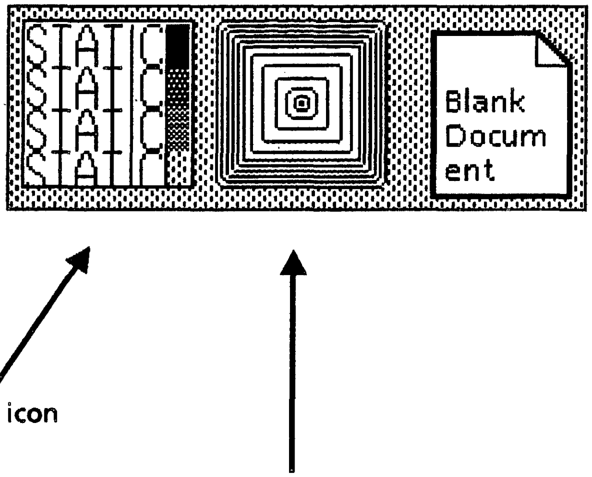
When converting from a Static table to a VP table, the FM Converter uses DocInterchangeDefs to create a new document, and then uses TableInterchangeDefs to create a VP table using the information from the Static file.

Assignment:

Part 1: You have to implement procedures in **FMConverterImplATemplate** that deal with enumerating the VP table information and storing it into the Static file. The procedures are *ConvertToFM, MyColumnProc, MyRowProc, MyTableProc*, and *TableProc*. The procedures *CreateDesktopFile*, *WriteReader*, and *WriteRowCol* will be handy for handling I/O and file creation.

Part 2: You have to implement procedures in **FMConverterImplBTemplate** that deal with extracting the table information from the Static file and creating a VP table in a new document; this new table should have the same content as the Static table. The procedures to implement are *ConvertToDoc* and *SetProperties*. You will find the I/O procedures *AttachStream, GetCols, GetRows* and *GetReaderBody helpful.*

Modules: FMConverter.config
FMConverterDefs.mesa
FMConverterImpl.mesa
FMConverterMsgImpl.mesa
FMConverterTableImplATemplate.mesa
FMConverterTableImplBTemplate.mesa
Static



Static file icon

FM Converter

Lab Solution: TableInterchangeDefs

FM Converter

```
-- FMConverterTableImplA.mesa
-- Copyright (C) Xerox Corporation 1986. All rights reserved.

DIRECTORY
  DocInterchangeDefs USING [
    AnchoredFrameProc, Close, Doc, Enumerate, EnumProcsRecord, Open, TextProc],
  Environment USING [Block, Byte],
  FMConverterDefs USING [FMFileType, GetMessage, MessageKey, TextSize, z],
  NSFile USING [Attribute, Create, Error, GetReference, Handle, nullHandle, OpenByReference],
  NSFileStream USING [Create],
  StarDesktop USING [AddReferenceToDesktop, GetCurrentDesktopFile],
  Stream USING [Block, Delete, Handle, PutBlock, PutWord],
  TableInterchangeDefs USING [ColumnsProc, EnumerateTable, EnumProcsRec, RowProc, TableProc],
  TextInterchangeDefs USING [EnumerateText, Text, TextEnumProcsRecord],
  XChar USING [Character],
  XCharSet0 USING [Make],
  XString USING [AppendChar, AppendReader, InsufficientRoom, NSStringFromReader,
    Reader, ReaderBody, WriterBody, WriterBodyFromBlock];

FMConverterTableImplA: PROGRAM
  IMPORTS DocInterchangeDefs, FMConverterDefs, NSFile, NSFileStream, StarDesktop, Stream,
TableInterchangeDefs, TextInterchangeDefs, XCharSet0, XString
  EXPORTS FMConverterDefs = {

  -- Call-back procs for table enumeration. They will be called in the following order:
  -- MyTableProc, MyColumnProc, & MyRowProc once for each row of the table.
  tableProcs: TableInterchangeDefs.EnumProcsRec ← [
    tableProc: MyTableProc, columnsProc: MyColumnProc, rowProc: MyRowProc];

  -- Enumerate through the document specified by handle. For each
  -- table encountered, invoke the call-back procedure TableProc.
  ConvertToFM: PUBLIC PROC [handle: NSFile.Handle] = {
    docProcs: DocInterchangeDefs.EnumProcsRecord ← [anchoredFrameProc: TableProc];
    doc: DocInterchangeDefs.Doc ← DocInterchangeDefs.Open[
      docFileRef: NSFile.GetReference[handle]].doc;
    [] ← DocInterchangeDefs.Enumerate[textContainer: [doc[h: doc]], procs: @docProcs];
    DocInterchangeDefs.Close[@doc];
  };
```

```
-- Create Static file of size "byteSize" and place it on the desktop.
-- You can attach a stream to this file in order to write it
CreateDesktopFile: PROC[byteSize: LONG CARDINAL]
  RETURNS[handle: NSFile.Handle ← NSFile.nullHandle] = {
  fileName: XString.ReaderBody ← FMConverterDefs.GetMessage[
    FMConverterDefs.MessageKey.fmStaticName.ORD];
  desktopHandle: NSFile.Handle ← NSFile.OpenByReference[StarDesktop.GetCurrentDesktopFile[]];
  attributes: ARRAY [0..3) OF NSFile.Attribute ← [
    [name[XString.NSStringFromReader[r: @fileName, z: FMConverterDefs.z]]],
    [sizeInBytes[byteSize]],
    [type[FMConverterDefs.FMFileType]]];
  handle ← NSFile.Create[directory: desktopHandle, attributes: DESCRIPTOR[attributes]];
  StarDesktop.AddReferenceToDesktop[reference: NSFile.GetReference[handle], place:[0,0]];
  };

-- Use this procedure to obtain the header information. When it is called,
-- each header should be appended to the stream passed in as clientData.
MyColumnProc: TableInterchangeDefs.ColumnsProc = {
  stream: LONG POINTER TO Stream.Handle ← LOOPHOLE[clientData];
  text: TextInterchangeDefs.Text;
  enumProc: TextInterchangeDefs.TextEnumProcsRecord ← [textProc: TextEnumProc];
  FOR i: CARDINAL IN [0..columns.length) DO
    -- extract the text from the specified column by calling the supplied
    -- proc and then enumerating the text object passed back.
    WITH x: columns[i].headerEntryRec.content SELECT FROM
      read => {text ← x.obtainTextProc[x.obtainTextData];
               [] ← TextInterchangeDefs.EnumerateText[
                 text: text, procs: @enumProc, clientData: stream]};
      ENDCASE;
    ENDLOOP;
  };

-- This proc is called once for each row of the table. The content of each
-- element should be appended to the stream passed in as clientData.
MyRowProc: TableInterchangeDefs.RowProc = {
  stream: LONG POINTER TO Stream.Handle ← LOOPHOLE[clientData];
  text: TextInterchangeDefs.Text;
  enumProc: TextInterchangeDefs.TextEnumProcsRecord ← [textProc: TextEnumProc];
  FOR i: CARDINAL IN [0..content.length) DO
    -- extract the text from the specified row by calling the supplied proc
    -- and then enumerating the text object passed back.
    WITH x: content[i].content SELECT FROM
      read => {text ← x.obtainTextProc[x.obtainTextData];
               [] ← TextInterchangeDefs.EnumerateText[
                 text: text, procs: @enumProc, clientData: stream]};
      ENDCASE;
    ENDLOOP;
  };
```

```
-- make private copy of text and pass the text back by assigning it to
-- the reader body that clientdata points to
TextEnumProc: DocInterchangeDefs.TextProc = {
  stream: LONG POINTER TO Stream.Handle ← clientData;
  WriteReader[text, stream↑];  };

-- The number of rows and columns are passed into this proc. Use this information to allocate a file
-- of the proper size where sizeInBytes = ((numCols * (numRows + 1) * text length) + 4.
-- Write the number of rows and columns to the stream. The FM application treats headers the same
-- as any other row; this explains the + 1 in the calculation. The + 4 reflects the  number of
-- bytes required to store the 2 CARDINALs row and col.
-- clientData is a LONG POINTER TO Stream.Handle and it is this stream that we attach to the newly
-- created file. The other call-back procs will then write information to the same stream.
MyTableProc: TableInterchangeDefs.TableProc = {
  stream: LONG POINTER TO Stream.Handle ← LOOPHOLE[clientData];
  fileSize: LONG CARDINAL ← props.numberOfColumns *
    (props.numberOfRows + 1) * FMConverterDefs.TextSize + 4;
  handle: NSFile.Handle ← CreateDesktopFile[fileSize];
  stream↑ ← NSFileStream.Create[handle];
  WriteRowCol[rows: props.numberOfRows + 1, cols: props.numberOfColumns, stream: stream↑];
  };

-- This procedure is called once for each table encounted in the doc. It should enumerate through
-- the table and pass a pointer to a NIL  stream as the clientData argument handle. The call-back
-- procs can then attach this stream to a file and use it to append information.
-- Delete the stream before returning from this procedure.
TableProc: DocInterchangeDefs.AnchoredFrameProc = {
  stream: Stream.Handle ← NIL;
  IF type # table THEN RETURN[];
  TableInterchangeDefs.EnumerateTable[table: content, procs: @tableProcs, clientData: @stream];
  Stream.Delete[stream! NSFile.Error => CONTINUE];
  };

-- Pad or truncate the reader so that it is Defs.TextSize bytes in length.
-- Write these characters to the Static file.
WriteReader: PROC[r: XString.Reader, stream: Stream.Handle] = {
  buffer: PACKED ARRAY [0..FMConverterDefs.TextSize) OF Environment.Byte;
  block: Environment.Block ← [@buffer, 0, FMConverterDefs.TextSize];
  wb: XString.WriterBody ← XString.WriterBodyFromBlock[block: block, inUse: 0];
  space: XChar.Character ← XCharSet0.Make[code: space];
  XString.AppendReader[to: @wb, from: r ! XString.InsufficientRoom => RESUME];
  DO            -- blank out rest of writer
    XString.AppendChar[to: @wb, c: space ! XString.InsufficientRoom => EXIT];
    ENDLOOP;
  Stream.PutBlock[sH: stream, block: block];
  };

-- Write the number of rows and columns to the Static file.
WriteRowCol: PROC[rows, cols: NATURAL, stream: Stream.Handle] = {
  Stream.PutWord [sH: stream,  word: rows];
  Stream.PutWord [sH: stream,  word: cols];
  };

}...
```

```
-- FMConverterTableImplB.mesa
-- Copyright (C) Xerox Corporation 1986. All rights reserved.

DIRECTORY
  DocInterchangeDefs USING [AppendAnchoredFrame, CheckAbortProc, Doc, FinishCreation, StartCreation],
  DocInterchangePropsDefs USING [FramePropsRecord],
  FMConverterDefs USING [TextSize, z],
  NSFile USING [Close, GetReference, Handle, Move, OpenByReference],
  NSFileStream USING [Create],
  StarDesktop USING [AddReferenceToDesktop, GetCurrentDesktopFile],
  Stream USING [GetWord, Handle],
  TableInterchangeDefs USING [AppendRow, ColumnInfo, ColumnInfoSeq, FillInTextProc, FinishTable,
    Handle, StartTable, TablePropsRec],
  TextInterchangeDefs USING [AppendTextToText],
  XString USING [
    AppendStream, NewWriterBody, Reader, ReaderBody, ReaderFromWriter, unknownContext, WriterBody];

FMConverterTableImplB: PROGRAM
  IMPORTS DocInterchangeDefs, FMConverterDefs, NSFile, NSFileStream, StarDesktop, Stream,
TableInterchangeDefs, TextInterchangeDefs, XString EXPORTS FMConverterDefs = {
  OPEN Defs: FMConverterDefs, DI: DocInterchangeDefs, TI: TableInterchangeDefs;

  -- Table CONSTANTS and TYPEs
  marginInPixels: CARDINAL = 35;
  micasPerPixel: CARDINAL = 9;
  tableWidth: CARDINAL = 3200;    -- micas
  tableHeight: CARDINAL = 3200;   -- micas
  rowMargin: CARDINAL = 100;  -- micas
  headerMargin: CARDINAL = marginInPixels * micasPerPixel;
  RbArray: TYPE = RECORD[a: SEQUENCE max: CARDINAL OF XString.ReaderBody];
  RbArrayPtr: TYPE = LONG POINTER TO RbArray;

  -- Open a stream on the specified file.
  AttachStream: PROC[file: NSFile.Handle] RETURNS[stream: Stream.Handle] = {
    stream ← NSFileStream.Create[file];
    };
```

```
-- Create a document with DocInterchangeDefs. Allocate the sequences that'll hold the information.
-- Read the # rows & columns from the data file to determine the size of the table. Set up the table
-- and table frame properties. Read the header data from the data file. Use TableInterchangeDefs to
-- start the creation of a table. Read the data  row by row into the table. Append the table
-- (anchored frame) to the document. Finish the document and place it on the desktop. Release
-- the handles to the new document and the data file.
ConvertToDoc: PUBLIC PROC [handle: NSFile.Handle] = {
  tableHandle: TI.Handle;
  docFile: NSFile.Handle;   -- the file to be created
  doc: DocInterchangeDefs.Doc ← DocInterchangeDefs.StartCreation[].doc;
  tableProps: TI.TablePropsRec;
  docFrameProps: DocInterchangePropsDefs.FramePropsRecord;
  -- Allocate seqs for column, header, and row info
  stream: Stream.Handle ← AttachStream[handle];
  rows: NATURAL ← GetRows[stream] - 1;  -- the headers counted as a row
  cols: NATURAL← GetCols[stream];
  columnInfo: TI.ColumnInfo ← Defs.z.NEW[TI.ColumnInfoSeq[cols]];
  headerRBs: RbArrayPtr ← Defs.z.NEW[RbArray[cols]];
  rowRBs: RbArrayPtr ← Defs.z.NEW[RbArray[cols]];
  [tableProps,docFrameProps] ← SetProperties[rows, cols, tableWidth, tableHeight];
  -- Fill in seq of column info
  FOR i: CARDINAL IN [0..cols) DO
    headerRBs[i] ← GetReaderBody[stream, Defs.z];
    columnInfo[i] ← [  -- all values except content and width taken from the null constant
      headerEntryRec: [
        subHeaders: NIL, line: [solid, w2], singleLineHint: FALSE, spare1: 0,
        content: [write[fillInTextProc: WriteText, clientData: @headerRBs[i]]]],
      name: NIL,
      description: NIL,
      divided: FALSE,
      subcolumns: 0,
      repeating: FALSE,
      subcolumnInfo: NIL,
      alignment: center,
      tabOffset: 0,
      width: tableWidth,
      leftMargin: 0,
      rightMargin: 0,
      type: any,
      required: FALSE,
      language: USEnglish,
      format: NIL,
      stopOnSkip: FALSE,
      range: NIL,
      length: 0,
      skipText: NIL,
      skipChoice: empty,
      fillin: NIL,
      fillinRuns: NIL,
      line: [solid, w2],
      sortKeys: NIL,
      spare1: 0];
    ENDLOOP;
  -- Create a table w/ appropriate props & column info
  tableHandle ← TI.StartTable[doc: doc, props: @tableProps, c: columnInfo];
  tableHandle.rc.topMargin ← tableHandle.rc.bottomMargin ← rowMargin;
  -- Read table entries from file  & store'em in row sequence
  THROUGH [0..rows) DO
    FOR j: CARDINAL IN [0..cols) DO
      rowRBs[j] ← GetReaderBody[stream, Defs.z];
      tableHandle.rc[j] ← [
        subRows: NIL,
        singleLineHint: FALSE,
        spare1: 0,
        content: [write[fillInTextProc: WriteText, clientData: @rowRBs[j]]]];
      ENDLOOP;
    TI.AppendRow[h: tableHandle, rc: tableHandle.rc];
    ENDLOOP;
  -- Finish table, Append table to document
  [] ← DocInterchangeDefs.AppendAnchoredFrame[
     to: doc, type: table, anchoredFrameProps: @docFrameProps,
     content: TI.FinishTable[tableHandle].table];
  -- Finish document and place it on the desktop
  [docFile,,] ← DocInterchangeDefs.FinishCreation[docPtr: @doc, checkAbortProc: DummyAbortProc];
  NSFile.Move[file: docFile, destination: NSFile.OpenByReference[
    StarDesktop.GetCurrentDesktopFile[]]];
  StarDesktop.AddReferenceToDesktop[reference: NSFile.GetReference[docFile], place: [0,0]];
  NSFile.Close[docFile];
  };

-- need dummy to avoid control fault, found 3/10/87
DummyAbortProc: DocInterchangeDefs.CheckAbortProc = {RETURN[FALSE]};

-- Help procedure to get the row and column info from the Static file
GetCols: PROC[stream: Stream.Handle] RETURNS[cols: NATURAL] = {
  cols ← LOOPHOLE[Stream.GetWord [sH: stream]];
  };

-- Help procedure to get the row and column info from the Static file
GetRows: PROC[stream: Stream.Handle] RETURNS[rows: NATURAL] = {
  rows ← LOOPHOLE[Stream.GetWord [sH: stream]];
  };

-- Retrieve Defs.TextSize bytes from the Static file and return the ReaderBody generated from them.
GetReaderBody: PROC[stream: Stream.Handle, zone: UNCOUNTED ZONE] RETURNS[rb: XString.ReaderBody] = {
  wb: XString.WriterBody ← XString.NewWriterBody[maxLength: FMConverterDefs.TextSize, z: zone];
  [] ← XString.AppendStream[to: @wb, from: stream, nBytes: FMConverterDefs.TextSize];
  rb ← XString.ReaderFromWriter[w: @wb]↑;
  };
```

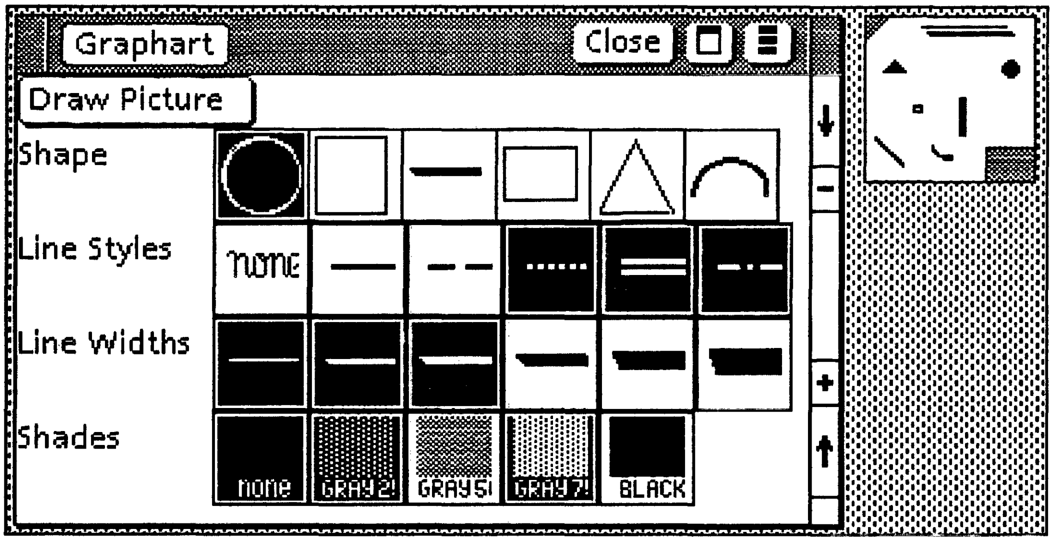
```
-- Set the properties for the table and for the frame containing the table
-- dim1 & dim2 are the parameters for frameDims in docFrameProps.
SetProperties: PROC[rows, cols, dim1, dim2: CARDINAL]
  RETURNS[tableProps: TI.TablePropsRec, docFrameProps: DocInterchangePropsDefs.FramePropsRecord] = {
  tableProps ← [   -- Set table props
    name: NIL,
    fillinByRow: TRUE,
    fixedRows: FALSE,
    fixedColumns: TRUE,
    numberOfColumns: cols,
    numberOfRows: rows,
    visibleHeader: TRUE,
    repeatHeader: TRUE,
    repeatTopCaption: TRUE,
    repeatBottomCaption: TRUE,
    borderLine: [none, w1],
    dividerLine: [solid, w4],
    horizontalAlignment: center,
    headerVerticalAlignment: centered,
    topHeaderMargin: headerMargin,
    bottomHeaderMargin: headerMargin,
    sortKeys: NIL,
    spare1: 0];

  docFrameProps ← [    -- Set table frame props
    borderStyle: solid,
    borderThickness: 2,
    frameDims: [dim1, dim2],
    fixedWidth: FALSE,
    fixedHeight: FALSE,
    span: fullColumn,
    verticalAlignment: floating,
    horizontalAlignment: centered,
    topMarginHeight: 0,
    bottomMarginHeight: 0,
    leftMarginWidth: 0,
    rightMarginWidth: 0,
    spare1: 0];
  };

-- call-back proc used to append text to table
WriteText: TableInterchangeDefs.FillInTextProc = {
  r: XString.Reader ← clientData;
  TextInterchangeDefs.AppendTextToText[to: text, text: r, textEndContext: XString.unknownContext];
  };

}...
```

Lab Exercise: GraphicsInterchangeDefs

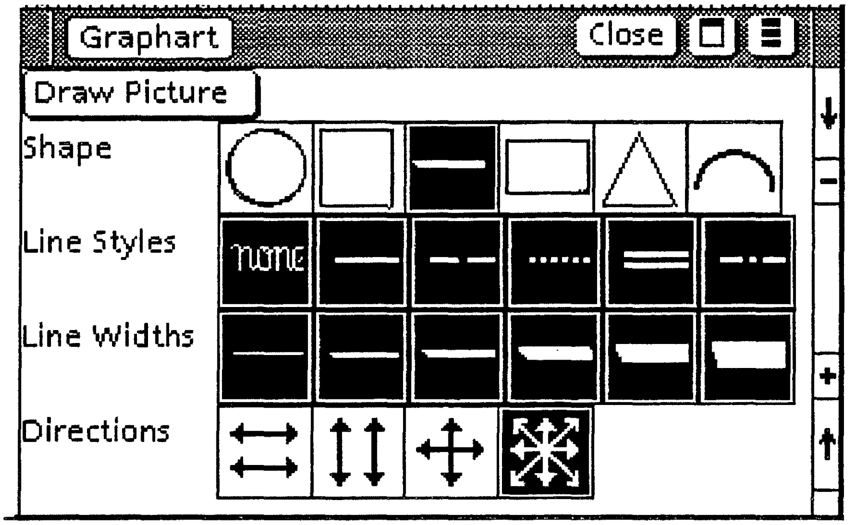
Graphart

Explanation of Application: Graphart generates a document with a graphics frame that contains randomly placed objects with user specified properties. The application takes a selected document as input and enumerates each object in every graphics frame that the document contains. Graphart then modifies the properties of the object by choosing a random size and location (orientation and length for lines and curves) and by randomly selecting between the property choices specified in the tool. Defaults when no values are specified are: **Line Style** - solid, **Line Widths** - one pixel, **Shades** - none, **Directions** (for lines only) - any. The object with its new set of properties is then appended to the destination document. This enumeration continues until all objects in the source document have been processed. The newly created document is then placed on the desktop for your viewing pleasure.

User Interface: The Graphart icon must first be opened as depicted below. The user then selects the desired shape for which to set a range of properties (the entire range of properties is selected by default). When Graphart encounters an object of the same type in the input document, it will randomly select between the values that the user specified. When the line shape is specified, the **Shades** choices will be replaced by a set of possible directions that the user can choose. After setting the desired properties, the user selects the desired input document and invokes the **Draw Picture** command. Graphart produces a blank document containing the randomized output from all matching shapes found in ANY Graphics frame within the input document. For example, in the figure below, Graphart would set the properties of ALL ellipses encountered to: a border with a dotted, double, or dash-dot; a line width of from 1 to 3 pixels; and 50% - 100% black shading. The other shapes might have different ranges of random values, depending on what changes the user made to their properties.



The Graphart tool (as it would appear with a circle selected) & icon



The Graphart tool with the line object selected.

Assignment: In this assignment, you will implement several procedures that enumerate and append graphics objects. The procedures are as follows: *CreateNewDocument*, *EnumerateObject*, *FinishNewDocument*, *GraphicProc*, and *MakeCurve*. Each of these procedures is located in the module called **GraphartDrawImplTemplate**; additional comments are located in this module.

Modules: Graphart.config
GraphartBitmapImpl.mesa
GraphartDefs.mesa
GraphartDrawImplTemplate.mesa
GraphartFormImpl.mesa
GraphartImpl.mesa
GraphartMsgImpl.mesa

Hints: *CreateNewDocument* generates a destination document and obtains its graphics handle; this handle is stored in the context for use by other procedures. *EnumerateObject* is called when a cluster or internal graphics frame is encountered; it must enumerate this new container using the same call-back procedures as the original enumeration. *FinishNewDocument* is invoked when the enumeration is complete; it must append a graphics frame to the destination document, close the source document, finish the destination document, and place it on the desktop. *GraphicProc* is called for each anchored frame in the source document and must enumerate any graphics frames that it encounters (the call-back procs can be found by searching for their types). *MakeCurve* is called when a curve is encountered in the source document. *MakeCurve* generates new properties and adds a curve to the destination graphics handle; these new properties can be generated by calling the appropriate *Get** procedure.

