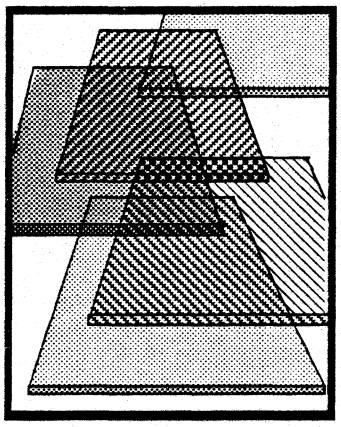
XEROX

Xerox Development Environment



Mesa Course

Version 12.0
September 1985
610E00230

Office Systems Division
Development Systems
Xerox Corporation
2400 Geng Road
Palo Alto, California 94303

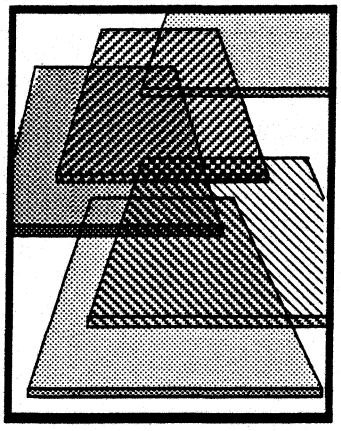


Table of contents

9 Variant records

10 Processes and monitors

11 Introduction to Tajo

12 The Exec interface

13 MFile

14 MSegment

15 Streams

16 The FormSWLayout Tool

17 Tool window interfaces

18 Tool building

19 Multiple instance tools

20 Terminal interface package (TIP)

21 Creating subwindows

Appendices

A Correcting compilation errors

B Setting breakpoints

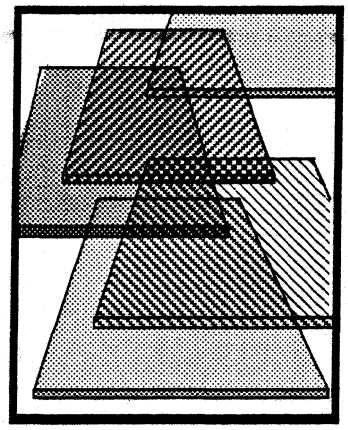
C Translating uncaught signals

D Debugging an address fault

E Answers to questions

F Trainer information

G Glossary



Introduction

The *Mesa Course* is a self-paced programming tutorial intended to give you hands-on experience with applications and systems programming in the Xerox Development Environment. The course introduces important concepts, illustrates those concepts with extensive examples, and provides exercises to ensure your familiarity with those concepts. The Mesa Course is intended for use at any XDE customer site.

The twenty one chapters of the Mesa Course are grouped into two major sections: the Mesa Language and the "Tajo" development environment. The experienced professional need only skim the Mesa Language chapters and can begin with serious study of the development environment, referring to language issues in the first section as required. The less experienced programmer should work through the material sequentially. The initial section of the course is designed to present Mesa programming to someone who is familiar with other structured languages, particularly Pascal, and has completed the *Introduction to XDE* on-line tutorials.

The Mesa Language section introduces you to Mesa programming concepts and essential components of the Xerox Development Environment. You will learn how to develop and run programs in our environment, including how to:

- convert standard Pascal constructs into their Mesa counterparts,
- use Mesa's interface mechanism to integrate independently developed programs and share information among them,
- allocate dynamic storage from a common pool,
- declare and manipulate strings, dynamic arrays, and variant records
- use processes and monitors effectively,
- handle exception occurrences via a software interrupt mechanism,
- debug your program when things go awry, and
- use the Mesa reference manuals to find the information you need.

Upon completing the first section you should have a well-grounded understanding of how to use Mesa and the development environment.

The last half of the course emphasizes advanced features of XDE and concentrates on fundamental aspects of tool creation. In this section you will learn how to

- write programs that run in the Executive window,
- interact with the Mesa file system including performing file I/O and attaching a stream to a file,
- allocate space from virtual memory and map it to a backing file,
- use the form subwindow layout tool to generate "standard" tool subwindow implementation code,
- implement tool features not provided by the form subwindow layout tool,
- handle terminal input for a tool, and
- paint into the windows of a tool

If you do not intend to be an active Mesa programmer, then this course is probably not for you. The *Introduction to XDE* on-line tutorials provide an explanation of the non-programming aspects of the development environment, and may be what you want.