Lab Solution: GraphicsInterchangeDefs

Graphart

```
-- Copyright (C) Xerox Corporation 1986. All rights reserved.

-- This module implements the Draw command using procs from GraphicsInterchangeDefs. All the Make*
-- procedures here are EnumProcs for GI.Enumerate. All the Get*Props procedures generate random
-- properties for the objects. SetUpArrays creates arrays from the choices made in the tool;
-- it implements the random generation of properties.

-- Create a new document and start a graphics frame.
-- The handle to the graphics frame will be stored in the data object.
CreateNewDocument: PROC[data: Defs.Data] = {
  data.newDoc ← DocInterchangeDefs.StartCreation[].doc;
  data.graphicsHandle ← GI.StartGraphics[doc: data.newDoc];
  };

-- Enumerate the graphics container passed in as if it were simply another
-- graphics frame. clientData points to the context.
EnumerateObject: PROC[graphicsContainer: DocInterchangeDefs.Instance, clientData: LONG POINTER] = {
  enumProcs: GI.EnumProcsRecord ← [
    clusterProc: EnumerateCluster,
    curveProc: MakeCurve,
    ellipseProc: MakeEllipse,
    frameProc: EnumerateFrame,
    lineProc: MakeLine,
    rectangleProc: MakeRectangle,
    triangleProc: MakeTriangle];
  data: Defs.Data ← NARROW[clientData, Defs.Data];
  [] ← GI.Enumerate[
    doc: data.oldDoc,
    graphicsContainer: graphicsContainer,
    procs: @enumProcs,
    clientData: data];
  };

-- Generate a random curve based on the user's data values. Call GetRandomBox to get the location
-- and size of the new curve in the container specified by data.graphicsHandle. The procedure
-- GetCurveProps should be used to get random curve properties.
MakeCurve: GI.CurveProc = {
  data: Defs.Data ← NARROW[clientData, Defs.Data];
  randomBox: GI.Box ← GetRandomBox[data];
  randomCurveProps: GI.CurveProps ← GetCurveProps[data, curveProps, randomBox];
  GI.AddCurve[
    h: data.graphicsHandle,
    box: randomBox,
    curveProps: LOOPHOLE[randomCurveProps, GI.ReadonlyCurveProps]];
};
```

```
-- Finish the graphics frame and append its content to the  new document. Close the old document,
-- finish the new one, and add the new document to the desktop.
FinishNewDocument: PROC[data: Defs.Data] = {
  newFile: NSFile.Handle ← NSFile.nullHandle;
  desktop: NSFile.Handle ← NSFile.OpenByReference[
    StarDesktop.GetCurrentDesktopFile[]];
  graphics: DocInterchangeDefs.Instance ←
    GI.FinishGraphics[data.graphicsHandle];
  data.frameProps.frameDims.w ← data.frameProps.frameDims.w / micasPerPixel;
  data.frameProps.frameDims.h ← data.frameProps.frameDims.h / micasPerPixel;
  [] ← DocInterchangeDefs.AppendAnchoredFrame[
    to: data.newDoc,
    type: graphics,
    anchoredFrameProps: @data.frameProps,
    content: graphics];
  DocInterchangeDefs.Close[docPtr: @data.oldDoc];
  [newFile,] ← DocInterchangeDefs.FinishCreation[
    docPtr: @data.newDoc, checkAbortProc: DummyAbortProc];
  NSFile.Move[file:newFile, destination: desktop];
  StarDesktop.AddReferenceToDesktop[reference:NSFile.GetReference[newFile], place:[0,0]];
  NSFile.Close[newFile];
  };

DummyAbortProc: DocInterchangeDefs.CheckAbortProc = {RETURN[FALSE]};

-- Enumerate through the objects in the source graphics frame and call procedures to add
-- corresponding objects to the new frame. The dims of the graphics frame are converted to micas
-- and saved in data since all dimensions used later are in these units.
GraphicProc: DocInterchangeDefs.AnchoredFrameProc = {
  enumProcs: GI.EnumProcsRecord ← [
    clusterProc: EnumerateCluster,
    curveProc: MakeCurve,
    ellipseProc: MakeEllipse,
    frameProc: EnumerateFrame,
    lineProc: MakeLine,
    rectangleProc: MakeRectangle,
    triangleProc: MakeTriangle];
  data: Defs.Data ← NARROW[clientData, Defs.Data];
  IF type # graphics THEN RETURN;
  data.frameProps ← anchoredFrameProps↑;
  data.frameProps.frameDims.w ← anchoredFrameProps.frameDims.w * micasPerPixel;
  data.frameProps.frameDims.h ← anchoredFrameProps.frameDims.h * micasPerPixel;
  [] ← GI.Enumerate[
    doc: data.oldDoc,
    graphicsContainer: anchoredFrame,
    procs: @enumProcs,
    clientData: data];
  };
```

Lab Exercise: Performance Issues

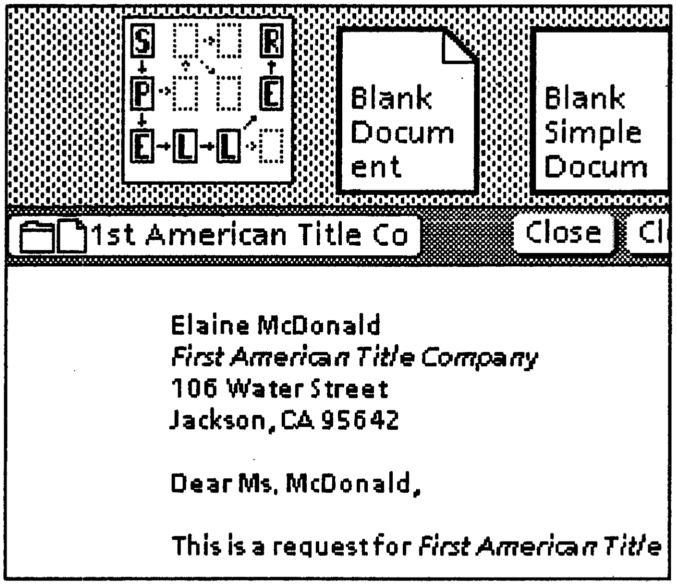
Speller

Explanation of Application: Speller is a spell-checking tool that allows the user to create a database of legal words and then check text against that database for misspelled words. Speller takes input text and stores the words in a database. Input text can be a selected document, a selected simple document, selected text from anywhere, or the text from the text item within the tool itself. The user can then specify text to be spell-checked by the tool. Of course, the user may also delete words from the database.

User Interface: The Speller icon must first be opened. The user specifies text, either to be inserted into the database or to be spell-checked, by either making a selection or by entering text into the *Word* field of the tool. The tool will always attempt to convert the current selection before reading the *Word* field of the tool.

If the user has the *Check Spelling* boolean set to true, the *Read* command will treat the input text as text to be spell-checked; conversely, if *Check Spelling* is false, the *Read* command will treat the input text as words to be inserted into the database. *Delete* interprets the input as words to be deleted from the database. *List* differs from the other commands in that it only takes the first "word" of the input (as defined by **XToken)** as the pattern to find all words in the database that start with that pattern.

Legal words contain only alphabetic characters (a through z, A through Z). All upper case characters are converted to lower case before they're stored in the database.

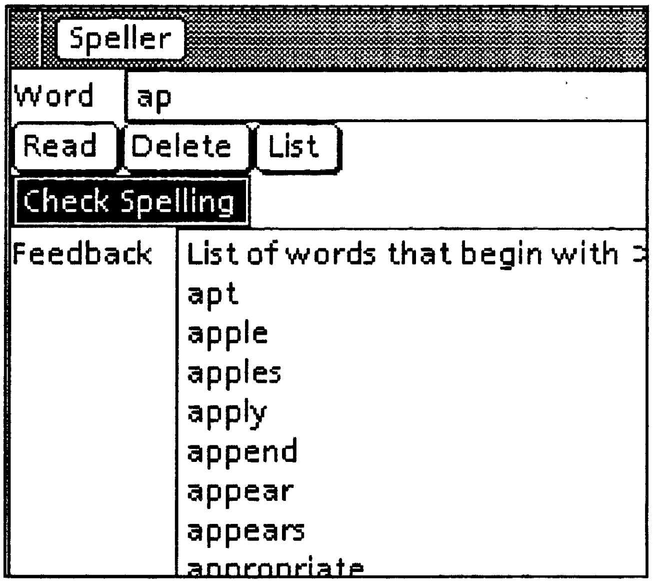


Elaine McDonald
First American Title Company
106 Water Street
Jackson, CA 95642

Dear Ms. McDonald,

This is a request for *First American Title*

The Speller icon & some possible forms of input (a document, simple document, text). Input must be the current selection or the text in the *Word* field of the Speller tool.

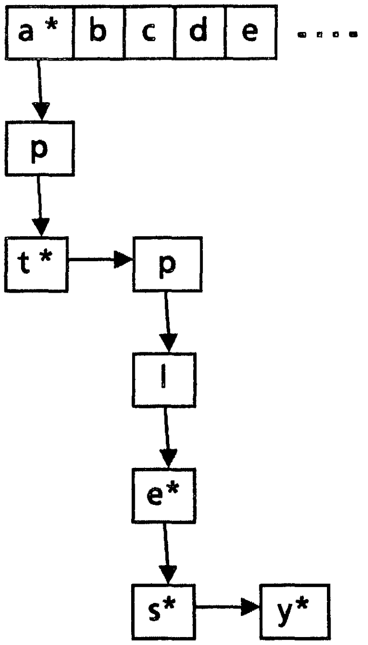


Word ap

Feedback List of words that begin with :
apt
apple
apples
apply
append
appear
appears
appropriate

The Speller tool after the *List* command was invoked.

Data Structure: The data structure implements a trie (rhymes with "fry"), which is a method of storing words by sharing letters in an orderly fashion. This particular trie consists of an array of root elements (a to z) where each root element may point to a *logical* subtree of letters (the subtrees are actually implemented in a large array). See the following illustration.



In this trie, the words a, apt, apple, apples, & apply are stored. Child letters are indicated with a downward arrow, while sibling letters are indicated with a sideways arrow. Keep in mind that this is only a logical structure; in the actual structure, the subtree under root element a would be implemented in an array.

The asterisk indicates that this letter is the end of a logical word.

Assignment: In this assignment, you will write code to help improve the performance of the Speller tool. Specifically, you will make the Read, Delete, and List commands run in the background, thereby freeing the Notifier for other processes. Use the **BackgroundProcess** interface when forking a process. In addition, you should modify the EnumerateDocument procedure so that the document is enumerated in a separate **NSFile.Session**.

You must not allow the user to run two commands at the same time within the same instance of the Speller tool. For example, you can't do a *Delete* while a *List* is executing. To facilitate this protection, create a boolean called **busy** and store it in the context. When one of the tool's commands is invoked, set the boolean to TRUE so that none of the other commands may be executed until that command finishes.

Also, you must not let the user close the tool while one of the background processes is executing. Therefore, you must write a **StarWindowShell.IsCloseLegalProc** for the tool.

You'll need to modify the procedures *ReadData, ListData, EnumerateDocument,* and *DeleteData* in **SpellerFormImplTemplate**. In addition, you'll have to change **SpellerDefs** to add the boolean to the context.

Modules: Speller.config,
SpellerDefs.mesa,
SpellerFormImplTemplate.mesa,
SpellerImplTemplate.mesa,
SpellerMsgImpl.mesa,
SpellerVMImpl.mesa.

Lab Solution: Performance Issues

Speller

If the **busy** field has not already been added to **SpellerDefs.DataObject**, then add it as the last field: **busy: BOOL ← FALSE**

From SpellerImpl:

```
-- MONITOR PROCs - The monitor invariant is data.busy

-- ensure that process is finished before destroying context data
IsCloseLegal: ENTRY StarWindowShell.IsCloseLegalProc = {
  body: Window.Handle ← StarWindowShell.GetBody[sws];
  data: Defs.Data ← GetContext[body];
  IF data.busy THEN RETURN[FALSE];
  RETURN[TRUE] };
```

From SpellerFormImpl:

```
-- MONITOR ENTRY PROCEDURES
-- data.busy is the monitor invariant. Refuse requests if another process is in progress.

-- Delete the word specified in the form. If the word in 0 chars long post a msg.
-- If the word is found post success message; otherwise, post not found.
DeleteData: ENTRY FormWindow.CommandProc = {
  data: Defs.Data ← Defs.GetContext[window];
  IF data.busy THEN RETURN;
  data.busy ← TRUE;
  Process.Detach[FORK DoBackgroundDelete[window]] };

-- Called when user wishes to list contents of dictionary. The text in the "word" field of the
-- form is used as a filter (eg if text = "to" then all words beginning with "to" would be listed).
ListData: ENTRY FormWindow.CommandProc = {
  data: Defs.Data ← Defs.GetContext[window];
  IF data.busy THEN RETURN;
  data.busy ← TRUE;
  Process.Detach[FORK DoBackgroundList[window]] };

-- Reset the monitor invariant
MakeBusyFalse: ENTRY PROC[data: Defs.Data] = {data.busy ← FALSE };

-- Gets the currently selected document/simple doc/text/ or word and  parses the information into
-- valid words. Each word is passed off to a procedure to see if it exists in the dictionary;
-- if not the word is saved. When all data has been processed, the list of words that were
-- not found is displayed to the user.
ReadData: ENTRY FormWindow.CommandProc = {
  data: Defs.Data ← Defs.GetContext[window];
  IF data.busy THEN RETURN;
  data.busy ← TRUE;
  Process.Detach[FORK DoBackgroundRead[window]] };

-- END OF MONITOR
```

```
DoBackgroundDelete: PROCEDURE[window: Window.Handle] = {
  ENABLE UNWIND => { -- restore monitor invariant
    data: Defs.Data ← Defs.GetContext[window];
    MakeBusyFalse[data]};

  RealDelete: BackgroundProcess.CallBackProc = {
    data: Defs.Data ← Defs.GetContext[window];
    wb: XString.WriterBody ← XString.NewWriterBody[maxLength: 50, z: Defs.z];
    xfo: XFormat.Object ← XFormat.WriterObject[@wb];
    text: XString.ReaderBody ← FormWindow.GetTextItemValue[
      window: window, item: FormItems.word.ORD, zone: Defs.z];
    IF XString.Empty[@text] THEN {
      rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.noWordSpecified.ORD];
      Attention.Post[@rb];
      MakeBusyFalse[data];
      RETURN[failure]};
    -- valid data so parse and delete each word
    BEGIN
      rb: XString.ReaderBody;
      word: XString.ReaderBody;
      tokenHandle: XToken.Handle ← XToken.ReaderToHandle[r: @text];
      DO
        word ← XToken.Filtered[h: tokenHandle,
        data: NIL, filter: XToken.Alphabetic, skip: nonToken];
        IF XString.Empty[@word] THEN EXIT;
        IF Defs.DeleteWord[data, @word] THEN {
          rb ← Defs.GetMessage [Defs.MessageKey.deleted.ORD];
          XFormat.Reader[@xfo, @word];
          XFormat.Reader[@xfo, @rb];
          XFormat.CR[@xfo]}
        ELSE {
          rb ← Defs.GetMessage [Defs.MessageKey.notFound.ORD];
          XFormat.Reader[@xfo, @word];
          XFormat.Reader[@xfo, @rb];
          XFormat.CR[@xfo]};
        [] ← XToken.FreeTokenString[r: @word];
        ENDLOOP;
      rb ← Defs.GetMessage [Defs.MessageKey.deletionComplete.ORD];
      XFormat.Reader[@xfo, @rb];
      FormWindow.SetTextItemValue[
        window: window, item: FormItems.feedback.ORD, newValue: XString.ReaderFromWriter[@wb]];
      [] ← XToken.FreeReaderHandle[h: tokenHandle];
      XString.FreeReaderBytes[@text, Defs.z];
      XString.FreeWriterBytes[@wb];
      END;
    MakeBusyFalse[data];
    RETURN[success];
    }; -- of RealDelete

  name: XString.ReaderBody ← XString.FromSTRING["Delete"];
  Process.SetPriority[Process.priorityBackground];
  [] ← BackgroundProcess.ManageMe[name: @name, callBackProc: RealDelete];
  };
```

```
-- Performs the listing in the background
DoBackgroundList: PROCEDURE[window: Window.Handle] = {
  ENABLE UNWIND => { -- restore monitor invariant
    data: Defs.Data ← Defs.GetContext[window];
    MakeBusyFalse[data]};

  RealList: BackgroundProcess.CallBackProc = {
    wb: XString.WriterBody;
    data: Defs.Data ← Defs.GetContext[window];
    text: XString.ReaderBody ← FormWindow.GetTextItemValue[
      window: window, item: FormItems.word.ORD, zone: Defs.z];
    tokenHandle: XToken.Handle ← XToken.ReaderToHandle[r: @text];
    word: XString.ReaderBody ← XToken.Filtered[
      h: tokenHandle, data: NIL, filter: XToken.Alphabetic, skip: nonToken];
    IF XString.Empty[@word] THEN {
      MakeBusyFalse[data];
      RETURN[failure]};
    wb ← Defs.List[data: data, r: @word];
    FormWindow.SetTextItemValue[
      window: window, item: FormItems.feedback.ORD, newValue: XString.ReaderFromWriter[@wb]];
    XString.FreeWriterBytes[@wb];
    [] ← XToken.FreeTokenString[@word];
    [] ← XToken.FreeReaderHandle[h: tokenHandle];
    MakeBusyFalse[data];
    RETURN[success];
    }; -- of RealList

  name: XString.ReaderBody ← XString.FromSTRING["List"];
  Process.SetPriority[Process.priorityBackground];
  [] ← BackgroundProcess.ManageMe[name: @name, callBackProc: RealList];
  };
```

```
DoBackgroundRead: PROCEDURE[window: Window.Handle] = {
ENABLE UNWIND => { -- restore monitor invariant
  data: Defs.Data ← Defs.GetContext[window];
  MakeBusyFalse[data]};

RealRead: BackgroundProcess.CallBackProc = {
  data: Defs.Data ← Defs.GetContext[window];
  blank: XString.ReaderBody ← XString.FromSTRING[" "L];
  head: NotFoundPtr ← Defs.z.NEW[NotFoundNode ←
    [word: XString.CopyToNewReaderBody[r:@blank, z:Defs.z]]];
  check: BOOLEAN ← FormWindow.GetBooleanItemValue[
    window: window, item: FormItems.checkSpelling.ORD];
  text: XString.ReaderBody;
  finalStatus ← success;
  BEGIN
    ENABLE
    BEGIN
      FileProblem => {MakeBusyFalse[data]; GOTO Exit};
      NoSelection => {
        text ← FormWindow.GetTextItemValue[
          window: window, item: FormItems.word.ORD, zone: Defs.z];
        CONTINUE};
      FileIsDoc => { -- docs are handled separately
        EnumerateDocument[data, check, head, window];
        MakeBusyFalse[data];
        GOTO Exit};
      END;
    text ← GetSelectionAsReader[data];
    END;
  CheckText[data, @text, check, head];
  XString.FreeReaderBytes[@text, Defs.z]; -- free the checked text
  IF check THEN DisplayWordsNotFound[head, window]
  ELSE {
    rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.insertionComplete.ORD];
    FormWindow.SetTextItemValue[window: window,item: FormItems.feedback.ORD,newValue: @rb]};
  MakeBusyFalse[data];
  EXITS Exit => NULL;
  };

name: XString.ReaderBody ← XString.FromSTRING["Read"];
Process.SetPriority[Process.priorityBackground];
[] ← BackgroundProcess.ManageMe[name: @name, callBackProc: RealRead];
};
```

```
-- Open and enumerate the selected doc. Check each word of text to see if it is in the dictionary
EnumerateDocument: PROC[
  data: Defs.Data, check: BOOLEAN, head: NotFoundPtr, window: Window.Handle] = {

  -- check each block of text found in the document
  TextProc: DocInterchangeDefs.TextProc = {CheckText[data, text, check, head]};

  element: Selection.Value ← Selection.Convert[file]; --get reference to file
  ref: LONG POINTER TO NSFile.Reference ← element.value;
  enumProcs: DocInterchangeDefs.EnumProcsRecord ← [textProc:TextProc];
  status: DocInterchangeDefs.OpenStatus;
  doc: DocInterchangeDefs.Doc;
  user: Atom.ATOM ← Atom.MakeAtom["CurrentUser"L];
  prop: Atom.ATOM ← Atom.MakeAtom["IdentityHandle"L];
  identity: Auth.IdentityHandle = Atom.GetProp[user, prop]↑.value;
  session: NSFile.Session ← NSFile.Logon[identity];
  -- open source document and enumerate in a separate NSFile session
  [doc, status] ← DocInterchangeDefs.Open[docFileRef: ref↑, session: session];
  IF status # ok THEN GOTO Exit;
  [] ← DocInterchangeDefs.Enumerate[
    textContainer: [doc[h:doc]], procs: @enumProcs, clientData: head];
  DocInterchangeDefs.Close[docPtr:@doc];
  NSFile.Logoff[session];
  IF check THEN DisplayWordsNotFound[head, window]
  ELSE {
    rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.insertionComplete.ORD];
    FormWindow.SetTextItemValue[
      window: window, item: FormItems.feedback.ORD, newValue: @rb]};
  EXITS Exit => {
    rb: XString.ReaderBody ← Defs.GetMessage[Defs.MessageKey.problemsWithDoc.ORD];
    Attention.Post[@rb]};
  };
```

DocInterchangeDefs
(and other Interfaces)

*** DRAFT Documentation ***

61

ChartDataInstallDefs

61.1 Overview

ChartDataInstallDefs provides the ability to install new data in a chart without regard to the type of the chart. Specifics such as line styles or shadings are not affected. Typical clients include those that have changing data depicted in chart form (e.g., one such client is the ViewPoint database package).

61.2 Interface Items

The primary type in this interface is the **Handle**, which points to a record of chart information. Clients may obtain a handle by either calling **GetChartFromInstance** or **GetChartFromSelection**. Once the client has a handle, several functions can be performed on the chart; installing new data or validating the chart are some examples.

```
Handle: TYPE = LONG POINTER TO Object;
```

```
Object: TYPE = RECORD [
   type: ChartType,
   instance: DocInterchangeDefs.Instance,
   validateChart: ValidateChartProc,
   validateData: ValidateDataProc,
   plot: PlotProc,
   free: FreeProc];
```

```
ChartType: TYPE = MACHINE DEPENDENT {bar(0), line(1), pie(2), last(15)};
```

type refers to the manner that the chart displays information. **instance** is a record that has pointers to the chart data. **validateChart** is a call-back procedure to check if the chart can be edited. **validateData** checks the validity of new data to be installed in the chart. **plot** actually installs the data, while **free** releases the handle.

```
ValidateChartProc: TYPE = PROC [h: Handle]
   RETURNS [ChartValidity];
```

```
ChartValidity: TYPE = MACHINE DEPENDENT {ok(0), closed(1), readOnly(2), last(15)};
```

ValidateChartProc checks the chart and returns its status. This procedure should be called before any attempt to operate on the chart.

```
ValidateDataProc: TYPE = PROC [
   h: Handle,
   data: Data,
   changes: Selections]
   RETURNS [DataValidity];
```

```
DataValidity: TYPE = MACHINE DEPENDENT RECORD [
   v(0): SELECT result(0): DataValidityResult FROM
      ok => NULL,
      invalidSource => NULL,
      sizeMismatch => [extraRows(1): INTEGER, extraCols(2): INTEGER],
      nonNumericValue => [row(1): CARDINAL, col(2): CARDINAL],
      illegalValue => [row(1): CARDINAL, col(2): CARDINAL],
      unknown => NULL,
      last => NULL,
   ENDCASE];
```

```
DataValidityResult: TYPE = MACHINE DEPENDENT {
   ok(0), invalidSource(1), sizeMismatch(2), nonNumericValue(3), illegalValue(4),
   unknown(5), last(15)};
```

ValidateDataProc checks the validity of the new data that the client intends to install. The data is not actually installed in this step. **data** is a pointer to the current data. **changes** specifies which items are being validated.

DataValidity indicates the validity of the data and in the case of bad data, some additional information. **extraRows** is the number of extra rows the new data has relative to the chart's current data table. A negative number means the chart currently has more rows than the data. **extraCols** is the analagous number for columns. **row** and **col** indicate the position of the charts problem.

```
PlotProc: TYPE = PROC [
   h: Handle,
   data: Data,
   changes: Selections];
```

```
Selections: TYPE = PACKED ARRAY Values OF BOOLEAN;
```

```
all: Selections = ALL[TRUE];
```

```
Values: TYPE = {title, data, rowLabels, colLabels, orientation};
```

PlotProc sets the chart's data and then redraws the chart. **data** is a pointer to the new data to be installed. **changes** specifies exactly which data is set.

```
Data: TYPE = LONG POINTER TO DataRec;
```

```
DataRec: TYPE = RECORD [
   title: XString.Reader ← NIL,
   data: DataValues,
   rowLabels: Labels ← NIL,
   colLabels: Labels ← NIL,
   orientation: Orientation ← row];

DataValues: TYPE = LONG POINTER TO RowSeq;

RowSeq: TYPE = RECORD [
   rows: SELECT format: DataFormat FROM
      string = > [SEQUENCE nRows: CARDINAL OF StringRow],
      numeric = > [SEQUENCE nRows: CARDINAL OF NumericRow]
   ENDCASE];

DataFormat: TYPE = {string, numeric};

Labels: TYPE = LONG POINTER TO LabelSeq;
LabelSeq: TYPE = RECORD [SEQUENCE length:CARDINAL OF XString.Reader];

Orientation: TYPE = {column, row};
```

DataRec contains the data to be installed. title is the title of the chart. data points to a sequence-containing record of data values. rowLabels and colLabels point to sequence-containing records of row and column labels, respectively. orientation specifies whether columns or rows are the chart's data sets.

```
StringRow: TYPE = LONG POINTER TO StringRowElements;
StringRowElements: TYPE = RECORD [SEQUENCE nCols: CARDINAL OF XString.Reader];

NumericRow: TYPE = LONG POINTER TO NumericRowElements;
NumericRowElements: TYPE = RECORD [SEQUENCE nCols: CARDINAL OF XLReal.Number];
```

StringRow points to a sequence-containing record of readers; similarly, NumericRow points to a sequence-containing record of numbers.

Hence, the data to be installed looks like Figure 61.1:

```
GetChartFromInstance: PROC [instance: DocInterchangeDefs.Instance]
   RETURNS [Handle];
```

This procedure returns a handle to the chart given by instance. The result will be NIL if the instance is not a chart. Note that a non-NIL handle doesn't guarantee that the chart is valid; it only guarantees that the instance is a chart. Clients should call the chart's validateChartProc to determine the chart's validity.

```
GetChartFromSelection: PROC RETURNS [Handle];
```

This procedure obtains a handle by converting the current selection. If the current selection is not a chart, NIL will be returned.

```
FreeProc: TYPE = PROC [h: Handle];
```

FreeProc frees the handle passed in by GetChartFromSelection or GetChartFromInstance.

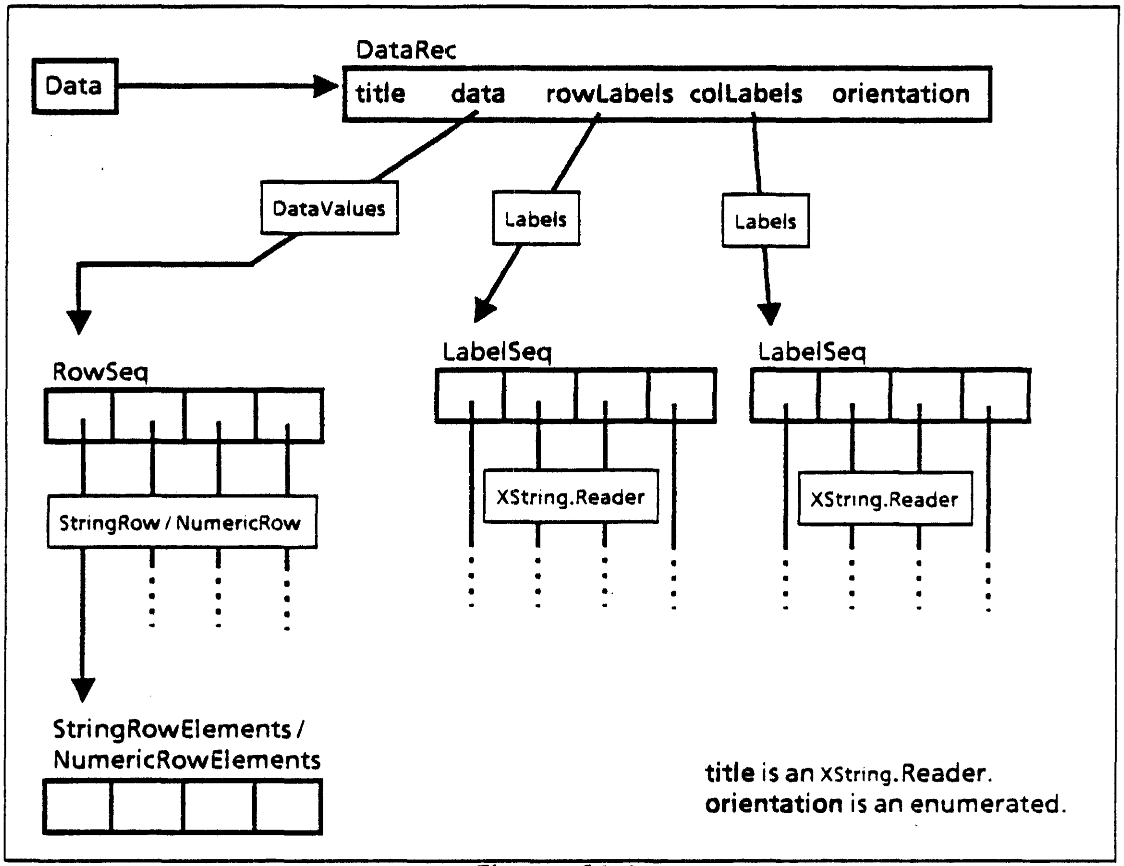

DataRec
Data
title data rowLabels colLabels orientation
DataValues
Labels
Labels
RowSeq
LabelSeq
LabelSeq
StringRow / NumericRow
XString.Reader
XString.Reader
StringRowElements / NumericRowElements
title is an XString.Reader.
orientation is an enumerated.

Figure 61.1

OutOfRoomForGraphics: ERROR;

Raised by **handle.plot** if there is no more room in the document to insert graphic object (the components of charts).

61.3 Usage

The typical pattern of use for this module is:

```
handle ← GetChartFromSelection[]
IF handle # NIL THEN {
DO
   <<get raw data;>>
   IF handle.validateChart[handle] # ok THEN {<<error;>> LOOP};
   <<determine which pieces of data are to be changed>>
   <<allocate and fillin data record>>
   IF handle.validateData[handle, data, selections] # ok THEN <<error;>>
   handle.plot[handle, data, selections];
   ENDLOOP;
handle.free[handle];
};
```

61.4 Index of Interface Items

62

DocInterchangeDefs

62.1 Overview

The **DocInterchangeDefs** interface enables clients to create a new ViewPoint document or read an existing one. However, it does not support inserting new information or changing or deleting the contents of a document.

DocInterchangeDefs provides procedures to create or read any of the basic document structures, such as text; textual "tiles;" fields; headings and footings; or frames of various types. It does not include procedures to manipulate contents of frames, however.

To create content within frames, the client must use interfaces specific to a particular frame type. **GraphicsInterchangeDefs** provides facilities for creating or reading graphics frames; **TableInterchangeDefs** provides facilities for creating or reading tables; **TextInterchangeDefs** provides facilities for creating or reading text frames. These are currently the only frame content interfaces available.

62.1.1 Creating Documents

To create a ViewPoint document, the first step is to call the procedure **StartCreation**. This sets up the data structures for the document and returns a **Doc**, which is a long pointer to an opaque type that represents the document. With **TextInterchangeDefs**, the client calls **AppendAnchoredFrame** first and then uses **TextInterchangeDefs** to append information to the text frame

The next step is to add information to the document with various **Append*** procedures: **AppendAnchoredFrame**, **AppendChar**, **AppendColumnBreak**, **AppendField**, **AppendNewParagraph**, **AppendPageBreak**, **AppendPFC** (Page Format Character), **AppendText**, or **AppendTile**.

With **AppendAnchoredFrame**, the client would typically call **GraphicsInterchangeDefs** or **TableInterchangeDefs** to create the contents of the frame, and then call **AppendAnchoredFrame** to add that frame and its contents to the document.

AppendField, **AppendPFC**, and **AppendTile** all have return values: this allows the client to call **Append*** routines recursively to add text and formatting information to fields, tiles, or PFC headers.

When the document contains all the desired information, the client should call **FinishCreation**, which returns an **NSFile.Handle** for the newly created file.

62.1.2 Enumerating documents

To enumerate the contents of an existing ViewPoint document, the client should start by calling **Open**, which opens the document and returns a **Doc** handle for that document. The next step is to call **Enumerate**, passing in the **Doc** and an **EnumProcs** record. The **EnumProcs** record contains a set of callback procedures, one for each of the following structures: anchored frame, column break, field, new paragraph, page break, page format character, text, tile.

Enumerate proceeds sequentially from the beginning of the document: as it comes to different structures within the document, it calls the appropriate callback procedure, which handles it appropriately. Each of these procedures returns a boolean value **stop**; if any one of the procedures returns **stop** = TRUE, the enumeration will terminate. If **stop** is never TRUE, the enumeration will continue to the end of the document.

When the enumeration is complete, the client should call **Close** to free all associated data structures and close any open file handles to the document.

62.2 Interface Items

62.2.1 Data types

The basic data structure of **DocInterchangeDefs** is the **TextContainer**, which is any object that can contain text. A **TextContainer** can be a caption, document, field, heading, footing, or spare (spares are for future compatability).

```
TextContainer: TYPE = RECORD [
    var: SELECT type: * FROM
        caption = > [h: Caption],
        doc = > [h: Doc],
        field = > [h: Field],
        heading = > [h: Heading],
        footing = > [h: Footing],
        spare1 = > [h: SpareTC],
        spare2 = > [h: SpareTC],
        spare3 = > [h: SpareTC],
        spare4 = > [h: SpareTC],
    ENDCASE];

Caption: TYPE = LONG POINTER TO CaptionObject;
CaptionObject: TYPE;
```

```
Doc: TYPE = LONG POINTER TO DocObject;
DocObject: TYPE;

Field: TYPE = LONG POINTER TO FieldObject;
FieldObject: TYPE;

Heading: TYPE = LONG POINTER TO HeadingObject;
HeadingObject: TYPE;

Footing: TYPE = LONG POINTER TO FootingObject;
FootingObject: TYPE;

Tile: TYPE = LONG POINTER TO TileObject;
TileObject: TYPE;Footing: TYPE = LONG POINTER TO FootingObject;

SpareTC: TYPE = LONG UNSPECIFIED;
```

Note that **TextContainers** must contain at least one **newParagraph** character, since the paragraph properties of any text are always inherited from the preceding **newParagraph** character. The **DocInterchange** implementation supplies the initial **newParagraph** characters as required; the client should assume they already exist.

62.2.2 Creating documents

62.2.2.1 Initializing a document

The client calls **StartCreation** to initiate the document creation process.

```
StartCreation: PROC [
    paginateOption: PaginateOption ← compress,
    wantHeadingHandles, wantFootingHandles: BOOL ← FALSE,
    initialFontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    initialParaProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    initialPageProps: DocInterchangePropsDefs.ReadonlyPageProps ← NIL]
    RETURNS [
        doc: Doc,
        leftHeading, rightHeading: Heading,
        leftFooting, rightFooting: Footing,
        status: StartCreationStatus];

PaginateOption: TYPE = MACHINE DEPENDENT {
    none(0), simple, compress, firstAvailable, lastAvailable(255)};
```

paginateOption specifies the type of pagination that will occur when the client calls **FinishCreation**. It is specified here rather than in **FinishCreation** to enable performance optimizations based on the type of pagination that will eventually occur.

compress pagination provides all the outward signs of pagination, such as page format properties, and leaves the structure of the document in its optimized form.

simple pagination provides the outward signs of pagination but does not leave the document in its optimized form, so subsequent editing may be slower than with **compress** pagination. **simple** pagination is slightly faster than **compress**.

none leaves the document in its raw form. This can lead to very slow editing, and potentially to loss of data. If the document will be more than a few pages long, client must specify a **paginateOption** other than **none** to avoid losing data.

wantHeadingHandles and **wantFootingHandles** specify whether the document will have headings and footings.

initialFontProcs, initialParaProcs, and **initialPageProps** specify the initial properties for the document. If you do not specify a field of initial properties, **StartCreation** will use the document default properties. (For information on document default properties, see the DocInterchangePropsDefs chapter)

In the **pageProps**, the client must ensure that the page margins leave at least one inch (72 points) for text. That is, (left margin + right margin + 72 < = page width), and (top margin + bottom margin + 72 < = page height).

StartCreation returns a **Doc** handle, handles for headings and footings, and a status. The **Doc** handle represents the new document. The client should pass this handle to the **Append*** procedures described below to add information to the document, and then eventually release the handle with a call to **FinishCreation**.

If the client releases the handle without ever calling any **Append*** routines, the file will contain a 1-page document containing a single **newParagraph** and **pageFormat** character, with the initial font, paragraph, and page props as specified.

The heading and footing handles that are returned will be NIL unless the client specified **wantHeadingHandles** or **wantFootingHandles** = TRUE. If the headings or footings are valid, the client should call various **Append*** routines to add text and formatting information, and then later release each handle with a call to **ReleaseHeading** or **ReleaseFooting**. See section 62.2.2.3 for details.

```
StartCreationStatus: TYPE = MACHINE DEPENDENT {
    ok(0), notEnoughDiskSpace, notEnoughVM, firstAvailable, lastAvailable(255)};
```

StartCreation returns a status code, which can have any of the following values:

ok	Everything was fine.
notEnoughDiskSpace	There isn't enough disk space to perform the operation.
notEnoughVM	There isn't enough contiguous virtual memory to create.

63.2.2.2 Adding to a document

The **Append*** routines below add various kinds of information to **TextContainers**.

AppendChar, AppendField, AppendNewParagraph, AppendText, and **AppendTile** take a **TextContainer** as a parameter and add the specified information to that container. The remaining procedures (**AppendAnchoredFrame, AppendColumnBreak,**

AppendPageFormatCharacter, AppendPageBreak) take only a Doc, and not a general purpose TextContainer; other TextContainers cannot contain the various special characters.

With all of these procedures, the client must manage the storage for the property records or other data structures passed in, except for handles obtained from the interface itself. The storage for the properties must remain valid during the call to Append*; after Append* returns, the client may do anything it chooses with the storage (typically, free it.)

The Append* procedures often allow the client to set font, paragraph, or page properties. Defaulting any of these arguments will cause the newly appended text or object to inherit the properties of the preceding text/object and not the application-wide default properties.

If an Append* routine returns a non-NIL handle, the client is responsible for later freeing that handle with a call to an appropriate Release* routine. See section 63.2.2.3 for details.

```
AppendAnchoredFrame: PROC [
    to: Doc,
    type: AnchoredFrameType,
    anchoredFrameProps: DocInterchangePropsDefs.ReadonlyFrameProps,
    content: Instance ← instanceNil,
    wantTopCaptionHandle, wantBottomCaptionHandle,
    wantLeftCaptionHandle, wantRightCaptionHandle: BOOL ← FALSE,
    anchorFontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [
        anchoredFrame: Instance,
        topCaption, bottomCaption,
        leftCaption, rightCaption: Caption];
```

AppendAnchoredFrame appends the anchored frame type with properties anchoredFrameProps to the document Doc.

```
AnchoredFrameType: TYPE = MACHINE DEPENDENT {
    nil(0), bitmap, cuspButton, equation, graphics, IMG, table, text,
    illustrator, firstAvailable, lastAvailable(255)};
```

content is the contents of the frame. Currently, there are interfaces to support creating graphics frames, tables, and text frames. For information on creating graphics frames, cuspButtons, or bitmap frames, see GraphicsInterchangeDefs. For information on creating table content, see TableInterchangeDefs. For information on creating text frames see TextInterchangeDefs.

want*CaptionHandle specifies which captions the frame should have. anchorFontProps specifies the font properties of the frame anchor. Changing the font properties of the anchor does not affect how that anchor appears on the display, but does affect the default properties that succeeding characters will inherit.

AppendAnchoredFrame returns handles to the frame and its captions. The caption handles will be non-NIL only if the client specified TRUE for the corresponding

want*CaptionHandle parameter. The client must later release each valid caption handle with **ReleaseCaption**.

The return parameter **anchoredFrame** is currently used only by the **TextInterchangeDefs** interface for appending text frames.

```
AppendChar: PROC [
    to: TextContainer,
    char: XChar.Character,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];
```

AppendChar appends one or more copies of the text character **char** to the specified **TextContainer**. **nToAppend** specifies the number of copies of the character that are to be appended; **fontProps** specifies the character properties.

```
AppendColumnBreak: PROC [
    to: Doc, fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];
```

AppendColumnBreak appends a column break character to a document. **fontProps** are the properties of the column break character; these properties do not affect the appearance of the character itself, but they do affect the properties that succeeding characters will inherit.

```
AppendField: PROC [
    to: TextContainer,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [field: Field];
```

AppendField appends a field to the specified **TextContainer**. **AppendField** returns a field; the client can then add information to the field by using the **Field** as the **TextContainer** in other calls to **Append*** routines (but not **AppendField** again.) When the client is through with the field, it must release it via **ReleaseField**. See section 62.2.2.3.

Note that the client cannot set the fill-in order of the fields when they are appended to the document.

```
AppendNewParagraph: PROC [
    to: TextContainer,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];
```

AppendNewParagraph appends one or more new paragraph characters to a **TextContainer** object. **nToAppend** specifies the number of characters to be appended. **paraProps** and **fontProps** specify the properties for the paragraph. If **paraProps** is NIL, the new paragraph inherits the props of the previous paragraph; otherwise, **paraProps** determines the properties of the paragraph.

Note that **TextContainers** always contain at least one **newParagraph** character. The client does not have to provides these initial **newParagraph** characters; **DocInterchange** implementation supplies them as required.

```
AppendPageBreak: PROC [
    to: Doc, fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];
```

AppendPageBreak appends a page break character to the text of a document. The **fontProps** do not affect the appearance of the page break character itself, but they do affect the properties that succeeding characters will inherit.

```
AppendPFC: PROC [
    to: Doc,
    pageProps: DocInterchangePropsDefs.ReadonlyPageProps,
    wantHeadingHandles, wantFootingHandles: BOOL ← FALSE,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [
        leftHeading, rightHeading: Heading,
        leftFooting, rightFooting: Footing];
```

AppendPFC appends a page format character to the main document text.

pageProps specify the properties for the new page. The client must ensure that the page margins leave at least one inch (72 points) for text. That is, (**left margin + right margin + 72 < = page width**), and (**top margin + bottom margin + 72 < = page height**).

The heading and footing handles that are returned will be NIL unless the client specified **wantHeadingHandles** or **wantFootingHandles = TRUE**.

If the heading and footing handles are valid, the client can then use them as **TextContainers** for further calls with **Append*** procedures. If the headers are to be the same on left and right pages, only **leftHeading** need contain the heading; **rightHeading** should be NIL. The same rule applies for **leftFooting** and **rightFooting**.

When creating a heading or footing, the client should note that there are no automatic positioning parameters for information in headers and footers; the client must call the appropriate **Append*** procedures to add the desired text and position it with standard text formatting, such as white-space characters, paragraph alignment, leading, line height, and tabs.

Additionally, there is no page number pattern; the client must place any surrounding text directly in the heading/footing text, inserting the # character at the position(s) where a page number is desired. (Note that there is a procedure, **DocInterchangePropsDefs.GetPageNumberDelimiter**, that returns this character.)

The client must later free every non-NIL heading or footing with a call to **ReleaseHeading** or **ReleaseFooting**.

```
AppendText: PROC [
    to: TextContainer,
    text: XString.Reader,
    textEndContext: XString.Context,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];
```

AppendText appends the text with the specified properties to the TextContainer. For efficiency, the client should pass the appropriate textEndContext if it is known (just like XString.AppendReader).

```
AppendTile: PROC [
    to: TextContainer,
    type: Atom.ATOM,
    data: LONG POINTER ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [tile: Tile];
```

AppendTile is for future use. The tile type and data format are defined elsewhere, agreed upon by parties on either side of this interface.

62.2.2.3 Releasing storage

```
ReleaseCaption: PROC [captionPtr: LONG POINTER TO Caption];

ReleaseField: PROC [fieldPtr: LONG POINTER TO Field];

ReleaseHeading: PROC [headingPtr: LONG POINTER TO Heading];

ReleaseFooting: PROC [footingPtr: LONG POINTER TO Footing];

ReleaseTile: PROC [tilePtr: LONG POINTER TO Tile];

ReleaseSpare1: PROC [ptr: LONG POINTER TO SpareTC];

ReleaseSpare2: PROC [ptr: LONG POINTER TO SpareTC];

ReleaseSpare3: PROC [ptr: LONG POINTER TO SpareTC];

ReleaseSpare4: PROC [ptr: LONG POINTER TO SpareTC];
```

The client must call ReleaseCaption, ReleaseField, ReleaseFooting, ReleaseHeading, ReleaseTile. or ReleaseSpare to release resources associated with any non-NIL handle obtained from any Append* procedure.

After calling Release*, the handle will be invalid. To help prevent use of an invalid handle, the Release* routines take a pointer to the handle, and set the handle itself to NIL. (This is similar to Mesa's FREE operation.)

62.2.2.4 Finalizing a document

```
FinishCreation: PROC [
    docPtr: LONG POINTER TO Doc,
    checkAbortProc: CheckAbortProc ← NIL,
    checkAbortClientData: LONG POINTER ← NIL]
    RETURNS [
docFile: NSFile.Handle,
session: NSFile.Session,
status: FinishCreationStatus];
```

When the document is complete, the client must call **FinishCreation** to finalize the document and release the **Doc** handle. **FinishCreation** returns an **NSFile.Handle** to the newly-created document, an **NSFile.Session**, and a status. The file handle is valid in the returned NSFile session. The **session** returned will be the default session if the client called **FinishCreation** normally (not by a forked background process). If **StartCreation** was called by a forked background process, then the returned session will be a private session created by **StartCreation**. The client must kill this private session by calling **NSFile.Logoff[session]** after it's finished processing the document file. The document that **FinishCreation** provides will be in paginated form if the client so specified in **StartCreation**.

```
CheckAbortProc: TYPE = PROC [clientData: LONG POINTER] RETURNS [abort: BOOL];
```

If the client specified a **checkAbortProc**, then a call-back procedure will be invoked before the call to **FinishCreation** returns; this call-back can abort the document's completion. **checkAbortClientData** is a client defined argument and is passed into the call-back procedure.

```
FinishCreationStatus: TYPE = MACHINE DEPENDENT {ok(0),
    aborted, okButNotEnoughDiskSpaceToPaginate, okBuNotEnoughVMToPaginate,
    okButUnknownPaginateProblem, firstAvailable, lastAvailable(255)};
```

FinishCreation also returns a status code, which can have any of the following values:

aborted	do not complete the document.
okButNotEnoughDiskSpaceToPaginate, **okBuNotEnoughVMToPaginate,** **okButUnknownPaginateProblem**	the document is finished but left unpaginated

This document file is temporary, and will be purged from the NSFile system if a reboot occurs before it is made permanent. To make the file permanent, the client should call move it to the current user desktop with **NSFile.Move**, followed by a call to **StarDesktop.AddReference** to put the icon on the display. (See section 62.3 for an example of this.)

```
AbortCreation: PROC [docPtr: LONG POINTER TO Doc];
```

AbortCreation aborts document creation and deallocates the storage associated with that document.

62.2.2.5 Utilities

The following procedures are utilities that may be of use to the client creating a document.

```
GetModeProps: PROC [doc: Doc, modeProps: DocInterchangePropsDefs.ModeProps];

SetModeProps: PROC [
    doc: Doc,
    modeProps: DocInterchangePropsDefs.ReadonlyModeProps,
    selections: DocInterchangePropsDefs.ModeSelections];
```

Get or set the mode properties for the document; these procedures may be called at any time. When setting mode properties, only those properties designated by TRUE selections will be changed.

```
SetCurrentParagraphProps: PROC [
    textContainer: TextContainer,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps];
```

SetCurrentParagraphProps can be called at any time, such as in the middle of a paragraph or (even if it makes no sense) repeatedly with different properties. If it is called repeatedly, only the most recent call will remain in effect. The client can call this procedure on any TextContainer, such as a document.

SetCurrentParagraphProps affects the entire current paragraph, including any portion not yet appended at the time it is called. The properties also affect all subsequent paragraphs unless the client overrides the properties with new ones passed to AppendNewParagraph, or by another call to SetCurrentParagraphProps.