Course structure

The course consists of twenty one chapters, six appendices, and a Glossary. The early chapters, Chapters 1 through 10, each concentrate on a single concept and build on the previous chapters. If this material is appropriate for your experience level, you should study each of these in order. The chapters of the environment section, from Chapter 11 on, are somewhat more independent and self-standing. Chapter 12 deals with the Executive, chapters 13 through 15 deal with aspects of the file system, chapters 16 through 19 cover fundamental aspects of tool construction, and chapters 20 and 21 discuss gathering input for tools and painting tool windows.

Some of the appendices cover basic debugging techniques. The remaining appendices, answers to questions, and the Glossary should be referenced as needed. The course suggests points when studying the appendices might be most helpful to you.

How to read a chapter

For the most part, each chapter contains the following sections in the following order:

- An *introduction* covering what it is about, what you will learn from it, and what you will do in it.
- A description of *preliminary readings* and where to find them. These are usually the sections in the reference documentation that describe the concepts to be discussed. You should read, but not disect, this information. We discuss the depth to which you should study these readings in the next section, Using the Course.
- A *glossary* of terms, which defines the terms new to that particular section.

- A *discussion* of the chapter's main topic. This section is the main body of the chapter. It usually takes the form of a general introduction to the concept, a discussion of the facilities you need, and at least one programming example.

- A *summary* of what you have learned. This helps you to check quickly that you have understood the major points of the chapter, and can later serve as a reference.

- A discussion of *style*-related issues related to the concept being learned. The section explains the choice and type of coding style used in the examples.

- A description of *reference materials* and where to find them. These are usually collected journal articles that relate to the concept being taught. Using these materials will extend the breadth of your knowledge or give you a different perspective on the topic.

- A set of *questions*. Questions and answers are provided so you can judge how well you have understood the material. The answers are collected in an appendix.

- A programming *exercise* that applies the new concept and provides experience with the Mesa language. It is primarily through these exercises, as well as through programming examples and readings in the *Mesa Language Manual*, and the *Mesa* and *Pilot Programmers Manuals*, that you will become familiar with the XDE.

Using the course

Beginning users of Mesa come with a wide range of experience. You can use the following guidelines to gauge the level appropriate for you and how best to use this course.

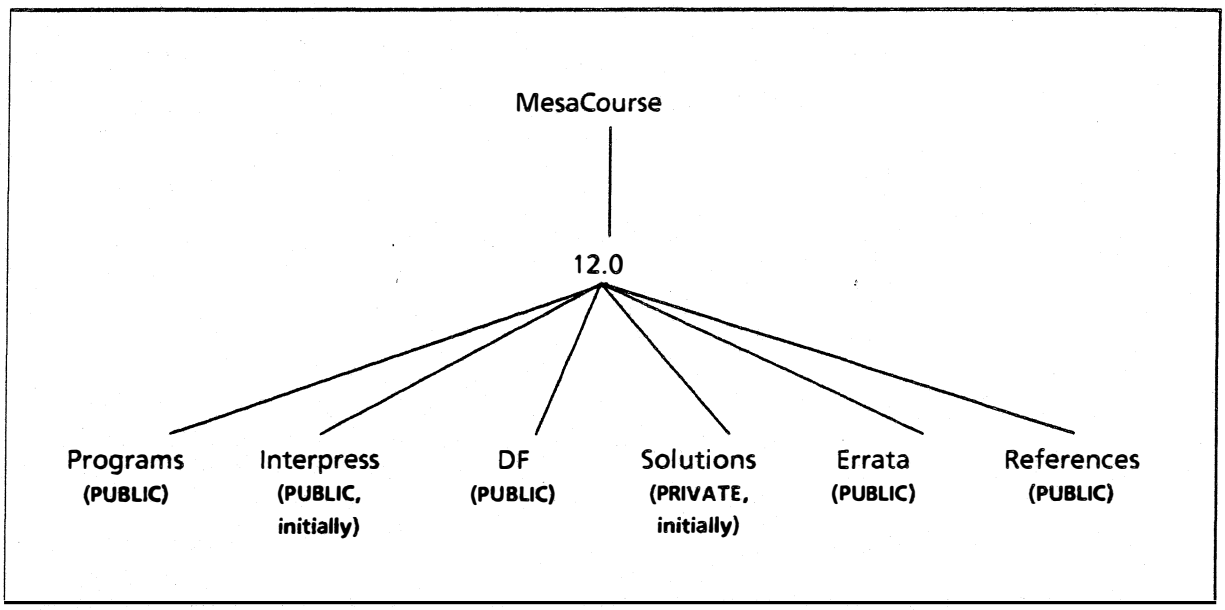
The primary purpose of this training is to initiate you to programming in the Xerox development environment. This environment is documented by well over one thousand pages of material. You need to know how to find, use, and understand information in these documents. The course presents the information in the reference materials around a framework of examples and exercises. There is no information in the course that is not also in at least one other document.

Many chapters ask you to do preliminary readings in reference manuals. If you understand the reference materials easily, then the chapter will not provide you with any more information. Instead, you may find it best, after completing the preliminary readings, to skim the chapter, check your understanding via the questions, and go straight to the exercises. On the other hand, if you find the reference readings overly difficult, do not pore over them. Instead, skim them and concentrate your efforts on the discussion section of the appropriate Mesa Course chapter. After you have finished the chapter, go back and re-read the reference material. This will give you more information on the subject, and will also give you experience in using the manuals.

Getting Started

This is version 12.0 of the Mesa Course. It assumes that you are using a Dandelion or Daybreak processor running the Sequoia release (12.0) of the Xerox Development

Environment with Tajo installed on a normal volume, CoPilot serving as a debugger for the volume on which Tajo is installed, and a User.cm that is set up for this configuration.



The Mesa Course Directory Structure

Interpress masters for the course text are stored electronically in the folder [CustomerNSFileServer]<MesaCourse>12.0>Interpress>. You can print copies of the course from these folders as you need them (universities may have this folder protected). Your local support group may have bound copies of the Mesa Course available.

The programs discussed in the chapters are stored in the [...] <...> ...> Programs>ChapterName(ChapterNumber) folder for each chapter. Retrieve all files from this folder before starting a chapter, e.g., retrieve all the files in [CustomerNSFileServer]<MesaCourse>12.0>Programs>Interfaces(2) before starting Chapter 2.

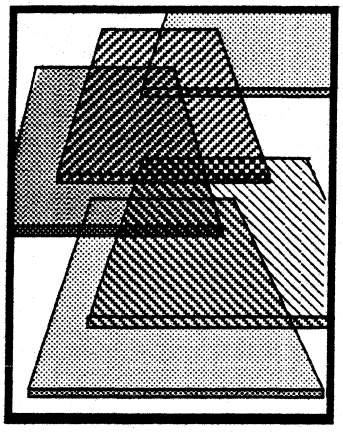
Solutions to programming exercises are stored in the [...] <...> ...> Solutions> folder. Your XDE training liaison will decide who has access rights to this folder: it may be read protected initially.

There are two papers cited in the Mesa Course that are not part of the XDE release documentation. They can be found in the [...] <...> ...> References> folder.

The Mesa Course is still under development, and we would appreciate your comments and corrections. We apologize for any inconveniences caused by inconsistencies or inaccuracies that have escaped our current review. Please check on [...] <...>...> Errata> for any update information.

If you run into any trouble getting started or while you are going through the course, do not hesitate to ask your XDE training mentor for help. Initially, please ask your mentor to make sure that your disk and User.cm are compatible with the course, and for the name of a CustomerNSFileServer near you that has a copy of the <MesaCourse> folder.

1



From Pascal to Mesa

This chapter will introduce you to the programming language Mesa by building on your knowledge of Pascal.