Note, however, that setting paragraph properties on a TextContainer will cause an error if that TextContainer does not contain any paragraph characters. Although DocInterchange does guarantee that every TextContainer will contain at least one new paragraph character, those paragraph characters are added (if necessary) during the Append* routines. Thus, calling SetCurrentParagraphProps before calling any Append* routines will cause an error. To avoid this problem, the client can simply call AddNewParagraph to ensure that the TextContainer does have a paragraph character. Since the Append* routines only add a new paragraph if necessary, this will not cause duplication.

62.2.3 Enumerating documents

62.2.3.1 Open

```
Open: PROC [
    docFileRef: NSFile.Reference,
    session: NSFile.Session ← NSFile.nullSession,
    password: XString.Reader ← NIL]
    RETURNS [doc: Doc, status: OpenStatus];
```

To enumerate a document, the first step is to call Open. Open takes a NSFile.Reference for a file and opens it for reading in the NSFiling session specified by the session argument. The client should then pass Doc returned to Enumerate, which will parse the document.

password is provided in anticipation of future password-locking of documents. password is currently ignored.

```
OpenStatus: TYPE = MACHINE DEPENDENT {
    ok(0), malFormed, incompatible, notLocal, outOfDiskSpace, outOfVM, busy,
    invalidPassword, firstAvailable, lastAvailable(255)};
```

Open also returns a status code, which can have any of the following values:

ok	Everything was fine
malFormed	The Document is inconsistent internally.
incompatible	The document is more than one version old, and the upgrader cannot handle it.
notLocal	The document is not on the workstation, so it cannot be opened.
outOfDiskSpace	There isn't enough disk space to open the document.
outOfVM	There isn't enough contiguous virtual memory.
busy	Another process is using the file (e.g. background pagination).
invalidPassword	The user does not have the credentials to open the document.

62.2.3.2 Enumerate

```
Enumerate: PROC [
    textContainer: TextContainer,
    procs: EnumProcs,
    clientData: LONG POINTER ← NIL]
    RETURNS [dataSkipped: BOOL];
```

Enumerate parses the contents of the specified **TextContainer**.

procs is a record that contains client-defined callback procedures to enumerate the various kinds of structures that can be found in a **TextContainer**.

```
EnumProcs: TYPE = LONG POINTER TO EnumProcsRecord;

EnumProcsRecord: TYPE = RECORD [
    anchoredFrameProc: AnchoredFrameProc ← NIL,
    columnBreakProc: ColumnBreakProc ← NIL,
    fieldProc: FieldProc ← NIL,
    newParagraphProc: NewParagraphProc ← NIL,
    pageBreakProc: PageBreakProc ← NIL,
    pfcProc: PFCProc ← NIL,
    textProc: TextProc ← NIL,
    tileProc: TileProc ← NIL,
    spare1Proc: SpareProc ← NIL,
    spare2Proc: SpareProc ← NIL,
    spare3Proc: SpareProc ← NIL,
    spare4Proc: SpareProc ← NIL];
```

Each of the procedures in an **EnumProcsRecord** takes as parameters the properties of the structure and its content when appropriate. Note that the storage for the properties passed to these procedures is temporary; the client must explicitly copy any properties it wishes to save. For a description of the various properties, see the corresponding **Append*** routines.

```
AnchoredFrameProc: TYPE = PROC [
    clientData: LONG POINTER,
    type: AnchoredFrameType,
    anchorFontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    anchoredFrame: Instance,
    anchoredFrameProps: DocInterchangePropsDefs.ReadonlyFrameProps,
    content: Instance,
    topCaption,
    bottomCaption,
    leftCaption,
    rightCaption: Caption]
    RETURNS [stop: BOOL ← FALSE];

Instance: TYPE[2];
instanceNil: Instance = LOOPHOLE[LONG[0]];

ColumnBreakProc: TYPE = PROC [
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps]
    RETURNS [stop: BOOL ← FALSE];

FieldProc: TYPE = PROC [
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    field: Field]
    RETURNS [stop: BOOL ← FALSE];

NewParagraphProc: TYPE = PROC [
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps]
    RETURNS [stop: BOOL ← FALSE];

PageBreakProc: TYPE = PROC [
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps]
    RETURNS [stop: BOOL ← FALSE];

PFCProc: TYPE = PROC [
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    pageProps: DocInterchangePropsDefs.ReadonlyPageProps,
    leftHeading, rightHeading: Heading,
    leftFooting, rightFooting: Footing]
    RETURNS [stop: BOOL ← FALSE];
```

In a PFCProc, if the headers are the same on left and right pages, only leftHeading will contain the heading; rightHeading will be NIL. (Of course, leftHeading can be NIL if it has no content.) The same rule applies for leftFooting and rightFooting.

```
TextProc: TYPE = PROC [
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    text: XString.Reader,
    textEndContext: XString.Context]
    RETURNS [stop: BOOL ← FALSE];
```

In a **TextProc**, **textEndContext** will always be accurate; it will never be **XString.unknownContext**.

```
TileProc: TYPE = PROC [
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    type: Atom.ATOM,
    data: LONG POINTER,
    tile: Tile]
    RETURNS [stop: BOOL ← FALSE];

SpareProc: TYPE = PROC [
    clientData: LONG POINTER,
    data: LONG UNSPECIFIED]
    RETURNS [stop: BOOL ← FALSE];
```

As it encounters an object of a particular type, **Enumerate** will call the appropriate procedure. If the client defaults a particular procedure, **Enumerate** will ignore any objects of that type.

Each procedure has a **stop** return parameter; the enumeration will stop if **stop** ever has the value **TRUE**. Some of the procedures also have a **TextContainer** handle as a parameter; the client can use this **TextContainer** recursively in other calls to **Enumerate** to obtain the contents of the **TextContainer**.

The **clientData** pointer passed in to Enumerate is passed to the callback procedures invoked by (that call to) Enumerate. (The **clientData** may be different at different recursion levels, of course.)

The handle (header, caption, etc.) supplied to the client in the call-back is readonly and is valid only during the call-back's invocation; the client is not reponsible for releasing this handle. It is possible for such a handle to be **NIL**; a **NIL** handle means that the corresponding object has no text content.

Note that the enumeration does include the default paragraph and page format characters supplied with the **TextContainer**. Thus, when copying a document into a new document, the client should be careful to avoid copying the default paragraph and page format properties, since that would cause duplication.

62.2.3.3 Close

```
Close: PROC [docPtr: LONG POINTER TO Doc];
```

When through with an enumeration, the client should call **Close**, which releases storage associated with the **Doc** handle and sets the **Doc** handle to **NIL**.

62.2.4 Errors

```
Error: ERROR [why: ErrorCode];

ErrorCode: TYPE = MACHINE DEPENDENT {
    containerFull(0), documentFull, readonlyDoc, outOfDiskSpace, outOfVM,
    objectIllegalInContainer, badParameter, unimplemented, firstAvailable,
    lastAvailable(255)};
```

Any of the Append* procedures can raise an error, which can be one of the following types:

containerFull	there is no more room to append to this container.
documentFull	no more room in the document.
readonlyDoc	document opened in ReadOnly mode.
outOfDiskSpace	not enough disk space for the operation.
outOfVM	not enough virtual memory for the operation.
objectIllegalInContainer	attempted to add an object to a container that does not support that object type.
badParameter	one of the arguments specified was invalid in this context.
unimplemented	this function is not supported

62.2.5 Fill-in Order

DocInterchangeDefs provides procedures to append, enumerate, and clear the fill-in order of fields and tables.

```
AppendItemToFillInOrder: PROC [
    doc: Doc,
    fillInOrderItemName: XString.Reader,
    itemType: FillInOrderItemType];
```

doc is the document that contains the field or table.

fillInOrderItemName is the name of the object being added to the fill-in order.

```
FillInOrderItemType: TYPE = MACHINE DEPENDENT {
    field(0), table, firstAvailable, lastAvailable(255)};
```

FillInOrderItemType specifies the type of object that will be added to the fill-in order.

```
EnumerateFillInOrder: PROC [
    doc: Doc,
    proc: FillInOrderProc,
    clientData: LONG POINTER ← NIL];

FillInOrderProc: TYPE = PROC [
    clientData: LONG POINTER,
    fillInOrderItemName: XString.Reader,
    itemType: FillInOrderItemType]
    RETURNS [stop: BOOL ← FALSE];
```

proc is a call-back procedure that is invoked once for each object in the fill-in order. The arguments passed into proc include the name of the enumerated object as well as its type. The **FillInorderProc** can return stop = TRUE to halt the enumeration.

clientData is client defined data that is passed to **proc**.

```
ClearFillInOrder: PROC [doc: Doc];
```

Clear the fill-in order for the entire document.

62.3 Usage/Examples

Here is an example of both enumeration and creation. This program adds the command DocEx to the Attention Window. When called, this command checks to see if the current selection is a document. If it is, then the program enumerates the contents of that document and copies the information into a new document. The time consuming process of copying the file is performed in the background by detaching the procedure **PerformCopy**.

```
DIRECTORY
... ;

DocExample: PROGRAM IMPORTS DocInterchangeDefs, ... = {

-- CopyData defines the handles and properties needed for each instance of the copy
-- A record of this type is allocated from the system zone each time the CopyEx command
-- is invoked; it is freed if an error occurs or when the copy completes.
CopyDataHandle: TYPE = LONG POINTER TO CopyData;
CopyData: TYPE = RECORD [
    sourceDoc, targetDoc: DocInterchangeDefs.Doc ← NIL,
    copyEnumProcs: DocInterchangeDefs.EnumProcsRecord ← [],
    fontProps: DocInterchangePropsDefs.FontPropsRecord ←
        DocInterchangePropsDefs.nullFontProps,
    paraProps: DocInterchangePropsDefs.ParaPropsRecord ←
        DocInterchangePropsDefs.nullParaProps,
    pageProps: DocInterchangePropsDefs.PagePropsRecord ←
        DocInterchangePropsDefs.nullPageProps,
    tabStops: TabStopsHandle ← NIL,
    backGroundSession: NSFile.Session ← NSFile.nullSession,
    ignoreNewPar, ignorePFC: BOOLEAN ← TRUE];

TabStopsHandle: TYPE = LONG POINTER TO TabStops;
TabStops: TYPE = RECORD [
 list: SEQUENCE length: CARDINAL OF DocInterchangePropsDefs.TabStop];

zone: UNCOUNTED ZONE = Heap.systemZone;
```

```
-- This is the command procedure. It first obtains a reference to the selected file
-- and then copies the contents of that document to a new document. The copy
-- occurs by forking a background procedure that runs in a separate NSFile.Session
MakeDoc: MenuData.MenuProc = {
    --get reference to selected file
    openStatus: DocInterchangeDefs.OpenStatus;
    IF Selection.CanYouConvert[file] THEN {
        selValue: Selection.Value ← Selection.Convert[file];
        docFileRef: NSFile.Reference = LOOPHOLE[
        selValue.value, LONG POINTER TO NSFile.Reference] ↑ ;

        -- allocate the data object from the system zone
        copyData: CopyDataHandle ← zone.NEW[CopyData ← []];

        --create a new session and open the source document in that session
        copyData.backGroundSession ← MakeNewSession[];
        [copyData.sourceDoc, openStatus] ←
            DocInterchangeDefs.Open[docFileRef, copyData.backGroundSession];

        -- if the input file is valid, fork a process to perform the copy
        IF openStatus = ok THEN Process.Detach[FORK PerformCopy[copyData]]
        ELSE {
            NSFile.Logoff[copyData.backGroundSession];
            zone.FREE[@copyData];
            UserTerminal.BlinkDisplay[]};
    }
    ELSE UserTerminal.BlinkDisplay[];
}; -- MakeDoc

-- create a new NSFiling session so that the copy can be performed in the background.
MakeNewSession: PROC [] RETURNS[session: NSFile.Session] = {
    user: Atom.ATOM ← Atom.MakeAtom["CurrentUser"L];
    prop: Atom.ATOM ← Atom.MakeAtom["IdentityHandle"L];
    identity: Auth.IdentityHandle = Atom.GetProp[user, prop] ↑ .value;
    session ← NSFile.Logon[identity];
};
```

```
-- Enumerate the source document in the background and append
-- its contents to a new document via call-back procs. When the
-- enumeration is complete, place the new file on the desktop.
PerformCopy: PROC [copyData: CopyDataHandle] = {
    docFile: NSFile.Handle; -- handle for the new file
    refDoc, refDt: NSFile.Reference; -- refs used when placing file on desktop
    desktop: NSFile.Handle;
    destSession: NSFile.Session; -- session for destination file
    startStatus: DocInterchangeDefs.StartCreationStatus;
    Process.SetPriority[Process.priorityBackground]; -- execute in the background

    -- initialize the enumeration call-back procs and retrieve the props
    -- from the source document to be used in creating the detination doc.
    copyData.copyEnumProcs ← InitEnumProcs[];
    GetInitialDocProps[copyData.sourceDoc, copyData];
    copyData.paraProps.tabStops ←
        IF copyData.tabStops = NIL THEN DESCRIPTOR[NIL, 0]
        ELSE DESCRIPTOR[@copyData.tabStops.list[0], copyData.tabStops.length];

    -- create the destination document using props obtained from the source doc
    [doc: copyData.targetDoc, status: startStatus] ←DocInterchangeDefs.StartCreation[
        paginateOption: simple,
        initialFontProps: @copyData.fontProps,
        initialParaProps: @copyData.paraProps,
        initialPageProps: @copyData.pageProps];

    -- free data allocated for StartCreation
    IF copyData.tabStops # NIL THEN zone.FREE[@copyData.tabStops];
    IF startStatus # ok THEN {
        DocInterchangeDefs.Close[@copyData.sourceDoc];
        NSFile.Logoff[copyData.backGroundSession];
        zone.FREE[@copyData];
        RETURN};

    -- enumerate the source document; copyData will be passed to each call-back
    [] ← DocInterchangeDefs.Enumerate[
        [doc[h: copyData.sourceDoc]], @copyData.copyEnumProcs, copyData];

    DocInterchangeDefs.Close[@copyData.sourceDoc];
    [docFile, destSession,] ← DocInterchangeDefs.FinishCreation[@copyData.targetDoc];

    -- if no errors occurred when finishing creation, place new file on desktop.
    refDoc ← NSFile.GetReference[docFile, destSession];
    refDt ← StarDesktop.GetCurrentDesktopFile[];
    desktop ← NSFile.OpenByReference[reference: refDt, session: destSession];
    NSFile.Move[file: docFile, destination: desktop, session: destSession];
    NSFile.Close[desktop, destSession];
    NSFile.Close[docFile, destSession];
    NSFile.Logoff[copyData.backGroundSession];
    NSFile.Logoff[destSession];
    StarDesktop.AddReferenceToDesktop[refDoc];
};
```

```
--The call-back procs for enumeration just add the specified structure to the new doc.

-- Add new paragraph to new document. If it is the first new paragraph
-- character, then ignore it, since new document will already have one.
AppendNewParToTargetDoc: DocInterchangeDefs.NewParagraphProc = {
    copyData: CopyDataHandle = clientData;
    IF copyData.ignoreNewPar THEN copyData.ignoreNewPar ← FALSE
    ELSE DocInterchangeDefs.AppendNewParagraph[
    [doc[h: copyData.targetDoc]], paraProps, fontProps];
};

--Append page break to new document
AppendPageBreakToTargetDoc: DocInterchangeDefs.PageBreakProc = {
    copyData: CopyDataHandle = clientData;
    DocInterchangeDefs.AppendPageBreak[copyData.targetDoc, fontProps];
};

-- Add page format character to new document. If it is the first format
-- character, then ignore it, since new document will already have one.
AppendPFCToTargetDoc: DocInterchangeDefs.PFCProc = {
    copyData: CopyDataHandle = clientData;
    IF copyData.ignorePFC THEN copyData.ignorePFC ← FALSE
    ELSE [] ← DocInterchangeDefs.AppendPFC[
        to: copyData.targetDoc, pageProps: pageProps, fontProps: fontProps];
};

--Append text to new document
AppendTextToTargetDoc: DocInterchangeDefs.TextProc = {
 copyData: CopyDataHandle = clientData;
 DocInterchangeDefs.AppendText[
    [doc[h: copyData.targetDoc]],
    text,
    textEndContext,
    fontProps];
};

-- initialize the enumeration call-back procedures
InitEnumProcs: PROC RETURNS[DocInterchangeDefs.EnumProcsRecord] = {
    RETURN[[newParagraphProc: AppendNewParToTargetDoc,
        pageBreakProc: AppendPageBreakToTargetDoc,
        pfcProc: AppendPFCToTargetDoc,
        textProc: AppendTextToTargetDoc]];
};
```

```
-- These procedures obtain the properties for the first paragraph and PFC
-- characters in the source file and store the information in the data object.

-- Copy the font, para, and page props of source document by
-- enumerating the first PFC and NewParagraph characters in the
-- source document and copying the properties to the data object.
GetInitialDocProps:    PROC[sourceDoc:    DocInterchangeDefs.Doc,copyData:
CopyDataHandle] = {
    tempEnumProcs: DocInterchangeDefs.EnumProcsRecord ← [
    newParagraphProc: FirstNewParProc,
    pfcProc: FirstPFCProc];
    [] ← DocInterchangeDefs.Enumerate[
    [doc[h: copyData.sourceDoc]], @tempEnumProcs, copyData];
};

-- save font, paragraph, and tab stop properties in
-- the data object then stop the enumeration.
FirstNewParProc: DocInterchangeDefs.NewParagraphProc = {
    copyData: CopyDataHandle = clientData;
    copyData.fontProps ← fontProps ↑ ;
    copyData.paraProps ← paraProps ↑ ;
    IF paraProps.tabStops # NIL THEN
        copyData.tabStops ← CopyTabs[paraProps.tabStops];
};

-- save page properties in data object
FirstPFCProc: DocInterchangeDefs.PFCProc = {
    copyData: CopyDataHandle = clientData;
    copyData.pageProps ← pageProps ↑ ;
    RETURN[TRUE]; -- return from enumeration
};

-- Copy the tabstop info into the data record
CopyTabs: PROC [tabs: DocInterchangePropsDefs.TabStops]
RETURNS[newTabs: TabStopsHandle] = {
    newTabs ← zone.NEW[TabStops[tabs.LENGTH]];
    FOR i: CARDINAL IN [0..tabs.LENGTH) DO
    newTabs[i] ← tabs[i];
    ENDLOOP;
};

-- Add command to attention menu.
Init: PROC = {
    name: XString.ReaderBody ← XString.FromSTRING["DocEx"L];
    Attention.AddMenuItem[MenuData.CreateItem[zone, @name, MakeDoc]];
};

Init[];

}.
```

63.4 Index of Interface Items

63

DocInterchangePropsDefs

63.1 Overview

This interface contains procedures and data types used to describe the properties in documents; it is intended for use with **DocInterchangeDefs**, **GraphicsInterchangeDefs**, **TableInterchangeDefs**, and **TextInterchangeDefs**.

63.2 Interface Items

63.2.1 Frame Properties

The chief type in this section is the **FramePropsRecord**, which describes the properties of an anchored frame.

```
FrameProps: TYPE = LONG POINTER TO FramePropsRecord;

ReadonlyFrameProps: TYPE = LONG POINTER TO READONLY FramePropsRecord;

FramePropsRecord: TYPE = RECORD [
    borderStyle: BorderStyle,
    borderThickness: CARDINAL,
    frameDims: FrameDims,
    fixedWidth,
    fixedHeight: BOOL,
    span: Span,
    verticalAlignment: VerticalAlignment,
    horizontalAlignment: HorizontalAlignment,
    topMarginHeight,
    bottomMarginHeight,
    leftMarginWidth,
    rightMarginWidth: CARDINAL,
    spare1: LONG CARDINAL];

BorderStyle: TYPE = MACHINE DEPENDENT {
    invisible(0),solid, dashed, broken, dotted, double, firstAvailable, lastAvailable(255)};
```

borderStyle specifies the characteristics of the lines that make up the frame border.

borderThickness specifies the thickness of the frame border. This value is in units of 1/72 inch.

```
FrameDims: TYPE = RECORD [w, h: CARDINAL];
```

frameDims specifies the height and width of the frame. These dimensions are also in units of 1/72 inch.

fixedWidth and **fixedHeight** indicate whether the frame will expand when necessary.

```
Span: TYPE = MACHINE DEPENDENT {partialColumn(0),
    fullColumn, partialPage, fullPage, firstAvailable, lastAvailable(255)};
```

span specifies how much of the page the frame occupies. This is not currently in use.

```
VerticalAlignment: TYPE = MACHINE DEPENDENT {
    top(0), bottom, floating, firstAvailable, lastAvailable(255)};

HorizontalAlignment: TYPE = MACHINE DEPENDENT {
    left(0), centered, right, floating, firstAvailable, lastAvailable(255)};
```

vertical and **horizontalAlignment** specify the alignment of the frame relative to the page.

topMarginHeight, **bottomMarginHeight**, **leftMarginWidth**, and **rightMarginWidth** are the margins of the frame, in units of 1/72 inch.

all items marked as **spare** are for future use.

63.2.2 Page Properties

The chief type in this section is the **PagePropsRecord**, which describes the various properties that can be associated with a page in a ViewPoint document.

```
PageProps: TYPE = LONG POINTER TO PagePropsRecord;

ReadonlyPageProps: TYPE = LONG POINTER TO READONLY PagePropsRecord;

PagePropsRecord: TYPE = RECORD [
    pageDims: PageDims, --layout
    topMarginHeight,
    bottomMarginHeight,
    leftMarginWidth,
    rightMarginWidth: CARDINAL,
    startingPageSide: PageSide,
    bindingMarginWidth: CARDINAL,
    nColumns: CARDINAL, -- column structure
    balancedColumns, unequalColumnWidths: BOOL,
    columnSpacing: CARDINAL,
    columnWidths: ColumnWidths,
    startingPageNumber: CARDINAL, --page numbering
    pageNumberFormat: NumberFormat,
    restartPageNumbering: BOOL,
    startingLineNumber, -- line numbering
    lineNumberInterval: CARDINAL,
    lineNumberFormat: NumberFormat,
    lineNumberLocation: LineNumberLocation,
    headingStartsOnThisPage, -- heading
    headingSameOnLeftRightPages,
    footingStartsOnThisPage, -- footing
    footingSameOnLeftRightPages: BOOL,
    spare1: LONG CARDINAL];

PageDims: TYPE = MACHINE DEPENDENT RECORD [w, h: CARDINAL];

PageSide: TYPE = MACHINE DEPENDENT {
    nil(0), left, right, firstAvailable, lastAvailable(255)};
```

pageDims are the width and height of the table, in units of 1/72 inch. topMarginHeight, bottomMarginHeight, leftMarginWidth, and rightMarginWidth describe the page margins; these values are also in units of 1/72 inch. startingPageSide indicates whether the first page of the document should be a left-hand page or a right-hand page; nil means that there is no difference between the two. bindingMarginWidth is the width of the binding margin, if there is one.

nColumns, balancedColumns, unequalColumnWidth, and columnSpacing determine column structure. nColumns is the number of columns; balancedColumns specifies whether the length of the column is equal to the length of the page. unequalColumnWidth indicates that the columns may have varying widths. columnSpacing is the amount of space between columns, in units of 1/72 inch.

```
ColumnWidths: TYPE = LONG POINTER TO ColumnWidthsRecord;

ColumnWidthsRecord: TYPE = RECORD [
    length: CARDINAL,
    spare1: LONG CARDINAL,
    widths: SEQUENCE maxLength: CARDINAL OF ColumnWidthRecord];

ColumnWidthRecord: TYPE = RECORD [
    w: CARDINAL,
    spare1: LONG CARDINAL];
```

The ColumnWidthsRecord record contains the number of columns (length) and a sequence that contains the width of each column. Spare fields are included in both the record and each of the sequence elements.

startingPageNumbers, pageNumberFormat, and restartPageNumbering describe the page numbering properties. startingPageNumber indicates the page number at which the numbering should start; restartPageNumbering specifies whether renumbering should restart for this page, or continue from where the last numbering left off.

```
NumberFormat: TYPE = MACHINE DEPENDENT {
    cardinal(0), lowerCaseLetter, upperCaseLetter, lowerCaseRoman,
    upperCaseRoman, firstAvailable, lastAvailable(255)};
```

pageNumberFormat specifies the format of the page number; currently only **cardinal** is implemented.

```
LineNumberLocation: TYPE = MACHINE DEPENDENT {
    leftMargin(0), rightMargin, outerMargin, bothMargins,
    firstAvailable,lastAvailable(255)};
```

startingLineNumber, **lineNumberInterval**, **lineNumberFormat**, and **lineNumberLocation** are not currently implemented.

The remaining properties describe headings and footings. **headingStartsOnThisPage** and **footingStartsOnThisPage** indicate whether the designated heading/footing should start on this page or the next; **headingSameOnLeftRightPage** and **footingSameOnLeftRightPages** specifies whether all pages have the same heading/footing. **spare1** is for future use.