Pascal has become the instructional language of choice in the computer science academic community and is gaining in general popularity. It is a language that has integrated a small set of features into a powerful and efficient programming tool. One of Pascal's most attractive features is user-defined data types that enable data structuring capability and data abstraction. Standard Pascal does have a significant shortcoming in terms of writing a large system: there is no way to break the system down into small separately compiled units and then integrate them into a consistent whole. This prevents the compiler from checking the type correctness of actual parameters in distinct units, inhibits the development of "libraries" to extend the language, and generally complicates the implementation of large systems constructed by a group of programmers. Furthermore, standard Pascal does not support dynamic array bounds; it is difficult to write general routines that process arrays of different sizes. Standard Pascal has no exception handling facilities and does not support concurrent processes.

Mesa is a strongly typed, block structured programming language whose syntax is similar to that of Pascal. Mesa extends Pascal in a number of ways intended to make it more effective for the development of large systems, while preserving Pascal's data structuring and data abstraction facilities. We begin this chapter by examining the common ground between Pascal and Mesa: shared language concepts and constructs. Then we look at some of the ways in which Mesa differs from Pascal.

1.1 Definition of terms

Most of the concepts found in Pascal have counterparts in Mesa. The list below defines terms that are either distinctive to both Pascal and Mesa or terms whose Pascal and Mesa definitions differ slightly.

type definitions

Type definitions are the mechanism for describing data of Mesa programs.

name

A *name* (or identifier) is a sequence of alphabetic and numeric characters beginning with an alphabetic

character. Identifiers in Mesa can be up to 256 characters long; character case is significant in Mesa identifiers.

static variables — *Static variables* are variables for which an explicit variable declaration has been made.

dynamic variables — *Dynamic variables* are generated by a special procedure (**NEW**) that yields a pointer or reference value that subsequently serves in place of a name to refer to the variable.

strongly typed — The Mesa compiler uses static analysis to deduce the type of every constant, variable, and expression to ensure that all programs are type correct. Languages in which such type correctness is determined at compile time are called *strongly typed.*

procedural abstraction — A *procedural abstraction* is a mapping from a set of inputs to a set of outputs that can be described by a specification. The specification must show how the outputs relate to the inputs, but it does not reveal or imply the way the outputs are to be computed.

actual procedure — An *actual procedure* is a procedure initialized so that its meaning (defined by its body) cannot change. You cannot assign a value to an actual procedure.

procedure variable — A *procedure variable* is a procedure initialized in such a way that the procedure's value (body) can be changed by assignment.

1.2 A comparison of Mesa and Pascal constructs

This section presents a sequence of examples showing analogous Mesa and (standard) Pascal constructs.

Mesa	Pascal
Comments	
--This is a comment terminated by EOL	*{This is a comment}*
--This is a comment terminated by dashes--	
<<This is a comment extending over more than one line>>	*{This is a comment extending over more than one line}*

Mesa

Pascal

Constant declarations

```
Pi: REAL = 3.14;
  --Note
  -- Mesa is case sensitive.
  -- Reserved words are capitalized.
  -- Constants have explicit types.

MinusPi: REAL = -Pi;

linesPerPage: INTEGER = 60;

shortPage: INTEGER = linesPerPage - 6;

capA: CHARACTER = 'A;

smallA: CHAR = 'a;
--CHARACTER and CHAR are equivalent

message: LONG STRING = "Hello there";
--String literal allocated in global frame.

anotherMessage: LONG STRING = "Boo"L;
--The string literal is allocated in the local frame
--of the innermost procedure enclosing the
--literal. Thus, in Mesa you can choose whether
--to allocate from a local or global frame.
```

```
CONST
  Pi = 3.14;
  {Pascal is not case sensitive.
   Capitalization is only for readability.
   Constants have implicit TYPE.}

  MinusPi = -Pi;

  linesPerPage = 60;

  {Pascal does not support general
   expression constants}

  capA = 'A';

  smallA = 'a';

  message = 'Hello there';
```