63.2.3 Field Properties

The chief field property is the **FieldPropsRecord**, which describes the properties of a field.

```
FieldProps: TYPE = LONG POINTER TO FieldPropsRecord;

ReadonlyFieldProps: TYPE = LONG POINTER TO READONLY FieldPropsRecord;

FieldPropsRecord: TYPE = RECORD [
    language: MultiNational.Language,
    length: CARDINAL,
    required: BOOL,
    skipIf: SkipIfChoiceType,
    stopOnSkip: BOOL,
    type: FieldChoiceType,
    fillInRule, -- changing fillInRule implies changing fillInRuleRuns
    description,
    format,
    name,
    range,
    skipIfField: XString.ReaderBody,
    fillInRuleRuns: FontRuns,
    spare1: LONG CARDINAL];
```

language determines the format of date and amount fields. There are many formats, so you would have to check the format for each particular language.

length is the length of the field, in characters.

required indicates whether the user is required to fill in the field. If **required** is TRUE, the user will not be able to use NEXT or SKIP to advance to the next field until this field has a value.

```
SkipIfChoiceType: TYPE = MACHINE DEPENDENT {
    empty(0), notEmpty, never, always, firstAvailable, lastAvailable(255)};
```

skipIf defines the conditions under which the field can be skipped when the user presses the NEXT key. **stopOnSkip** specifies whether the skipping action should stop at this field or not.

```
FieldChoiceType: TYPE = MACHINE DEPENDENT {
    any(0), text, amount, date, firstAvailable, lastAvailable(255)};
```

type specifies the type of data that can be in the field. **any** indicates that the field can contain any characters, including frames and other fields. **text** indicates that the field can contain only letters, digits, and symbols entered from the keyboard. **amount** indicates that the field can contain only numbers, spaces, and the following symbols: +__ * $, . (). **date** specifies that entries in the field can contain only a date.

fillInRule defines the fill-in rule for this field.

description is posted for each field entered with the NEXT key when "PROMPT FOR FIELDS" is invoked. **format** controls the format in which information is presented. For a **type** of **text**, this property defines a required pattern that must be matched. For a **type** of **amount** or **date**, this field controls the form in which the contents of the field are presented, regardless of how the user enters them. For a **type** of **any**, the format property is not used.

name is the name of the field. If no name is provided, the field will automatically be named Field*n*, as in Field1, Field2, and so on.

range defines a specific range of acceptable entries.

skipIfField contains the name of the field that will appear in the property sheet Skip if field.

fillInRuleRuns is an auxiliary data structure that the client can attach to the **XString.Reader** that describes the fill-in rule for the field. A font run describes the subsequences of characters within a **Reader** that have the same font attributes.

63.2.4 Font Properties

The **PropsRecord** is the chief type in this Section. Section 63.2.4.1 describes the **fontDesc** field; section 63.2.4.2 describes the other fields in a **FontPropsRecord**.

```
FontProps: TYPE = LONG POINTER TO FontPropsRecord;

ReadonlyFontProps: TYPE = LONG POINTER TO READONLY FontPropsRecord;

FontPropsRecord: TYPE = RECORD [
    fontDesc: FontDescription,
    offset: INTEGER,
    foregroundBackground: ForegroundBackground,
    nUnderlines: CARDINAL,
    strikeout: BOOL,
    placement: Placement,
    toBeDeleted,
    revised: BOOL,
    width: Width,
    spare1: LONG CARDINAL];
```

63.2.4.1 FontDescription

```
FontDescription: TYPE = RECORD [
    family: Family,
    designVariant: DesignVariant,
    posture: Posture,
    weight: Weight,
    pointSize: CARDINAL,
    serifness: Serifness,
    spare1: LONG CARDINAL];

Family: TYPE = MACHINE DEPENDENT {
    century(0), frutiger(1), titan(2), pica(3), trojan(4), vintage(5), elite(6),
    letterGothic(7), master(8), cubic(9), roman(10), scientific(11), gothic(12),
    bold(13), ocrB(14), spokesman(15), xeroxLogo(16), centuryThin(17),
    scientificThin(18), helvetica(19), helveticaCondensed(20), optima(21),
    times(22), baskerville(23), spartan(24), bodoni(25), palatino(26),
    caledonia(27), memphis(28), excelsior(29), olympian(30), univers(31),
    universCondensed(32), trend(33), boxPS(34), terminal(35), ocrA(36), logo1(37),
    logo2(38), logo3(39), geneva2(40), times2(41), square3(42), courier(43),
    futura(44), prestige(45), aLLetterGothic(46), centurySchoolBook(47),
    firstUnused(48), lastUnused(510), backstop(511)};
```

family is the font family.

```
DesignVariant: TYPE = MACHINE DEPENDENT {
    null(0), roman, italic, firstAvailable, lastAvailable(255)};
```

designVariant specifies whether the character is roman or italic. null is not currently a valid value.

```
Posture: TYPE = MACHINE DEPENDENT {
    null(0), upright, slanted, backslanted, firstAvailable, lastAvailable(255)};
```

posture indicates the slant (stress) of the character, if any. null is not currently a valid value.

```
Weight: TYPE = MACHINE DEPENDENT {
    null(0), ultraLight, extraLight, light, semiLight, medium, semiBold, bold,
    extraBold, ultraBold, firstAvailable, lastAvailable(255)};
```

weight is the thickness of the character.

pointSize is the size of the font. Note that this value must be in the subrange [0..1023].

```
Serifness: TYPE = MACHINE DEPENDENT {
    null(0), serif, sansSerif, firstAvailable, lastAvailable(255)};
```

serifness indicates whether or not the character has serifs. **null** is not currently a valid value.

spare1 is for future use.

63.2.4.2 The other fields in FontProposRecord

offset is the offset of the character from the baseline.

```
ForegroundBackground: TYPE = MACHINE DEPENDENT {
    null(0), blackOnWhite, whiteOnBlack, firstAvailable, lastAvailable(255)};
```

foregroundBackground indicates the color of the character relative to the display.

nUnderlines indicates the number of times that the character is underlined; the value must be in the range [0..3].

strikeout indicates whether or not the character has been marked for deletion.

```
Placement: TYPE = MACHINE DEPENDENT{
    null(0), sub, subSub, subSuper, super, superSub, superSuper, userSpecified,
    firstAvailable, lastAvailable(255)};
```

placement indicates the position of the character relative to the line.

toBeDeleted indicates "normal" text that has been marked for deletion in Redlining mode.

revised indicates text that was typed in while Redlining was on but before finalizing the Redlined revisions.

```
Width: TYPE = MACHINE DEPENDENT {
    proportional(0), quarter, third, half, threeQuarters, full, firstAvailable,
    lastAvailable(255)};
```

width is used for Japanees text and should be set to **proportional** to get normal characters.

63.2.5 Font Runs

FontRuns are used to associate font properties with text. **XString** provides no facilities for associating font properties with text; **DocInterchangePropsDefs** allows the client to create font information structures that point into **XString** structures to make the association.

The data structures in this section mark font runs, which are consecutive characters with the same font. A **FontRunsRec** describes the font, while a cardinal value describes where the font starts in the text.

In addition, this interface allows the client to enumerate the font runs in a given **XString** body of text.

```
Run: TYPE = RECORD [
    props: FontPropsRecord,
    index: CARDINAL,
    context: XString.Context,
    spare1: LONG CARDINAL];
```

A **Run** indicates the beginning of a font run. **props** is the field describing the font used in the font run. **index** is the byte offset in the byte sequence that holds the text; it is the byte offset from the beginning of the byte sequence to the byte after the byte run. **context** is the **XString** context describing the next byte run. The context of the first byte run is contained in the reader body. See the next section for further explanation.

```
FontRuns: TYPE = LONG POINTER TO FontRunsRec;
```

```
FontRunsRec: TYPE = RECORD [
    length: CARDINAL,
    spare1: LONG CARDINAL,
    runs: SEQUENCE maxLength: CARDINAL OF Run];
```

FontRuns points to **FontRunsRec**, which is a record containing a sequence of **Runs**.

```
EnumerateFontRuns: PROC [
    r: XString.Reader,
    runs: FontRuns,
    proc: FontRunProc,
    clientData: LONG POINTER ← NIL]
    RETURNS [stopped: BOOL];
```

```
FontRunProc: TYPE = PROC [
    r: XString.Reader,
    fontProps: FontProps,
    clientData: LONG POINTER]
    RETURNS [stop: BOOL ← FALSE];
```

EnumerateFontRuns allows you to perform some action for each font run in an **XString.Reader**. **FontRunProc** is a call-back procedure that you pass to **EnumerateFontRuns**. If **FontRunProc** returns **stopped = TRUE**, the enumeration stops and **EnumerateFontRuns** returns **stopped = TRUE**. **clientData** is client defined data that you pass to **EnumerateFontRuns**, which passes it to **FontRunProc** every time **FontRunProc** is invoked.

63.2.5.1 Meaning of Index and Context Fields in Run

As stated earlier, **index** is the index into the XString of the byte following that run. **context** is the **XString.Context** in effect after that run. Here are two examples:

A ReaderBody with offset = 0, limit = 12, with bytes *abcdefghijkl*; font runs that describe the first three bytes as *fontA*, the next four as *fontB*, and the last five as *fontC* would be:

```
fontRun 0: [props: fontA, index: 3, context: ...]
fontRun 1: [props: fontB, index: 7, context: ...]
fontRun 2: [props: fontC, index: 12, context: ...]
```

A ReaderBody with offset = 7, limit = 19, with bytes *abcdefghijkl*; font runs that describe the first three bytes as *fontA*, the next four as *fontB*, and the last five as *fontC* would be:

```
fontRun 0: [props: fontA, index: 10, context: ...]
fontRun 1: [props: fontB, index: 14, context: ...]
fontRun 2: [props: fontC, index: 19, context: ...]
```

63.2.6 Paragraph Properties

The chief type in this section is the **ParaPropsRecord**, which describes the possible paragraph properties.

```
ParaProps: TYPE = LONG POINTER TO ParaPropsRecord;

ReadonlyParaProps: TYPE = LONG POINTER TO READONLY ParaPropsRecord;

ParaPropsRecord: TYPE = RECORD [
    basicProps: BasicPropsRecord,
    tabStops: TabStops,
    spare1: LONG CARDINAL];
```

basicProps describes all standard paragraph properties (those on the Paragraph property sheet); **tabStops** describes the current tab settings (the information on the Tab Settings property sheet).

The following sections describe the **BasicPropsRecord** and **TabStops** records in detail.

63.2.6.1 BasicPropsRecord

```
BasicProps: TYPE = LONG POINTER TO BasicPropsRecord;

ReadonlyBasicProps: TYPE = LONG POINTER TO READONLY BasicPropsRecord;

BasicPropsRecord: TYPE = RECORD [
    preLeading,
    postLeading,
    leftIndent,
    rightIndent,
    lineHeight: CARDINAL,
    paraAlignment: ParaAlignment,
    justified,
    hyphenated,
    keepWithNextPara: BOOL,
    language: MultiNational.Language,
    streakSuccession: StreakSuccession,
    defaultTabStopSpacing: DefaultTabStopSpacing,
    defaultTabStopAlignment: TabStopAlignment,
    spare1: LONG CARDINAL];
```

preLeading and **postLeading** are the spacing before and after the paragraph respectively; these values are in units of 1/72 inch.

leftIndent and **rightIndent** are the left and right paragraph margins; these values are in units of 1/72 inch.

lineHeight is the default line height for the paragraph; this value is in units of 1/72 inch.

```
ParaAlignment: TYPE = MACHINE DEPENDENT {
    left(0), center, right, firstAvailable, lastAvailable(255)};
```

paraAlignment indicates the alignment of the paragraph relative to the containing text column or text block.

justified when TRUE, causes the text in the paragraph to stretch to make a straight right edge.

hyphenated indicates whether the paragraph will be hypenated at the end of lines to improve justification. This parameter currently does nothing and should be set to false.

keepWithNextPara indicates whether the paragraph should always be on the same page as the succeeding paragraph.

language is the language for the paragraph; this information is used for formatting decimal tabs. It is also used when items are added to the paragraph (e.g., a field inherits the paragraph language when added to the paragraph).

```
StreakSuccession: TYPE = MACHINE DEPENDENT {
    leftToRight(0), rightToLeft, firstAvailable, lastAvailable(255)};
```

streakSuccession specifies whether a "streak" of characters should logically be read from left to right (e.g. English) or right to left (e.g. Hebrew).

```
TabStopOffset: TYPE = CARDINAL;
```

```
DefaultTabStopSpacing: TYPE = CARDINAL;
```

defaultTabStopSpacing is the default number of spaces between tabs.

```
TabStopAlignment: TYPE = MACHINE DEPENDENT {
    left(0), center, right, decimal, firstAvailable, lastAvailable(255)};
```

defaultTabStopAlignment is the default alignment for tabs: tabs can be relative to the left paragraph margin, the center of the paragraph, the right paragraph margin, or decimal points.

spare1 is for future use.

63.2.6.2 Tabs

```
TabStops: TYPE = LONG DESCRIPTOR FOR ARRAY OF TabStop;
```

```
TabStop: TYPE = RECORD [
    dotLeader: BOOLEAN,
    tabStopOffset: TabStopOffset,
    tabStopAlignment: TabStopAlignment,
    spare1: LONG CARDINAL];
```

tabStops describes the currently set tabs for the paragraph.

dotLeader indicates whether the tab has leader dots. tabStopOffset indicates the location of the tab, relative to the paragraph margin. tabStopAlignment indicates the alignment of the tab.

```
nTabsMax: CARDINAL = 100;
```

nTabsMax is the maximum number of tabs that there can be in a paragraph.

63.2.7 Mode Properties

Mode properties describe the commands in the document and auxillary menus of a ViewPoint document.

```
ModeProps: TYPE = LONG POINTER TO ModePropsRecord;

ReadonlyModeProps: TYPE = LONG POINTER TO READONLY ModePropsRecord;

ModePropsRecord: TYPE = RECORD [
    structureShowing,
    nonPrintingShowing,
    coverSheetShowing,
    promptFields: BOOL,
    spare1: LONG CARDINAL];
```

structureShowing, nonPrintingShowing, coverSheetShowing, and promptFields specify the appearance of the displayed document.

```
BooleanFalseDefault: TYPE = BOOL ← FALSE;

ModeSelections: TYPE = PACKED ARRAY ModeElements OF BooleanFalseDefault;

ModeElements: TYPE = {
    structureShowing, nonPrintingShowing, coverSheetShowing, promptFields,
    spare1, spare2, spare3, spare4, spare5, spare6, spare7, spare8};
```

ModeSelections are used to specify which ModeElements of a document should be acted upon.

63.2.7 Constants

The null*Props constants are declared so that clients may initialize property records with "neutral" properties. In most cases, each field value is the same as what would be set by the corresponding Get*PropsDefaults operation (sec 63.2.8).

```
nullFrameProps: FramePropsRecord = [
    borderStyle: solid,
    borderThickness: 2,
     frameDims: [72, 72],
     fixedWidth: FALSE,
     fixedHeight: FALSE,
     span: partialColumn,
     verticalAlignment: floating,
     horizontalAlignment: centered,
     topMarginHeight: 0,
     bottomMarginHeight: 0,
     leftMarginWidth: 0,
     rightMarginWidth: 0,
     spare1: 0];

nullPageProps: PagePropsRecord = [
     pageDims: [0, 0],
     topMarginHeight: 0,
     bottomMarginHeight: 0,
     leftMarginWidth: 0,
     rightMarginWidth: 0,
     startingPageSide: left,
     bindingMarginWidth: 0,
     nColumns: 1,
     balancedColumns: FALSE,
     unequalColumnWidths: FALSE,
     columnSpacing: 0,
     columnWidths: NIL,
     startingPageNumber: 1,
     pageNumberFormat: cardinal,
     restartPageNumbering: FALSE,
     startingLineNumber: 1,
     lineNumberInterval: 1,
     lineNumberFormat: cardinal,
     lineNumberLocation: leftMargin,
     headingStartsOnThisPage: TRUE,
     headingSameOnLeftRightPages: TRUE,
     footingStartsOnThisPage: TRUE,
     footingSameOnLeftRightPages: TRUE,
     spare1: 0];

nullColumnWidth: ColumnWidthRecord = [
     w: 0,
     spare1: 0];

nullFieldProps: FieldPropsRecord = [
     language: USEnglish,
     length: 0,
     required: FALSE,
     skipIf: never,
     stopOnSkip: FALSE,
     type: any,
     fillInRule: XString.nullReaderBody,
     description: XString.nullReaderBody,
     format: XString.nullReaderBody,
    name: XString.nullReaderBody,
    range: XString.nullReaderBody,
    skipIfField: XString.nullReaderBody,
    fillInRuleRuns: NIL,
    spare1: 0];

nullFontProps: FontPropsRecord = [
    fontDesc: nullFontDescription,
    offset: 0,
    foregroundBackground: blackOnWhite,
    nUnderlines: 0,
    strikeout: FALSE,
    placement: null,
    toBeDeleted: FALSE,
    revised: FALSE,
    width: proportional,
    spare1: 0];

nullFontDescription: FontDescription = [
    family: modern,
    designVariant: roman,
    posture: upright,
    weight: medium,
    pointSize: 12,
    serifness: sansSerif,
    spare1: 0];

nullRun: Run = [
    props: nullFontProps,
    index: 0,
    context: XString.unknownContext,
    spare1: 0];

classic: Family = century;

modern: Family = frutiger;

nullParaProps: ParaPropsRecord = [
    basicProps: nullBasicProps,
    tabStops: DESCRIPTOR[NIL, 0],
    spare1: 0];

nullBasicProps: BasicPropsRecord = [
    preLeading: 0,
    postLeading: 0,
    leftIndent: 0,
    rightIndent: 0,
    lineHeight: 12,
    paraAlignment: left,
    justified: FALSE,
    hyphenated: FALSE,
    keepWithNextPara: FALSE,
    language: USEnglish,
    streakSuccession: leftToRight,
    defaultTabStopSpacing: 18,
    defaultTabStopAlignment: left,
    spare1: 0];

nullTabStop: TabStop = [
    dotLeader: FALSE,
    tabStopOffset: 0,
    tabStopAlignment: left,
    spare1: 0];

nullModeProps: ModePropsRecord = [
    structureShowing: FALSE,
    nonPrintingShowing: FALSE,
    coverSheetShowing: FALSE,
    promptFields: FALSE,
    spare1: 0];
```

63.2.8 Default Properties

The **Get*PropsDefaults** procedures are called to obtain default values for property fields. These procedures differ from the constants in that they may obtain information from the user profile. Before calling any of these procedures, the client must declare a record of the appropriate type and pass its address to the **Get*PropsDefaults** procedure. None of these procedures allocate any additional data that the client would later have to free.

```
GetFramePropsDefaults: PROC [props: FrameProps];

GetPagePropsDefaults: PROC [props: PageProps];

GetFieldPropsDefaults: PROC [props: FieldProps];

GetFontPropsDefaults: PROC [props: FontProps];

GetParaPropsDefaults: PROC [props: ParaProps];

GetModePropsDefaults: PROC [props: ModeProps];

GetPageNumberDelimiter: PROC RETURNS [XChar.Character];
```

62.3 Index of Interface Items

Item	Page

64

GraphicsInterchangeDefs

64.1 Overview

GraphicsInterchangeDefs provides utilities for creating and enumerating the contents of anchored graphics frames. It is typically used in conjunction with **DocInterchangeDefs**.

64.1.1 Creating Graphics

To create new graphics, the client starts by calling **StartGraphics**, which initializes a graphics frame so that information can be added to it. This procedure returns a **Handle**, which is a pointer to an opaque type that contains, among other things, a graphics container. A graphics container is just an object that can contain graphic objects: a graphics container can be an anchored graphics frame, a nested graphics frame, a cusp button within a graphics frame, or another similar construct, such as a chart.

Once the client has a **Handle**, it can pass that **Handle** to various **Add*** routines to add new graphics objects, such as curves, rectangles, bitmaps, and text frames, to the graphics frame.

The client can also add nested frames, such as non-anchored graphics frames, cusp buttons, or graphics clusters, to the anchored frame. To create these structures, the client should call **StartGraphicsFrame, StartCuspButton**, or **StartCluster**, respectively. Each of these procedures takes a graphics container as a parameter, and returns another graphics handle. The client can then use this as the graphics container in other calls to **Add*** routines.

When everything has been added to a graphics container, the final step is to call a **Finish*** routine: **FinishGraphics**, **FinishButton**, **FinishGraphicsFrame**, or **FinishCluster**. **FinishGraphics** returns a graphics handle that can be passed to **DocInterchangeDefs**.

Thus, the scenario for creating a document with a floating graphics frame nested within an anchored graphics frame looks something like this:

1. Call **DocInterchangeDefs.StartCreation** to get a document handle (**doc**.)

2. Call **StartGraphics[doc]** to get an anchored frame handle (**h**).

3. Call **Add*[h]** to add graphics to the anchored frame.

4. Call **StartGraphicsFrame** to get a handle for a nested graphics frame (**gfh**).

5. Call **Add*[gfh]** to add graphics to the nested frame.

6. Call **FinishGrapicsFrame[gfh]** to finish the nested frame.

7. Call **FinishGraphics[h]** to complete the anchored frame and get an object of type DocInterchangeDefs.Instance (graphics).

8. Call DocInterchangeDefs.AppendAnchoredFrame[graphics].

9. Call DocInterchangeDefs.FinishCreation[@doc].

64.1.2 Reading Graphics

GraphicsInterchangeDefs also includes the facilities to read the contents of graphics frames. To read a graphics frame, the client should call **Enumerate**. **Enumerate** takes as parameters a graphics container, and a record of call back procedures, one for each of the following graphics objects: {bitmap frame, cusp button, cluster, curve, ellipse, form field, frame, image, line, point, rectange, text, triangle, other}.

Enumerate reads the contents of the graphics container, calling the appropriate procedure for each object that it encounters. If the client does not provide a procedure for a particular type of object, objects of that type will be ignored. Each of the client-supplied enumeration procedures can stop the enumeration if it so desires.

There are similar procedures to enumerate the contents of cusp buttons. **EnumerateButtonProgram** takes a button program and a record to handle the various objects that can be in a button program: new paragraphs and text.

64.2 Interface Items

64.2.1 Creating graphics

64.2.1.1 Start routines

To create new graphics objects, the client must first call **StartGraphics** to get an anchored frame handle.

```
StartGraphics: PROC [
   doc: DocInterchangeDefs.Doc]
   RETURNS [h: Handle];
```

StartGraphics creates a new graphics frame within **doc**.

```
Handle: TYPE = LONG POINTER TO Object;
Object: TYPE;
```

There are also similar routines to create nested frames within a graphics container. **StartCluster**, **StartGraphicsFrame**, and **StartButton** each initialize a nested frame within a graphics container. All **Start** routines return a **Handle**, which the client can then pass to the various **Add*** routines to add graphics to that graphics container.

```
StartCluster: PROC [h: Handle, box: Box] RETURNS [ch: Handle];
```

StartCluster initializes a set of graphics objects in **h**. **box** describes the size and location of the cluster relative to the anchored frame; **place** and **dims** are in micas.

```
Box: TYPE = RECORD [place: Place, dims: Dims];  -- micas

Place: TYPE = RECORD [x, y: LONG INTEGER];

Dims: TYPE = RECORD [w, h: LONG INTEGER];

StartGraphicsFrame: PROC [
    h: Handle,
    box: Box,
    frameProps: ReadonlyFrameProps,
    name, description: XString.Reader ← NIL,
    spareProps: LONG POINTER ← NIL,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        gfh: Handle,
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

StartGraphicsFrame initializes a nested graphics frame in **h**. **box** indicates the size and location of the nested frame relative to the graphics container; these values are in micas.

frameProps are the properties for the frame.

```
FrameProps: TYPE = LONG POINTER TO FramePropsRec;

ReadonlyFrameProps: TYPE = LONG POINTER TO READONLY FramePropsRec;

FramePropsRec: TYPE = RECORD [
    brush: Brush,
    fixedShape: BOOL,
    margins: ARRAY Side OF LONG CARDINAL,
    captionContent: ARRAY Side OF DocInterchangeDefs.Caption,
    spare1: LONG CARDINAL];

Brush: TYPE = RECORD [
    wthbrush: LONG CARDINAL,
    stylebrush: StyleBrush];
```

```
StyleBrush: TYPE = MACHINE DEPENDENT{
    invisible(0), solid(1), dashed(2), dotted(3), double(4), broken(5), (15)};
```

brush describes the properties of the lines that make up the frame. The brush width is in micas. The standard brush widths on the property sheet are 35, 71, 106, 141, 176 and 212.

fixedShape indicates whether the frame will expand in a uniform fashion.

margins are the frame margins, in points.

```
Side: TYPE = {top, bottom, left, right};
```

captionContent is an array of captions associated with the frame. Note that the **captionContent** parameter is only meaningful during enumeration, and not during **Start** or **Add*** routines, since the caption content is added after the frame is created.

spare1 is for future use.

name and **description** are the name and description of the graphics frame as it appears in the property sheet.

spareProps is for future use.

want*CaptionHandle indicates whether the client wants the frame to have the corresponding captions. If the client passes TRUE for one of these values, the corresponding return value will be non-NIL. The client can then use **DocInterchangeDefs** routines to add text to the caption. Note that the caption must eventually be freed with DocInterchangeDefs.ReleaseCaption.

gfh is a handle to the newly created graphics frame.

```
StartButton: PROC [
    h: Handle,
    box: Box,
    buttonProps: ReadonlyButtonProps,
    frameProps: ReadonlyFrameProps,
    wantProgramHandle,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        bfh: Handle,
        buttonProgram: ButtonProgram,
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

StartButton initializes a cusp button as a graphics container. **box** describes the size and location of the cusp button relative to the graphics container **h**.

```
ButtonProps: TYPE = LONG POINTER TO ButtonPropsRec;
```

```
ReadonlyButtonProps: TYPE = LONG POINTER TO READONLY ButtonPropsRec;

ButtonPropsRec: TYPE = RECORD [
    name: XString.Reader,
    spare1: LONG CARDINAL];

ButtonProgram: TYPE = LONG POINTER TO ButtonProgramObject;

ButtonProgramObject: TYPE;
```

buttonProps are the default properties for the button. If the client defaults this parameter, **StartButton** will generate a new unique name for the button.

wantProgramHandle specifies whether the client wants to be able to add to the button's program. If the client specifies TRUE, and is returned a valid button program handle, then it must later free that handle with a call to **ReleaseButtonProgram** (see section 64.2.1.3). **GraphicsInterchangeDefs** provides several procedures that the client can use to add data to the cusp program; see section, *Adding to a cusp button*, for information on these procedures.

All other properties are as described for **StartGraphicsFrame**.

64.2.1.2 Getting and Setting Frame Properties

```
SetExtraAnchoredFrameProps: PROC [
    doc: DocInterchangeDefs.Doc,
    anchoredFrame: DocInterchangeDefs.Instance,
    name, description: XString.Reader,
    spareProps: LONG POINTER ← NIL,
    selections: ExtraAnchoredFramePropsSelections];

ExtraAnchoredFramePropsSelections: TYPE = PACKED ARRAY
    ExtraAnchoredFramePropsElements OF DocInterchangePropsDefs.BooleanFalseDefault;

ExtraAnchoredFramePropsElements: TYPE = {name, description, spareProps};
```

The client can associate a name and description with an anchored frame by calling **SetExtraAnchoredFrameProps**. **doc** is the document that contains the anchored frame. **anchoredFrame** is the frame in which the client intends to add a name or description. **spareProps** are for future use and should be left defaulted to NIL. **selections** indicate which properties the client intends to add.

```
GetExtraAnchoredFrameProps: PROC [
    doc: DocInterchangeDefs.Doc,
    anchoredFrame: DocInterchangeDefs.Instance,
    spareProps: LONG POINTER ← NIL,
    zone: UNCOUNTED ZONE]
    RETURNS [name, description: XString.ReaderBody];
```

The **name** and **description** properties can be retrieved from an anchored frame with **GetExtraAnchoredFrameProps**. **spareProps** and **zone** are for future use. The returned values are not allocated from zone and so are read-only; the client should not attempt to call **XString.FreeReaderBytes** on them.

64.2.1.3 Adding information to a graphics container

After calling a **Start*** routine to initialize a graphics container, the client will typically call various **Add*** routines to add information to the graphics container. The **Add*** routines defined below add an object to the end of the list of objects in the specified graphics container.

The **Add*** routines are divided into two groups: geometrics, and frames. The geometrics section discusses how to add simple graphics objects: curves, ellipses, lines, points, rectangles, and triangles. The frames section describes how to add graphics objects in frames: bitmaps, form fields, images, and text frames. There is also a catch-all routine, **AddOther**, which provides the capability to add, as yet undefined graphics objects.

There are also routines that add textual information to a cusp button program or to a text frame. These routines are at the end of the section.

Geometrics

```
AddCurve: PROC [h: Handle, box: Box, curveProps: ReadonlyCurveProps];

CurveProps: TYPE = LONG POINTER TO CurvePropsRec;

ReadonlyCurveProps: TYPE = LONG POINTER TO READONLY CurvePropsRec;
CurvePropsRec: TYPE = RECORD [
    brush: Brush,
    lineEndNW: LineEnd,
    lineEndSE: LineEnd,
    lineEndHeadNW: LineEndHead,
    lineEndHeadSE: LineEndHead,
    direction: LineDirection,
    placeNW, placeApex, placeSE, placePeak: Place,
    fixedAngle: BOOL,
    spare1: LONG CARDINAL];

LineEnd: TYPE = MACHINE DEPENDENT {
    flush(0), square(1), round(2), arrow(3), (7)};

LineEndHead: TYPE = MACHINE DEPENDENT {
    none(0), h1(1), h2(2), h3(3), (15)};

LineDirection: TYPE = MACHINE DEPENDENT {
    WE(0), NS(1), NwSe(2), SwNe(3)};
```

AddCurve adds the curve described by **curveProps** to the specified graphics container. **box** specifies the location of the curve relative to the graphics frame. If **box.dims** is smaller than the curve, only that part of the curve that fits within **box.dims** will be displayed.

brush indicates the line properties of the curve; **brush** is as described earlier for **StartGraphicsFrame.**

lineEnd* describe the properties of the ends of the curve. **lineEndNW** describes the end that would paint first if the curve is traced clockwise; **lineEndSE** describes the end that would paint last tracing clockwise (Figure 64.1).

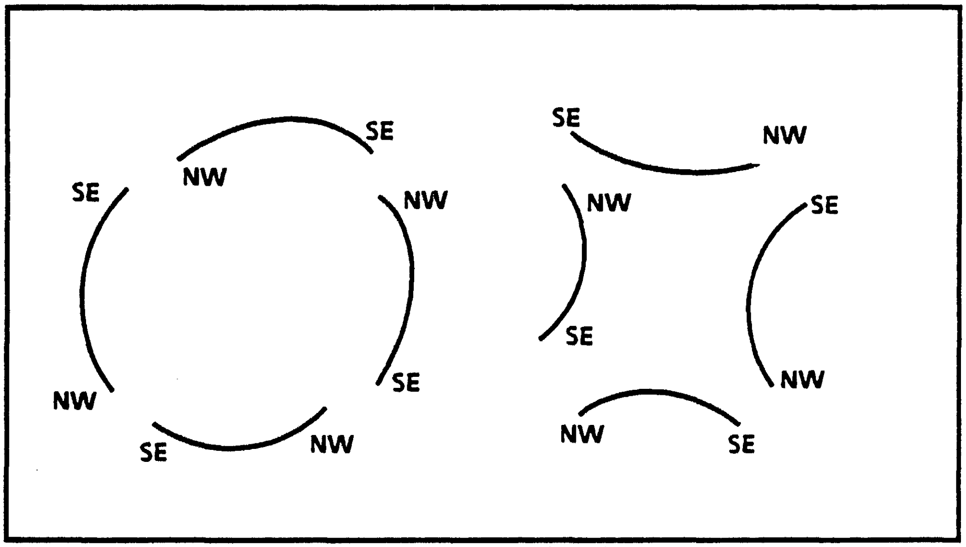


Figure 64.1 Curve Direction

If **lineEnd** = **arrow**, then **lineEndHead** describes the type of arrow: **h1** is the thinnest arrowhead; **h3** is the thickest as shown in Figure 64.2. If **lineEnd** ≠ **arrow**, then **lineEndHead** should be **none**.

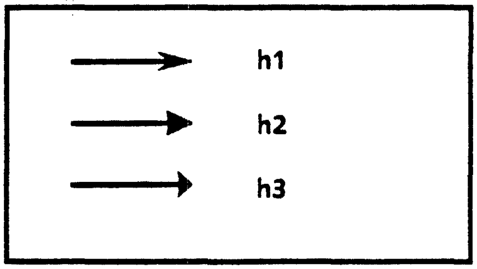


Figure 64.2 Arrowheads

direction is ignored; the client should always set this to **WE**.

place* defines the curve by specifying its endpoints, apex, and peak. These points are relative to **box**, and not the frame itself. Recall that curves paint clockwise; clients must ensure that the **NW** endpoint appears before the **SE** endpoint when tracing the curve clockwise. Figure 64.3 illustrates these four points for two different curves; the triangle marks the apex, the square marks the peak, and the circles mark the endpoints.

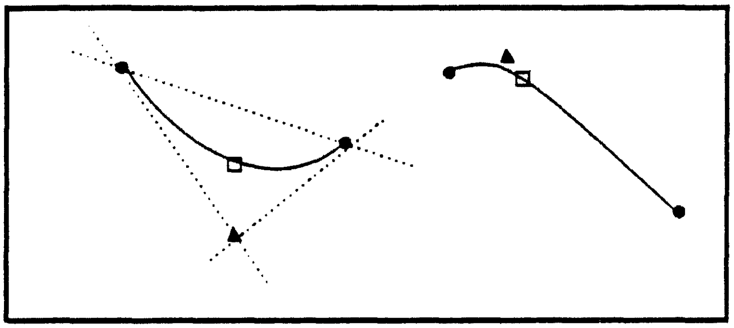


Figure 64.3 Defining curves

fixedAngle indicates that the curve will maintain its shape when grown or shrunk. **spare1** is for future use.

```
AddEccentricCurve: PROC [
    h: Handle,
    box: Box,
    eccentricCurveProps: ReadonlyEccentricCurveProps];

EccentricCurveProps: TYPE = LONG POINTER TO EccentricCurvePropsRec;

ReadonlyEccentricCurveProps: TYPE =
    LONG POINTER TO READONLY EccentricCurvePropsRec;

EccentricCurvePropsRec: TYPE = RECORD [
    brush: Brush,
    lineEndNW: LineEnd,
    lineEndSE: LineEnd,
    lineEndHeadNW: LineEndHead,
    lineEndHeadSE: LineEndHead,
    direction: LineDirection,
    placeNW, placeApex, placeSE: Place,
    eccentricity: CARDINAL,
    fixedAngle: BOOL,
    spare1: LONG CARDINAL];
```

AddEccentricCurve is just like **AddCurve** except that the curve is specified by its endpoints, apex, and eccentricity, rather than by endpoints, apex and peak.

eccentricity is a fraction represented by **eccentricity / LAST[CARDINAL]** . This allows the highest possible precision for eccentricities between 0 and 1.

```
AddEllipse: PROC [
    h: Handle,
    box: Box,
    ellipseProps: ReadonlyEllipseProps];

EllipseProps: TYPE = LONG POINTER TO EllipsePropsRec;
```

```
ReadonlyEllipseProps: TYPE = LONG POINTER TO READONLY EllipsePropsRec;

EllipsePropsRec: TYPE = RECORD [
    brush: Brush,
    shading: Shading,
    fixedShape: BOOL,
    spare1: LONG CARDINAL];
```

AddEllipse adds an ellipse to the specified graphics container. **box.dims** determine the size and shape of the ellipse; **box.place** determines its location relative to the frame **h**.

```
Shading: TYPE = RECORD [gray: Gray, textures: Textures];

Gray: TYPE = MACHINE DEPENDENT {
    none(0), gray25(1), gray50(2), gray75(3), black(4), (15)};

Textures: TYPE = PACKED ARRAY Texture OF BOOLEAN;

Texture: TYPE = MACHINE DEPENDENT {
    vertical(0), horizontal(1), nwse(2), swne(3), polkadot(4), (11)};
```

Within the **EllipseProps**, **brush** describes the ellipses' border, and **shading** describes its interior. The shading of the interior can be 25%, 50%, or 75% gray or solid black; the texture can be horizontal, vertical, or diagonal lines, or dots. **fixedShape** is the same for all shapes and indicates that the proportions of the object will not change when the user grows or shrinks it. **spare1** is for future use for all shapes.

```
AddLine: PROC [
    h: Handle,
    box: Box,
    lineProps: ReadonlyLineProps];

LineProps: TYPE = LONG POINTER TO LinePropsRec;

ReadonlyLineProps: TYPE = LONG POINTER TO READONLY LinePropsRec;

LinePropsRec: TYPE = RECORD [
    brush: Brush,
    lineEndNW: LineEnd,
    lineEndSE: LineEnd,
    lineEndHeadNW: LineEndHead,
    lineEndHeadSE: LineEndHead,
    direction: LineDirection,
    fixedAngle: BOOL,
    spare1: LONG CARDINAL];
```

AddLine adds a line to the graphics container at location **box.place**. **LineProps** are all as described above for curves.

```
AddPoint: PROC [
    h: Handle,
    box: Box,
    pointProps: ReadonlyPointProps];

PointProps: TYPE = LONG POINTER TO PointPropsRec;

ReadonlyPointProps: TYPE = LONG POINTER TO READONLY PointPropsRec;

PointPropsRec: TYPE = RECORD [
    wthbrush: LONG CARDINAL,
    pointStyle: PointStyle,
    pointFill: PointFill,
    spare1: LONG CARDINAL];

PointStyle: TYPE = MACHINE DEPENDENT {round(0), square(1), triangle(2), cross(3), (255)};

PointFill: TYPE = MACHINE DEPENDENT {solid(0), hollow(1), (255)};
```

AddPoint adds the point described by **oointProps** to the graphics container **h** at location **box.place**. **wthbrush** is in micas. **pointStyle** and **pointFill** are as shown in the Point object property sheet.

```
AddRectangle: PROC [
    h: Handle,
    box: Box,
    rectangleProps: ReadonlyRectangleProps];

RectangleProps: TYPE = LONG POINTER TO RectanglePropsRec;

ReadonlyRectangleProps: TYPE = LONG POINTER TO READONLY RectanglePropsRec;

RectanglePropsRec: TYPE = RECORD [
    brush: Brush,
    shading: Shading,
    fixedShape: BOOL,
    spare1: LONG CARDINAL];
```

AddRectangle adds the rectangle specified by **box.dims** to the graphics container at the location **box.place**. Rectangle properties are as described above for ellipses.

```
AddTriangle: PROC [
    h: Handle,
    box: Box,
    triangleProps: ReadonlyTriangleProps];

TriangleProps: TYPE = LONG POINTER TO TrianglePropsRec;

ReadonlyTriangleProps: TYPE =
    LONG POINTER TO READONLY TrianglePropsRec;

TrianglePropsRec: TYPE = RECORD [
    brush: Brush,
    shading: Shading,
    place1, place2, place3: Place, -- corners of triangle
    fixedShape: BOOL,
    spare1: LONG CARDINAL];
```

AddTriangle adds a triangle to the graphics conainter at location **box.place**. **brush** and **shading** are as described for ellipses; **place1**, **place2**, and **place3** are the corners of the triangle, relative to **box**.

Frame objects

The following **Add*** routines add various types of frame objects to the graphics container. Each of these routines has a parameter of type **FrameProps** that describes the frame, and **want*CaptionHandle** parameters that determine the captions for that frame. These parameters are as described in section 64.2.1.1, **StartGraphicsFrame.**

```
AddBitmap: PROC [
    h: Handle,
    box: Box,
    bitmapProps: ReadonlyBitmapProps,
    frameProps: ReadonlyFrameProps,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

bitmapProps describes a bitmap frame. **bitmapProps** largely correspond to the properties that the user sees in the property sheet.

```
BitmapProps: TYPE = LONG POINTER TO BitmapPropsRec;
```

```
ReadonlyBitmapProps: TYPE = LONG POINTER TO READONLY BitmapPropsRec;
```

```
BitmapPropsRec: TYPE = RECORD [
    opaque: BOOLEAN,
    xOffset, yOffset: LONG INTEGER,
    printFile: XString.ReaderBody,
    displaySource: BmDisplay,
    scalingProps: BitmapScalingProps,
    spare1: LONG CARDINAL];
```

opaque specifies whether the bitamp is opaque or transparent. **xOffset** and **yOffset** control the position of the bitmap within the bitmap frame. Setting both to 0 will position the bitmap flush in the upper left-hand corner. These values are in pixels.

printFile is the source for the bitmap to print. This is usually the same as the display source, but the client may specify a file name as an alternate print source if desired.

```
BmDisplay: TYPE = RECORD [
    SELECT type: * FROM
        internal = > [bm: LONG POINTER TO BitmapData],
        file = > [name: XString.ReaderBody],
    ENDCASE];
```

The source for the displayed bitmap is in one of two locations: either internal (the bits are copied into the document), or in a file on the desktop.

```
BitmapData: TYPE = RECORD [
    signature: INTEGER ← bmSignature, -- do not use any other value
    xScale: Interpress.Rational,
    yScale: Interpress.Rational,
    xDim: CARDINAL, -- # of bits wide
    yDim: CARDINAL, -- # of bits tall
    bpl: CARDINAL, -- Bits Per Line = ((xDim + 15) / 16) * 16
    pages: NSSegment.PageCount,
    bits: PACKED ARRAY [0..0) OF Environment.Byte];

bmSignature: INTEGER = 23456;
```

The actual bitmap is described by a **BitmapData** record. **signature** is a validity check for the bitmap. If a bitmap signature is anything but **bmSignature**, the implementation will not recognize it as a valid bitmap.

xScale and **yScale** specify the bitmap scale. At present, the only scale that is supported is 72 spots per inch, so the client should always set **xScale** and **yScale** to [254, 720,000]. (The default unit for an **Interpress.Rational** is meters; converting inches to meters yields 720,000 spots per 254 meters, since 1 inch = 2.54 cms.)

xDim and **yDim** describe the size of the bitmap. **bpl** is the width of the bitmap, rounded to the nearest word boundary. The client should ensure that the bitmap's x dimension is equal to **bpl**.

pages is the number of pages that the bitmap occupies, and **bits** is the actual bitmap.

```
BitmapScalingProps: TYPE = RECORD [
    SELECT type: * FROM
    printerResolution = > [resolution: CARDINAL],
    fixed = > [
        horizontalAlignment: {center, right, left},
        verticalAlignment: {center, bottom, top},
        scalingPercentage: CARDINAL [0..1024)],
    automatic = > [shape: {similar, fillUp}],
    other = > [spare1: PACKED ARRAY [2..15) OF [0..1], spare2: CARDINAL],
    ENDCASE];
```

scalingProps specifies one of the 3 bitmap scaling modes: **automatic**, **fixed** or **printerResolution**. The client will generally default the mode to **automatic** with **shape = similar**; this ensures that the bitmap will be automatically magnified/shrunk to fit just inside the bitmap frame until either the vertical or horizontal edge reaches the frame's edge. If **fillUp** is specified, the vertical and horizontal scaling factors are individually determined so that the bitmap completely fills the frame.

The **fixed** mode requires the client to control the bitmap's alignment (by Alignment parameters) and scaling (by Scale parameter). The **scalingPercentage** allows the client to shrink or magnify the bitmap. A **scalingPercentage** value of 100 means that the bitmap should be displayed and printed the same size as the original. A value of 50 means that the bitmap is shrunk to one half both vertically and horizontally. **scalingPercentage** must be in the range 1 - 1000.

printerResolution indicates the resolution of the printer; typical values are: 72, 75, 150, 200, and 300. Other values can be specified and must be the number of spots per inch.

```
AddFormField: PROC [
    h: Handle,
    box: Box,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    frameProps: ReadonlyFrameProps,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    expandRight, expandBottom: BOOL ← FALSE,
    wantFieldHandle,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        field: DocInterchangeDefs.Field,
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

AddFormField adds the specified field to **h** at the location **box**.

If the client specifies **wantFieldHandle** = TRUE, **AddFormField** will return a **DocInterchangeDefs.Field**; the client must eventually free this field with a call to **DocInterchangeDefs.ReleaseField**. To add information to the field, the client should use the facilities of **DocInterchangeDefs**.

```
AddImage: PROC [
    h: Handle,
    box: Box,
    imageProps: ReadonlyImageProps,
    frameProps: ReadonlyFrameProps,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

```
ImageProps: TYPE = LONG POINTER TO ImagePropsRec;
```

```
ReadonlyImageProps: TYPE = LONG POINTER TO READONLY ImagePropsRec;
```

```
ImagePropsRec: TYPE = RECORD [
    name: XString.Reader,
    spare1: LONG CARDINAL];
```

AddImage adds an image frame to the specified graphics container.

```
AddTextFrame: PROC [
    h: Handle,
    box: Box,
    frameProps: ReadonlyFrameProps,
    textFrameProps: ReadonlyTextFrameProps,
    wantTextHandle,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        text: TextInterchangeDefs.Text,
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchange.Caption];
```

```
TextFrameProps: TYPE = LONG POINTER TO TextFramePropsRec;
```

```
ReadonlyTextFrameProps: TYPE =
    LONG POINTER TO READONLY TextFramePropsRec;
```

```
TextFramePropsRec: TYPE = RECORD [
    expandRight, expandBottom, transparent: BOOL,
    tFrameProps: TextInterchangeDefs.TFramePropsRec,
    spare1: LONG CARDINAL];
```

AddTextFrame adds a text frame to the specified graphics container. If the client specifies wantTextHandle = TRUE, it will return a handle to a text frame.

Adding to a cusp button

The following routines allow the client to add textual information to a cusp button program.

```
AppendCharToButtonProgram: PROC [
    to: ButtonProgram,
    char: XChar.Character,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];
```

Add a character to the button program. nToAppend is the number of copies of the character to be added; fontProps are the properties of the character.

```
AppendNewParagraphToButtonProgram: PROC [
    to: ButtonProgram,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
nToAppend: CARDINAL ← 1];
```

Add a new paragraph character with specified properties to the button program.

```
AppendTextToButtonProgram: PROC [
    to: ButtonProgram,
    text: XString.Reader,
    textEndContext: XString.Context,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];
```

Add a string with specified properties to button program. For efficiency, the client should include **textEndContext** if known.

Adding miscellaneous graphics

```
AddOther: PROC [
    h: Handle,
    box: Box,
    instance:DocInterchangeDefs.Instance];
```

AddOther is provided to allow addition of charts and other as yet undefined objects. For information on charts, see **ChartDataInstallDefs**.

64.2.1.3 Release routines

```
ReleaseButtonProgram: PROC [
    bpPtr: LONG POINTER TO ButtonProgram];
```

ReleaseButtonProgram releases the handles obtained from **AddButtonProgram**. Like Mesa's FREE operator, this routine take a pointer to the object to be freed, and sets the handle itself to NIL. Thus, after a call to **ReleaseButtonProgram**, **ButtonProgram** will be NIL.

64.2.1.4 Finish routines

When everything has been added to a graphics container, the client should call a **Finish** routine.

```
FinishButton: PROC [bfh: Handle];
```

```
FinishCluster: PROC [ch: Handle];
```

```
FinishGraphics: PROC [h: Handle]
    RETURNS [graphics: DocInterchangeDefs.Instance];
```

```
FinishGraphicsFrame: PROC [gfh: Handle];
```

bfh, **ch**, **h**, and **gfh** are the handles obtained from the corresponding **Start** routines. The client will typically pass the **DocInterchangeDefs.Instance** returned by **FinishGraphics** to **DocInterchangeDefs.AppendAnchoredFrame**.

64.2.2 Reading graphics

To read the contents of a graphics frame, the client should call **Enumerate**. **Enumerate** takes as parameters a graphics container and a list of call back procedures, one for each of the kinds of items that might be in the graphics container. **Enumerate** will proceed through the graphics container, calling the appropriate procedure for each item that it encounters.

Each enumeration procedure takes parameters that describe the properties of the object. These properties are temporary, and will be destroyed after the client procedure returns. If the client wishes to save any of these properties, it must explicitly copy them.

Client **EnumProcs** do not need to call any **Release*** routines on anything passed them as a parameter. The Enumerator always releases containers after calling each **EnumProc**.

In the case of a cusp button, cluster, or nested graphics frame, the client can recursively call **Enumerate** to get the contents of the nested frame. There are also related enumeraters, **TextInterchangeDefs.EnumerateText** and **EnumerateButtonProgram**, that enumerate the contents of a text frame and a cusp button, respectively.

```
Enumerate: PROC [
    doc: DocInterchangeDefs.Doc,
    graphicsContainer: DocInterchangeDefs.Instance,
    procs: EnumProcs,
    clientData: LONG POINTER ← NIL]
    RETURNS [dataSkipped: BOOLEAN];

EnumProcs: TYPE = LONG POINTER TO EnumProcsRecord;

EnumProcsRecord: TYPE = RECORD [
    bitmapProc: BitmapProc ← NIL,
    buttonProc: ButtonProc ← NIL,
    clusterProc: ClusterProc ← NIL,
    curveProc: CurveProc ← NIL,
    ellipseProc: EllipseProc ← NIL,
    formFieldProc: FormFieldProc ← NIL,
    frameProc: FrameProc ← NIL,
    imageProc: ImageProc ← NIL,
    lineProc: LineProc ← NIL,
    otherProc: OtherProc ← NIL,
    pointProc: PointProc ← NIL,
    rectangleProc: RectangleProc ← NIL,
    textFrameProc: TextFrameProc ← NIL,
    triangleProc: TriangleProc ← NIL];

BitmapProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    bitmapProps: ReadonlyBitmapProps,
    frameProps: ReadonlyFrameProps]
    RETURNS [stop: BOOLEAN ← FALSE];
```

```
ButtonProc: TYPE = PROC[
    clientData: LONG POINTER,
    graphicsContainer:DocInterchangeDefs.Instance,
    box: Box,
    buttonProps: ReadonlyButtonProps,
    frameProps: ReadonlyFrameProps,
    buttonProgram: ButtonProgram]
    RETURNS [stop: BOOLEAN ← FALSE];

ClusterProc: TYPE = PROC[
    clientData: LONG POINTER,
    graphicsContainer: DocInterchangeDefs.Instance,
    box: Box]
    RETURNS [stop: BOOLEAN ← FALSE];

CurveProc: TYPE = PROC[
    clientData: LONG POINTER,
    box: Box,
    curveProps: ReadonlyCurveProps]
    RETURNS [stop: BOOLEAN ← FALSE];

EllipseProc: TYPE = PROC[
    clientData: LONG POINTER,
    box: Box,
    ellipseProps: ReadonlyEllipseProps]
    RETURNS [stop: BOOLEAN ← FALSE];

FormFieldProc: TYPE = PROC[
    clientData: LONG POINTER, box: Box,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    frameProps: ReadonlyFrameProps,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    expandRight, expandBottom: BOOL,
    content: DocInterchangeDefs.Field]
    RETURNS [stop: BOOLEAN ← FALSE];

FrameProc: TYPE = PROC[
    clientData: LONG POINTER,
    graphicsContainer: DocInterchangeDefs.Instance,
    box: Box,
    frameProps: ReadonlyFrameProps,
    name, description: XString.Reader,
    spareProps: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

ImageProc: TYPE = PROC[
    clientData: LONG POINTER,
    box: Box,
    imageProps: ReadonlyImageProps,
        frameProps: ReadonlyFrameProps]
        RETURNS [stop: BOOLEAN ← FALSE];

    LineProc: TYPE = PROC [
        clientData: LONG POINTER,
        box: Box,
        lineProps: ReadonlyLineProps]
        RETURNS [stop: BOOLEAN ← FALSE];

    OtherProc: TYPE = PROC [
        clientData: LONG POINTER,
        box: Box,
        instance: DocInterchangeDefs.Instance,
        objectType: OtherObjectType]
        RETURNS [stop: BOOLEAN ← FALSE];

    OtherObjectType: TYPE = MACHINE DEPENDENT {
        illusFrame(0), barchart, linechart, piechart, pieslice, table, equation,
        firstAvailable, lastAvailable(255)};

    PointProc: TYPE = PROC [
        clientData: LONG POINTER,
        box: Box,
        pointProps: ReadonlyPointProps]
        RETURNS [stop: BOOLEAN ← FALSE];

    RectangleProc: TYPE = PROC [
        clientData: LONG POINTER,
        box: Box,
        rectangleProps: ReadonlyRectangleProps]
        RETURNS [stop: BOOLEAN ← FALSE];

    TextFrameProc: TYPE = PROC [
        clientData: LONG POINTER,
        box: Box,
        frameProps: ReadonlyFrameProps,
        textFrameProps: ReadonlyTextFrameProps,
        content: TextInterchangeDefs.Text]
        RETURNS [stop: BOOLEAN ← FALSE];

    TriangleProc: TYPE = PROC [
        clientData: LONG POINTER,
        box: Box,
        triangleProps: ReadonlyTriangleProps]
        RETURNS [stop: BOOLEAN ← FALSE];
```

Enumerating button programs

```
    EnumerateButtonProgram: PROC [
        buttonProgram: ButtonProgram,
        procs: ButtonProgramEnumProcs,
    clientData: LONG POINTER ← NIL]
    RETURNS [dataSkipped: BOOLEAN];

ButtonProgramEnumProcs: TYPE =
    LONG POINTER TO ButtonProgramEnumProcsRecord;

ButtonProgramEnumProcsRecord: TYPE = RECORD [
    newParagraphProc: DocInterchangeDefs.NewParagraphProc ← NIL,
    textProc: DocInterchangeDefs.TextProc ← NIL];
```

EnumerateButtonProgram enumerates the contents of buttonProgram, calling the client-supplied procs as appropriate. clientData is passed to each of the call-back procedures during enumeration.

64.2.3 Constants

```
nullBitmapProps: BitmapPropsRec = [
    opaque: TRUE,
    xOffset: 0,
    yOffset: 0,
    printFile: XString.nullReaderBody,
    displaySource: nullBmDisplay,
    scalingProps: nullBitmapScalingProps,
    spare1: 0];
    nullBmDisplay: BmDisplay = [internal[bm: NIL]];

nullBitmapScalingProps: BitmapScalingProps = [automatic[shape: similar]];

nullButtonProps: ButtonPropsRec = [name: NIL, spare1: 0];

nullCurveProps: CurvePropsRec = [
    brush: [0, solid],
    lineEndNW: flush,
    lineEndSE: flush,
    lineEndHeadNW: none,
    lineEndHeadSE: none,
    direction: WE,
    placeNW: [0, 0],
    placeApex: [0, 0],
    placeSE: [0, 0],
    placePeak: [0, 0],
    fixedAngle: FALSE,
    spare1: 0];

nullEccentricCurveProps: EccentricCurvePropsRec = [
    brush: [0, solid],
    lineEndNW: flush,
    lineEndSE: flush,
    lineEndHeadNW: none,
    lineEndHeadSE: none,
    direction: WE,
    placeNW: [0, 0],
    placeApex: [0, 0],
    placeSE: [0, 0],
    eccentricity: 0,
    fixedAngle: FALSE,
    spare1: 0];

nullEllipseProps: EllipsePropsRec = [
    brush: [0, solid],
    shading: [none, ALL[FALSE]],
    fixedShape: FALSE,
    spare1: 0];

nullFrameProps: FramePropsRec = [
    brush: [0, solid],
    fixedShape: FALSE,
    margins: ALL[0],
    captionContent: ALL[NIL],
    spare1: 0];

nullImageProps: ImagePropsRec = [name: NIL, spare1: 0];

nullLineProps: LinePropsRec = [
    brush: [0, solid],
    lineEndNW: flush,
    lineEndSE: flush,
    lineEndHeadNW: none,
    lineEndHeadSE: none,
    direction: WE,
    fixedAngle: FALSE,
    spare1: 0];

nullPointProps: PointPropsRec = [
    wthbrush: 0,
    pointStyle: round,
    pointFill: solid,
    spare1: 0];

nullRectangleProps: RectanglePropsRec = [
    brush: [0, solid],
    shading: [none, ALL[FALSE]],
    fixedShape: FALSE,
    spare1: 0];

nullTextFrameProps: TextFramePropsRec = [
    expandRight: FALSE,
    expandBottom: FALSE,
    transparent: FALSE,
    tFrameProps: TextInterchangeDefs.nullTFrameProps,
    spare1: 0];

nullTriangleProps: TrianglePropsRec = [
    brush: [0, solid],
shading: [none, ALL[FALSE]],
place1: [0, 0],
place2: [0, 0],
place3: [0, 0],
fixedShape: FALSE,
spare1: 0];
```

68.3 Index of Interface Items

65

TableInterchangeDefs

65.1 Overview

TableInterchangeDefs allows clients to read the contents of a table, create a new table, or add information to an existing table. This interface should be used in conjunction with **DocInterchangeDefs**.

A table is described by three sets of properties: table properties, column properties, and row properties. Table properties include the name of the table, a description of table headers and the number of columns and rows in the table; column properties include whether the columns are divided, and the alignment of text within the columns; and row properties include information about how the text is aligned within a given row. The actual content of a table is included with the row information.

65.1.1 Table building

To create a new table, the client should start by calling **StartTable**. This procedure takes table properties and column properties as parameters, and returns a table handle. **Handle** points to **Object**, which is a record that contains, along with table-related data, a pointer to the actual table content (See section 65.2.1.1 Diagram of Table Structure, Fig. 65.2). Initially, the row properties have default values and the table has no content; the client should initialize row properties and content after the call to **StartTable**.

To add content to the table, the client can pass the table handle to **AppendRow**, which adds new information to the table. When all of the rows have been added, the final step is to call **FinishTable**, which creates the final structure for the table. Once the table is created, the client can pass this table to the procedures in **DocInterchangeDefs** to append it to a document.

FinishTable returns an **DocInterchangeDefs.Instance** for the table, which the client can pass to **DocInterchangeDefs.AppendAnchoredFrame**.

To add information to an existing table, the client should call **StartExistingTable** instead of **StartTable**. This procedure also returns a table handle, which the client can then pass to **AppendRow** and **FinishTable**. **StartExistingTable** takes an **DocInterchangeDefs.Instance** as a parameter; the client will typically call TableSelectionDefs.TableFromSelection to get the currently selected table as a value of type DocInterchangeDefs.Instance.

65.1.2 Table reading

To read the contents of a table, the client typically starts by calling **Enumerate**. **Enumerate** takes as arguments a table object (**DocInterchangeDefs.Instance**) and a record of three call back procedures: a **TableProc**, a **ColumnsProc**, and a **RowProc**.

Enumerate will call the **TableProc** and the **ColumnsProc** once for a given table; these procedures obtain the table and column properties. Since the content of the table is stored with the rows, **Enumerate** will call the **RowProc** once for each row in the table.

There is a also a procedure **EnumerateSpecificRows**, which is just like **Enumerate** except that it enumerates a specific list of rows within a table rather than the entire table. **EnumerateSpecificRows** will call the **RowsProc** once for each row in the specified range of rows.

65.2 Interface Items

65.2.1 Table building operations

65.2.1.1 Creating a new table

```
StartTable: PROC [
    doc: DocInterchangeDefs.Doc,
    props: TableProps,
    c: ColumnInfo]
    RETURNS [h: Handle];
```

StartTable creates a document table in **doc**. **props** describes the properties of the table itself; c describes the properties of the columns. The **Handle** that is returned contains a description of row properties and table content.

The following sections describe **TableProps**, **ColumnInfo**, and **Handles** in detail.

StartTable can raise **Error[documentFull]** if the table and header row will not fit in the document. If **StartTable** raises this error, the table cannot be added to the document due to lack of space.

Table properties

A **TablePropsRec** describes the properties of a table and its headers.

```
TableProps: TYPE = LONG POINTER TO TablePropsRec;
```

```
TablePropsRec: TYPE = RECORD [
    name: XString.Reader,
    fillinByRow,
    fixedRows,
    fixedColumns: BOOL,
    numberOfColumns,
    numberOfRows: NATURAL,
    visibleHeader,
    repeatHeader,
    repeatTopCaption,
    repeatBottomCaption: BOOL,
    borderLine,
    dividerLine: Line,
    horizontalAlignment: HeaderAlignment,
    headerVerticalAlignment: VerticalAlignment,
    topHeaderMargin, bottomHeaderMargin: LONG CARDINAL,
    sortKeys: SortKeys,
    spare1: LONG CARDINAL];
```

name is the name of the table.

fillinByRow determines what happens when the user presses the NEXT key. If **fillInByRow** is TRUE, pressing the NEXT key advances through the table one row at a time, and the table is expanded by rows. In this case, the number of columns is fixed and the number of rows can be either fixed or varying. If **fillinByRow** is FALSE, then pressing the NEXT key advances through the table one column at a time, and the table is expanded by columns. In this case, the number of rows is fixed and the number of columns can be either fixed or varying. **fixedRows** and **fixedColumns** indicate whether the user can change the number of rows and columns in the table.

numberOfColumns and **numberOfRows** are used as hints for **StartTable**.

visibleHeader indicates whether there should be a visible header at the top of the table; **repeatHeader**, **repeatTopCaption**, **repeatBottomCaption** indicates whether or not to repeat these items on every page if the table occupies multiple pages.

borderLine describes the table border (not the frame border), and **dividerLine** describes the line between the header row and the rest of the table. In tables, a line is either solid or invisible; a solid line can have a width anywhere from one pixel to six pixels.

```
Line: TYPE = RECORD [
    linestyle: Linestyle,
    linewidth: Linewidth];
```

```
Linestyle: TYPE = MACHINE DEPENDENT{
    none(0), solid, dashed, dotted, double, broken, firstAvailable,lastAvailable(255)};
```

```
Linewidth: TYPE = MACHINE DEPENDENT {w1(0), w2(1), w3(2), w4(3), w5(4), w6(5)};
```

horizontalAlignment and **headerVerticalAlignment** specify the alignment of the text within a header.

```
HeaderAlignment: TYPE = HorizontalAlignment [left..right];
```

```
HorizontalAlignment: TYPE = MACHINE DEPENDENT{left(0), center(1), right(2), decimal(3)};
```

```
VerticalAlignment: TYPE = MACHINE DEPENDENT {flushtop(0), centered(1), flushbottom(2)};
```

topHeaderMargin and bottomHeaderMargin specify the amount of white space that should appear between above and below each header element.

```
SortKeys: TYPE = LONG POINTER TO SortKeysRec;

SortKeysRec: TYPE = RECORD [
    length: CARDINAL,
    spare1: LONG CARDINAL,
    keys: SEQUENCE maxLength: CARDINAL OF SortKey];

SortKey: TYPE = RECORD [
    columnName: XString.Reader,
    sortOrder: XString.SortOrder,
    ascending: BOOL,
    spare1: LONG CARDINAL];
```

The SortKeysRec contains a sequence of optional SortKeys for a table or column. A column must be divided-repeating in order to have sort keys. Each SortKey contains the column's name, its sortOrder and whether to sort in ascending or descending order. Spare1 is for future use.

Column properties

```
ColumnInfo: TYPE = LONG POINTER TO ColumnInfoSeq;

ColumnInfoSeq: TYPE = RECORD [SEQUENCE length: CARDINAL OF ColumnInfoRec];

ColumnInfoRec: TYPE = RECORD [
    headerEntryRec: HeaderEntryRec,
    name, description: XString.Reader,
    divided: BOOL,
    subcolumns: NATURAL,
    repeating: BOOL,
    subcolumnInfo: ColumnInfo,
    alignment: HorizontalAlignment,
    tabOffset, -- Micas! (different from DocInterchangePropsDefs.TabStop)
    width,
    leftMargin,
    rightMargin: LONG CARDINAL,
    type: DocInterchangePropsDefs.FieldChoiceType,
    required: BOOL,
    language: MultiNational.Language,
    format: XString.Reader,
    stopOnSkip: BOOL,
    range: XString.Reader,
    length: CARDINAL,
    skipText: XString.Reader,
    skipChoice: DocInterchangePropsDefs.SkipIfChoiceType,
    fillin: XString.Reader,
    fillinRuns: DocInterchangePropsDefs.FontRuns,
    line: Line,
    sortKeys: SortKeys,
    spare1: LONG CARDINAL];
```

A **ColumnInfoSeq** describes all the columns of a table; a **ColumnInfoRec** describes one column in detail. Within a **ColumnInfoRec**, the most complicated field is a **headerEntryRec**; all of the other fields correspond directly to the fields on the property sheet that the user sees. The next section, discusses the header properties; and the section, Other column properties, discusses the remaining column properties.

For a more complete description of any of these properties, see the user documentation.

Column header properties

```
HeaderEntryRec: TYPE = RECORD [
    subHeaders: HeaderInfo,
    line: Line,
    singleLineHint: BOOL,
    spare1: LONG CARDINAL,
    content: EntryContent];
```

A **HeaderEntryRec** describes the textual content of a column header. Header text can contain only one font and one set of paragraph properties per column header.

subHeader describes the headers for each of the subcolumns. This field is only interesting if the column is **divided**. **subHeader** points to a sequence that contains a HeaderEntryRec for each subcolumn. Each subcolumn may in turn be subdivided, in which case that subcolumn's **HeaderEntryRec subHeader** field would point to another sequence.

```
HeaderInfo: TYPE = LONG POINTER TO HeaderInfoSeq;
```

```
HeaderInfoSeq: TYPE = RECORD [SEQUENCE length: CARDINAL OF HeaderEntryRec];
```

line describes the properties of line that divides the header from subheaders; **line** is only visible if the column is subdivided.

singleLineHint is a hint that the header only contains one line of text; this makes the code slightly faster by simplifying the calculation of header size.

spare1 is for future use.

```
EntryContent: TYPE = RECORD [
    SELECT mode: * FROM
        read = > [obtainTextProc: ObtainTextProc, obtainTextData: ObtainTextData],
        write = > [fillInTextProc: FillInTextProc ←NIL, clientData: LONG POINTER ←NIL],
    ENDCASE];
```

```
ObtainTextData: TYPE[4];
```

```
ObtainTextProc: TYPE = PROC [obtainTextData: ObtainTextData]
    RETURNS [text: TextInterchangeDefs.Text];
```

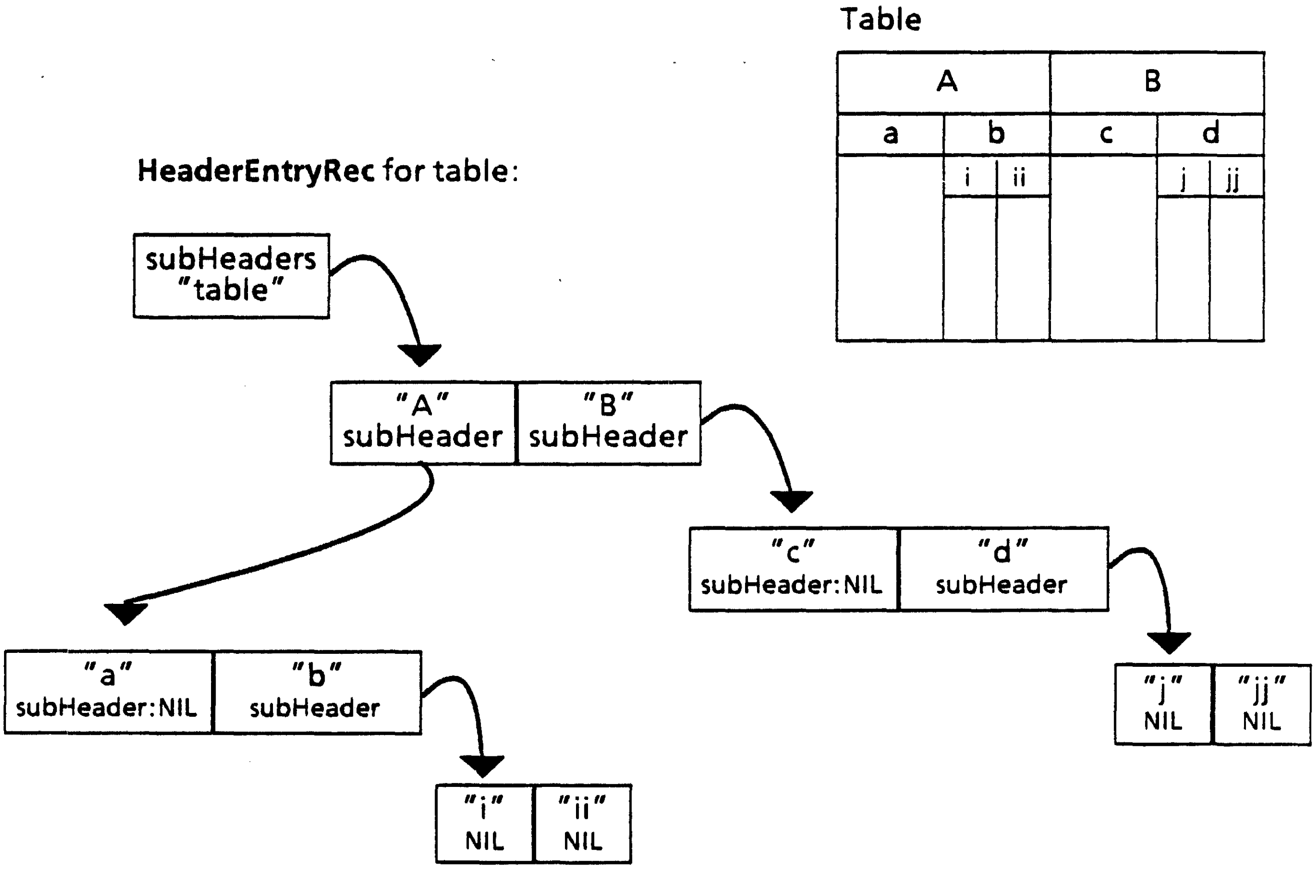


Figure 65.1 Table and **HeaderEntryRec** for table

FillInTextProc: TYPE = PROC [**text**: TextInterchangeDefs.Text, **clientData**: LONG POINTER];

content is a varaint record that describes the content for a header entry. When enumerating a table, all the header entries will be of the form [**read**[...]]. The client may call the **ObtainTextProc** for an entry to obtain a TextInterchangeDefs.**Text** object, which may then be enumerated via TextInterchangeDefs.**EnumerateText**.

When creating a table, the client must set all header entries to [**write**[...]]. The client may set the **fillInTextProc** to a proc to be called back to fill in the entry with text. **clientData** will be passed to the client's **fillInTextProc**. The client may default the **fillInTextProc** to NIL so that the entry will be empty.

Other column properties

name and **description** are the name and description of the table as it appears in the property sheet.

divided specifies whether the columns can be divided. **subcolumns** is the number of subcolumns; **repeating** indicates that subcolumns can have subrows, and **subcolumnInfo**

is the recursive description of the subcolumns. **subcolumns**, **repeating**, and **subcolumnInfo** are ignored if divided = FALSE.

alignment describes the alignment of the text within a column.

tabOffset specifies where the tabs should be set, relative to the margin. **tabOffset** only applies if **alignment** = **decimal**; it is in micas.

width is the width of the column; **leftMargin** and **rightMargin** are the margins for the column. These values are also in micas.

type indicates the type of content that will appear in a column.

required indicates that the entry is required, and that the user must fill it in before proceeding to another entry in the table.

language affects the format of date and amount fields. It is used when items are added to the paragraph (e.g., a field inherits the paragraph language when added to the paragraph).

format allows the user to define a format to which the data in the table must conform.

stopOnSkip When the user presses SKIP, the skipping action shoudl stop at the entry that has this option.

range is used to define a specific range of acceptable entries. Once defined, any entry not within the defined range is not acceptable. See the user documentation for information on how ranges are defined.

length allows the user to define the maxiumum number of characters that will be accepted in the column entries.

skipText and **skipChoice** defines the conditions under which an area may be skipped when the user presses NEXT. See the user documentation for more detail.

fillin and **fillinRuns** describe the fill-in rules for completing the table.

line describes the properties of vertical line to right of column.

sortKeys describes the optional sort keys for the column.

spare1 is for future use.

Return values

StartTable returns a handle:

```
Handle: TYPE = LONG POINTER TO Object;

Object: TYPE = RECORD [
    zone: UNCOUNTED ZONE,
    table: DocInterchangeDefs.Instance,
    tableHeight: LONG CARDINAL,
    tableWidth: LONG CARDINAL,
    rc: RowContent,
    spare1: LONG CARDINAL,
    private: ARRAY [0..0) OF WORD];
```

zone is the zone from which dynamic storage specific to this operation is allocated. **table** is the table itself.

tableHeight is initially equal to the height of the header row and is updated after each call to **AppendRow**. **tableHeight** and **tableWidth** are in micas. **rc** points to a record used as temporary storage for a new row.

```
RowContent: TYPE = LONG POINTER TO RowContentSeq;
```

```
RowContentSeq: TYPE = RECORD [
    topMargin, bottomMargin: LONG CARDINAL ← 0,
    line: Line ← [solid, w2],
    verticalAlignment: VerticalAlignment ← flushtop,
    spare1: LONG CARDINAL ← 0,
    rowdata: SEQUENCE length: CARDINAL OF RowEntryRec];
```

RowContentSeq describes row properties and content. The margins are the row margins; **line** is the properties of the line separating the rows. **verticalAlignment** specifies the alignment of text within a row. **spare1** is for future use. **rowdata** describes the content.

```
RowEntryRec: TYPE = RECORD [
    subRows: SubRows,
    singleLineHint: BOOL,
    spare1: LONG CARDINAL,
    content: EntryContent];
```

A **RowEntryRec** describes the textual content of a given row entry.

```
SubRows: TYPE = LONG POINTER TO SubRowsRec;
```

```
SubRowsRec: TYPE = RECORD [
    length: CARDINAL,
    spare1: LONG CARDINAL ← 0,
    rows: SEQUENCE maxLength: CARDINAL OF RowContent];
```

SubRowsRec describes subrow properties and content. If **subRows** is non-NIL, then the rest of the **RowEntryRec** record is unused, since the information will be in the individual subrow records.

Note that subrows may only exist if the parent column is divided.

The remaining fields are as described for header properties.

65.2.1.2 Opening an existing table

```
StartExistingTable: PROC [
    table: DocInterchangeDefs.Instance,
hi: HeaderInfo ← NIL,
rowPropsSource: NATURAL ← 0,
deleteExistingRows: BOOL ← TRUE,
numberOfRowsHint: NATURAL ← 0]
RETURNS [h: Handle];
```

StartExistingTable sets things up to append rows to an existing table. **table** is the table object. The **table** passed in to **StartExistingTable** is often obtained from a call to **TableSelectionDefs.TableFromSelection**, which returns the current selection as a table.

hi describes the desired properties for the table headers. If **hi** = NIL then the existing column headers are used.

rowPropsSource is the index of a row in the table; this is the row from which the default properties should be taken; the rows are numbered from [0..n].The horizontal alignment for each entry is taken from first new paragraph character in corresponding element of first row.

deleteExistingRows indicates whether the implementation should delete the existing contents of the table before adding new information. **numberOfRowsHint** is a hint about the number of rows that the table will contain; this is for efficiency purposes.

Like **StartTable**, **StartExistingTable** returns a **Handle**, which the client can then pass to **AppendRow**.

This procedure will raise **Error[tableNotEditable]** if the document is read-only.

65.2.1.3 Appending rows

```
AppendRow: PROC [h: Handle, rc: RowContent];
```

AppendRow adds the row described by **rc** to the table described by **h**. Typically, **h** will be a handle obtained from either **StartTable** or **StartExistingTable**.

RowContent is as described in section 65.2.1.1, return values for **StartTable**.

65.2.1.4 Finishing a table

```
FinishTable: PROC [h: Handle]
    RETURNS [
    table: DocInterchangeDefs.Instance,
    tableWidth, tableHeight: LONG CARDINAL];
```

The client should call **FinishTable** when it is through editing a table. The **table** that is returned is intended to be passed as the **content** argument to **DocInterchangeDefs.AppendFrame**. This operation deletes **h.zone**. **tableWidth** and **tableHeight** are in micas.

65.2.1.5 Miscellaneous utilities

```
MaxTableElements: PROC RETURNS [NATURAL];
```

This procedure returns the maximum size of the table, allowing clients to be smarter about large tables.

```
DefaultFontProps: PROC [font: DocInterchangePropsDefs.FontProps];
```

```
DefaultParaProps: PROC [para: DocInterchangePropsDefs.ParaProps];
```

These procedures take a properties record and fill in reasonable default values.

```
GetTablePropsFromName: PROC [
    doc: DocInterchangeDefs.Doc,
    tableName: XString.Reader,
    tableProps: TableProps,
    zone: UNCOUNTED ZONE];
```

doc is the document from which to retrieve the properties of the table specified by tableName.

tableProps.name will be NIL.but the remainder of tableProps will contain the table's properties.

If tableProps.sortKeys is not NIL, it will be allocated from zone.

65.2.2 Table reading operations

```
EnumerateTable: PROC [
    table: DocInterchangeDefs.Instance,
    procs: EnumProcs,
    clientData: LONG POINTER ← NIL];
```

```
EnumProcs: TYPE = LONG POINTER TO EnumProcsRec;
```

```
EnumProcsRec: TYPE = RECORD [
    tableProc: TableProc ← NIL,
    columnsProc: ColumnsProc ← NIL,
    rowProc: RowProc ← NIL];
```

To parse the contents of a table, clients call Enumerate or EnumerateSpecificRows. Enumerate takes as parameters a table handle and a record of callback procedures: one for table properties, one for column properties, and one for row properties.

```
TableProc: TYPE = PROC [
    clientData: LONG POINTER,
    props: TableProps]
    RETURNS [stop: BOOL ← FALSE];
```

```
ColumnsProc: TYPE = PROC [
    clientData: LONG POINTER,
    columns: ColumnInfo]
    RETURNS [stop: BOOL ← FALSE];

RowProc: TYPE = PROC [
    clientData: LONG POINTER,
    content: RowContent]
    RETURNS [stop: BOOL ← FALSE];
```

Enumerate calls the **TableProc** and the **ColumsnProc** once, passing in the appropriate property information. Because the content of the table is stored with the rows, **Enumerate** calls the **rowProc** once for each row in the table.

Each of these callback procedures has a boolean return value. If **stop** is ever TRUE, then the enumeration will stop.

```
EnumerateSpecificRows: PROC [
    tr: TableRows,
    procs: EnumProcs,
    clientData: LONG POINTER ← NIL];

TableRows: TYPE = RECORD [
    table, firstRow, lastRow: DocInterchangeDefs.Instance];
```

EnumerateSpecificRows describes a certain subset of rows in a table. As with **Enumerate**, the **tableProc** and the **columnsProc** will each be called once to describe the appropriate properties; the column information will describe the columns intersecting the described rows. The **RowProc** will be called once for each row in **[firstRow..lastRow]**.

```
tableRowsNil: TableRows = [
    DocInterchangeDefs.instanceNil,
    DocInterchangeDefs.instanceNil,
    DocInterchangeDefs.instanceNil];
```

tableRowsNil specifies a default for **TableRows**. **tableRowsNil** is what you get if a table is not selected in a call to **EnumericSpecificRows**.

65.2.3 Diagram of table structure

Figure 65.2 is a diagram of a **table** structure. **RowContent** is a pointer to **RowContentSeq**; **table** is a record that contains two pointers to the actual instance of the table. (Note that **table** itself is not a pointer.)

65.2.4 Constants

The following constants can be used to initialize the various table properties to reasonable default values.

```
nullLine: Line = [linestyle: solid, linewidth: w1];

nullSortKey: SortKey = [
    columnName: NIL,
    sortOrder: standard,
```

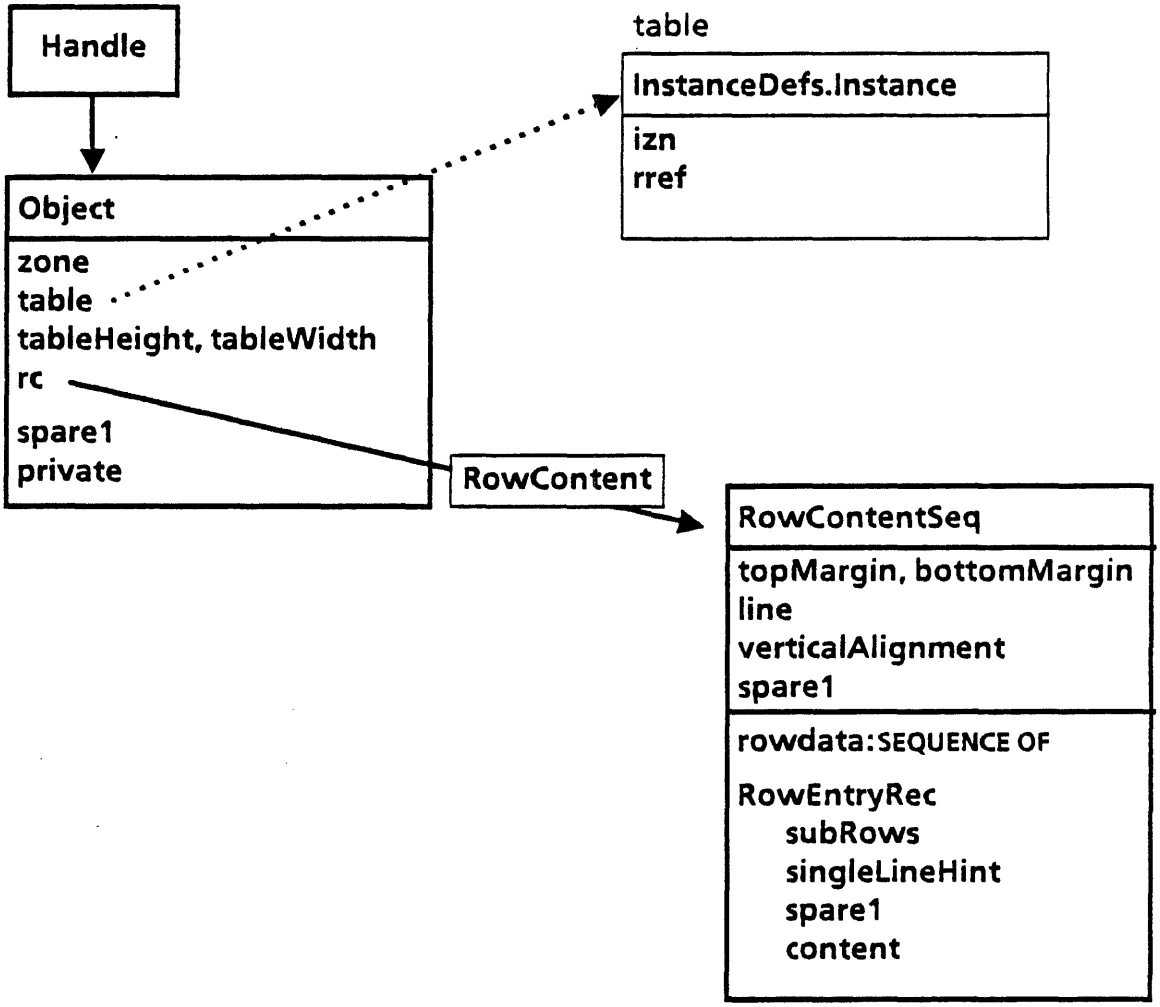


Figure 65.2 Diagram of Table Structure

```
    ascending: TRUE,
    spare1: 0];

nullColumnInfo: ColumnInfoRec = [
    headerEntryRec: nullHeaderEntry,
    name: NIL,
    description: NIL,
    divided: FALSE,
    subcolumns: 0,
    repeating: FALSE,
    subcolumnInfo: NIL,
    alignment: center,
    tabOffset: 0,
    width: 2540,
    leftMargin: 0,
    rightMargin: 0,
    type: any,
    required: FALSE,
    language: USEnglish,
    format: NIL,
    stopOnSkip: FALSE,
    range: NIL,
      length: 0,
    skipText: NIL,
    skipChoice: empty,
    fillin: NIL,
    fillinRuns: NIL,
    line: [solid, w2],
    sortKeys: NIL,
    spare1: 0];

nullHeaderEntry: HeaderEntryRec = [
    subHeaders: NIL,
    line: [solid, w2],
    singleLineHint: FALSE,
    spare1: 0,
    content: [write[]]];

nullRowEntry: RowEntryRec = [
    subRows: NIL,
    singleLineHint: FALSE,
    spare1: 0,
    content: [write[]]];

nullTableProps: TablePropsRec = [
    name: NIL,
    fillinByRow: TRUE,
    fixedRows: FALSE,
    fixedColumns: TRUE,
    numberOfColumns: 0,
    numberOfRows: 0,
    visibleHeader: TRUE,
    repeatHeader: TRUE,
    repeatTopCaption: TRUE,
    repeatBottomCaption: TRUE,
    borderLine: [none, w1],
    dividerLine: [solid, w4],
    horizontalAlignment: center,
    headerVerticalAlignment: centered,
    topHeaderMargin: 0,
    bottomHeaderMargin: 0,
    sortKeys: NIL,
    spare1: 0];
```

65.2.5 Errors

```
TableError: SIGNAL [type: ErrorType];
```

```
ErrorType: TYPE = MACHINE DEPENDENT{
    tableTooWide, tableTooTall, tableHeaderTooTall, firstAvailable, lastAvailable(255)};
```

tableTooWide — **StartTable** will raise this error if the specified table is too wide to fit in the document.

tableTooTall — **AppendRow** will raise this error if the specified table is too tall to fit in the document.

tableHeaderTooTall **StartTable** will raise this error if the specified headers are too tall.

65.3 Usage/Examples

Here is an example of a basic program that runs from the Attention Menu. It registers two commands: Make Table, which creates a new document with a table, and Add To Table, which adds four new rows to the selected table.

```
DIRECTORY
...;

TableExample: PROGRAM
    IMPORTSTableInterchangeDefs, TableSelectionDefs, TextInterchangeDefs, ... = {



tableWidth: CARDINAL = 1600;        --micas
headerMargin: CARDINAL = 35 * 9;    --micas; margin should be 9 pixels
rowMargin: CARDINAL = 100;

<<Menu Proc for Create Table command. Creates new document, creates new table,
appends table to document, and then adds document to desktop.>>
MakeDocument: MenuData.MenuProc = {
    rows, columns: CARDINAL ← 3; -- arbitrary
    doc: DocInterchangeDefs.Doc ← DocInterchangeDefs.StartCreation[
        paginateOption: none].doc;
    table: DocInterchangeDefs.Instance = BuildSimpleTable[doc, rows, columns];
    props: DocInterchangePropsDefs.FramePropsRecord ←
        DocInterchangePropsDefs.nullFrameProps;
    props.frameDims ← [tableWidth, tableWidth];
    [ ] ← DocInterchangeDefs.AppendAnchoredFrame[
        to: doc,
        type: table,
        anchoredFrameProps: @props,
        content: table];
    AddFileToDeskTop[doc]; --
};
```

```
<<Create table inside doc with specified number of rows and columns. The content will
be the string "abc." >>
BuildSimpleTable: PROC [doc: DocInterchangeDefs.Doc, rows, columns: CARDINAL]
RETURNS [table: DocInterchangeDefs.Instance ← DocInterchangeDefs.instanceNil] = {
    h: TableInterchangeDefs.Handle;
    contentString: XString.ReaderBody ← XString.FromSTRING["abc"L];
    c: TableInterchangeDefs.ColumnInfo ← Heap.systemZone.NEW[
        TableInterchangeDefs.ColumnInfoSeq[columns]];
    props: TableInterchangeDefs.TablePropsRec ← [
        name: NIL,
        fillinByRow: TRUE,
        fixedRows: FALSE,
        fixedColumns: TRUE,
        numberOfColumns: columns,
        numberOfRows: rows,
        visibleHeader: TRUE,
        repeatHeader: TRUE,
        repeatTopCaption: TRUE,
        repeatBottomCaption: TRUE,
        borderLine: [none, w1],
        dividerLine: [solid, w4],
        horizontalAlignment: center,
        headerVerticalAlignment: centered,
        topHeaderMargin: headerMargin,
        bottomHeaderMargin: headerMargin,
        sortKeys: NIL,
        spare1: 0];
    FOR i: CARDINAL IN [0..columns) DO
        c[i] ← TableInterchangeDefs.nullColumnInfo;
        c[i].width ← tableWidth;
    ENDLOOP;
    -- start creating table
    h ← TableInterchangeDefs.StartTable[doc: doc, props: @props, c: c];
    Heap.systemZone.FREE[@c];
    -- set row props and content
    h.rc.topMargin ← rowMargin;
    h.rc.bottomMargin ← rowMargin;
    FOR i: CARDINAL IN [0..rows) DO
        FOR j: CARDINAL IN [0..columns) DO
            h.rc[j] ← [
                subRows: NIL,
                singleLineHint: FALSE,
                spare1: 0,
                content: [write[fillInTextProc: FillInText,clientData: @contentString]] ];
        ENDLOOP;
        TableInterchangeDefs.AppendRow[h, h.rc];
    ENDLOOP;
    RETURN [TableInterchangeDefs.FinishTable[h].table];
};
```

```
<<call-back procedure that writes text into the Text field of the table. The text to write
is specified by the clientData argument.>>
FillInText: TableInterchangeDefs.FillInTextProc = {
<<PROC [text: TextInterchangeDefs.Text, clientData: LONG POINTER]; >>
    r: XString.Reader ← NARROW[clientData, XString.Reader];
    TextInterchangeDefs.AppendTextToText[
        to: text,
        text: r,
        textEndContext: XString.unknownContext];
};

AddFileToDeskTop: PROC[doc: DocInterchangeDefs.Doc] = {
    docFile: NSFile.Handle ← DocInterchangeDefs.FinishCreation[@doc].docFile;
    refDoc: NSFile.Reference = NSFile.GetReference[docFile];
    refDt: NSFile.Reference = StarDesktop.GetCurrentDesktopFile[];
    fileDt: NSFile.Handle = NSFile.OpenByReference[refDt];
    NSFile.Move[docFile, fileDt];
    NSFile.Close[fileDt];
    NSFile.Close[docFile];
    StarDesktop.AddReferenceToDesktop[refDoc]
};

<<Menu Proc for Add To Table command. Just adds four new blank rows to selected
table.>>
AddToTable: MenuData.MenuProc = {
    h: TableInterchangeDefs.Handle ← NIL;
    table: DocInterchangeDefs.Instance =
    TableSelectionDefs.TableFromSelection[];
    --if current selection is not a table, then return. Otherwise,
    --add new rows to it. If doc is not editable, then return.
    IF table = DocInterchangeDefs.instanceNil THEN RETURN
    ELSE {
        h ← TableInterchangeDefs.StartExistingTable[
            table: table, deleteExistingRows: FALSE -- catch error if doc is not editable
            ! TableInterchangeDefs.TableError => GOTO Exit];
        THROUGH [0..4) DO
            TableInterchangeDefs.AppendRow[h, h.rc];
         ENDLOOP;
    };
    [] ← TableInterchangeDefs.FinishTable[h];
    EXITS Exit => NULL;
};
```

```
Init: PROC = {
    makeTable: XString.ReaderBody ← XString.FromSTRING["MakeTable"L];
    addToTable: XString.ReaderBody ← XString.FromSTRING["AddToTable"L];
    Attention.AddMenuItem[
        MenuData.CreateItem[
            zone: Heap.systemZone,
            name: @makeTable,
            proc: MakeDocument]];
    Attention.AddMenuItem[
        MenuData.CreateItem[
            zone: Heap.systemZone,
            name: @addToTable,
            proc: AddToTable]];
};

Init[];

}.
```

71.4 Index of Interface Items

66

TableSelectionDefs

66.1 Overview

TableSelectionDefs provides procedures to obtain the current selection as a table, or a selection of rows within a table. This interface is meant to be used in conjunction with TableInterchangeDefs.

66.2 Interface Items

```
TableFromSelection: PROC RETURNS [DocInterchangeDefs.Instance];
```

TableFromSelection returns the current selection as an object of type InstanceDefs.Instance. The client will typically pass this value to TableInterchangeDefs.StartExistingTable.

```
TableRowsFromSelection: PROC RETURNS [tr: TableInterchangeDefs.TableRows];
```

TableRowsFromSelection returns the current selection as a series of rows in a table. The client will typically pass this value as a parameter to TableInterchangeDefs.EnumerateSpecificRows.

```
tableTarget: Selection.Target;

tableRowsTarget: Selection.Target;

TableFromValue: PROC [v: Selection.Value]
  RETURNS [DocInterchangeDefs.Instance];

TableRowsFromValue: PROC [v: Selection.Value]
  RETURNS [tr: TableInterchangeDefs.TableRows];
```

tableTarget, tableRowsTarget, TableFromValue and TableRowsFromValue are not currently implemented.

```
GetHostDocAccess: PROC [instance: DocInterchangeDefs.Instance]
  RETURNS [Access];

Access: TYPE = MACHINE DEPENDENT {readOnly(0), readWrite, (255)};
```

Access specifies whether or not the document is in edit mode.

66.3 Index of Interface Items

67

TextInterchangeDefs

67.1 Overview

TextInterchangeDefs provides procedures to create and enumerate text that resides in locations that **DocInterchangeDefs** does not define. This interface is used by **GraphicsInterchangeDefs** to handle the content of nested text frames and by **TableInterchangeDefs** to handle the content of header and row entries. **TextInterchangeDefs** also provides procedures to handle the content of anchored text frames. All dimensions are in micas.

67.2 Interface Items

67.2.1 Data types

The basic data structure of **TextInterchangeDefs** is **Text**, which is a pointer to an opaque text containing object.

```
Text: TYPE = LONG POINTER TO TextObject;
```

```
TextObject: TYPE;
```

TFrameProps specify the properties of anchored text frames. Appending and enumerating text frames is covered later in this chapter.

```
TFrameProps: TYPE = LONG POINTER TO TFramePropsRec;
```

```
ReadonlyTFrameProps: TYPE = LONG POINTER TO READONLY TFramePropsRec;
```

```
TFramePropsRec: TYPE = RECORD [
    innerMargin: LONG CARDINAL,
    name, description: XString.Reader,
    spare1: LONG CARDINAL];
```

innerMargin is the uniform spacing between the text and the frame border. The client can vary the **innerMargin** from 0 to 7 pixels. This feature enables footnote frames that begin and end at the column margins.

name and **description** are the name and description of the table as it appears in the property sheet.

spare1 is intended for future use.

```
nullTFrameProps: TFramePropsRec = [
innerMargin: 141,
name: NIL,
description: NIL,
spare1: 0];
```

nullTFrameProps provides default initialization values for the **TFramePropsRec**

67.2.2 Creating an Anchored Text Frame

StartTextInAnchoredFrame is used to begin appending text to the body of an anchored text frame. After an anchored text frame has been appended to a document via **DocInterchangeDefs.AppendAnchoredFrame, StartTextInAnchoredFrame** may be called to permit text to be appended to its body.

```
StartTextInAnchoredFrame: PROC [
doc:DocInterchangeDefs.Doc,
anchoredFrame: DocInterchangeDefs.Instance,
props: ReadonlyTFrameProps]
RETURNS [text: Text];
```

doc is the document containing the new anchored text frame.

anchoredFrame is the **DocInterchangeDefs.Instance** returned by the call to **DocInterchangeDefs.AppendAnchoredFrame.**

67.2.3 Append Operations

The following **append*** procedures are similar to the ones found in **DocInterchangeDefs**; the only difference is that these procedures append to text objects obtained from **StartTextInAnchoredFrame**.

```
AppendCharToText: PROC [
    to: Text,
    char: XChar.Character,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];

AppendFieldToText: PROC [
    to: Text,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [field:DocInterchangeDefs.Field];

AppendNewParagraphToText: PROC [
    to: Text,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];

AppendTextToText: PROC [
    to: Text,
    text: XString.Reader,
    textEndContext: XString.Context,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];

AppendTileToText: PROC [
    to: Text,
    type: Atom.ATOM,
    data: LONG POINTER ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [tile: DocInterchangeDefs.Tile];
```

67.2.4 Enumeration

To extract the content of a text object, clients call EnumerateText.

```
EnumerateText: PROC [
    text: Text,
    procs: TextEnumProcs,
    clientData: LONG POINTER ← NIL]
    RETURNS [dataSkipped: BOOLEAN];
```

procs is a pointer to a record containing client defined call-back procedures; these enumerate the various kinds of structures that can be found in text.

clientData is a client defined argument that will be passed to each of the call-back procedures.

```
TextEnumProcs: TYPE = LONG POINTER TO TextEnumProcsRecord;

TextEnumProcsRecord: TYPE = RECORD [
    fieldProc: DocInterchangePropsDefs.FieldProc ← NIL,
    newParagraphProc: DocInterchangePropsDefs.NewParagraphProc ← NIL,
    textProc: DocInterchangePropsDefs.TextProc ← NIL,
    tileProc: DocInterchangePropsDefs.TileProc ← NIL];
```

TextForAnchoredFrame is used when a client wants to enumerate the body of an anchored text frame.

```
TextForAnchoredFrame: PROC [
    doc: DocInterchangePropsDefs.Doc,
    content: DocInterchangePropsDefs.Instance,
    props: TFrameProps]
    RETURNS [text: Text];
```

doc is the document containing the anchored text frame to enumerate and content is the value passed to the DocInterchangeDefs.AnchoredFrameProc. The text object that is returned may be passed to **EnumerateText**. After enumerating the text, the client must call **ReleaseText** on the text object returned by **TextForAnchoredFrame**.

TextForAnchoredFrame fills in **props** with text specific properties of the frame.

67.2.5 Releasing Text

The client should release the text object after calling **StartTextInAnchoredFrame** or **TextForAnchoredFrame**.

```
ReleaseText: PROC [textPtr: LONG POINTER TO Text];
```

67.3 Example

The proper sequence of calls to append an anchored text frame having content is:

```
props: TextInterchangeDefs.TFramePropsRec ← [...];
[anchoredFrame, ...] ← DocInterchangeDefs.AppendAnchoredFrame[to: doc, type: text, ...];
text ← TextInterchangeDefs.StartText[doc, anchoredFrame, @props];
TextInterchangeDefs.AppendTextToText[text, reader, ...];
TextInterchangeDefs.ReleaseText[@text];
```

It is not mandatory that the client call **StartTextInAnchoredFrame** after appending an anchored frame. Failing to call **StartTextInAnchoredFrame** simply means that the anchored text frame will be empty, except for one new paragraph character that has default paragraph and font properties. Note that if the client does not call **StartTextInAnchoredFrame**, then the client should not call **ReleaseText**.

67.3 Index of Interface Items

Item **Page**