Type declarations: One dimensional ARRAYS

```
Name: TYPE = ARRAY[0..9] OF CHAR;

packName: TYPE = PACKED ARRAY
  [0..9] OF CHAR;

Dashes: TYPE = ARRAY[0..7) OF CHAR ← ALL['-];
  --[0..n + 1) equivalent to [0..n]

RARRAY: TYPE = ARRAY[0..8) OF REAL;
```

```
TYPE
  Name = ARRAY[0..9] OF CHAR;

  packName = PACKED ARRAY[0..9] OF CHAR;

  Dashes = ARRAY[0..6] OF CHAR;
  {No default initialization}

  RARRAY = ARRAY[0..7] OF REAL;
```

Mesa

Pascal

Type declarations: Two dimensional ARRAYs

```
M3by4: TYPE = ARRAY[1..3] OF ARRAY[1..4]
  OF INTEGER ← ALL[0];
```

```
M3by4 = ARRAY[1..3] OF ARRAY[1..4]
    OF INTEGER;
{ No default initialization}

{or}

ALT3by4 = ARRAY[3,4] OF INTEGER;
{Compact representation of two dimensional ARRAY,
 no default initialization}
```

Type declarations: Records

```
Coordinate: TYPE = RECORD[
  horizontal: REAL ← 0.00;
  vertical: INTEGER ← 0];
  -- default field initialization

--or

Coordinate: TYPE = RECORD[
  horizontal: REAL,
  vertical: INTEGER] ← [0.00,0]
  -- default TYPE initialization
```

```
Coordinate =
    RECORD
        horizontal: REAL; {no initialization}
        vertical: INTEGER
    END;
```

Type declarations: Variant Records

```
Shape: TYPE = {point, line, circle};

Figure TYPE = RECORD[
  figureName: Name,
  specificFigure: SELECT fieldID: Shape FROM
    point = > [position: Coordinate],
    line = > [xCoef, yCoef, slope: REAL],
    circle = > [center: Coordinate,
                radius: REAL];
  ENDCASE];
```

```
Shape = (point, line, circle);

Figure =
    RECORD
        figureName: Name;
        CASE tag: Shape OF
            point :
                (postion: Coordinate);
            line:
                (xCoef, yCoef, slope: REAL);
            circle:
                (center: Coordinate;
                 radius: REAL);
    END;
```

Mesa

```
personPtr: TYPE = LONG POINTER TO Person;

Person: TYPE = RECORD[
   name: Name,
   age: [21..120].
   sex: {male, female},
   party: {Demo, GOP},
   contribution:[0..10000]];

link: TYPE = LONG POINTER TO Node;

Node:TYPE = RECORD[
   voter: Person,
   next: link];
```

Pascal

```
personPtr = ↑ Person;

Person =
    RECORD
        name: Name;
        age: 21..120;
        sex: (male, female);
        party: (Demo, GOP);
        contribution : (0..10000)
    END;

link = ↑ Node;

Node =
    RECORD
        voter: Person;
        next: link
    END;
```

Type declarations: Records containing pointers

Variable declarations

```
b: BOOLEAN ← TRUE;
--BOOLEAN and BOOL are equivalent

li, lj: LONG INTEGER ← -7;

i, j: INTEGER ← 41;
iSquared: INTEGER ← i*i;
k: INTEGER ← iSquared - i + 1;

a: RARRAY ;

mxy: M3by4;

control: [1..15];
```

```
VAR
b:BOOLEAN;  {no initialization possible}

{no double precision or initialization}

i, j: INTEGER;
iSquared: INTEGER;
k: INTEGER;
{Initialization of iSquared and k must be done
 in statement section.}

a: RARRAY;

mxy: M3by4;
altmxy: ALT3by4

control: 1..15;
```

Mesa

Pascal

Variant record variables

```
figure: Figure;
```

```
figure, pointFigure, lineFigure, circleFigure: Figure;
```

"Bound" variant record variables

```
pointFigure: point Figure;
lineFigure: line Figure;
circleFigure: circle Figure;
```

```
{Pascal has no concept of bound variant RECORDS.}
```

Dynamic storage allocation

```
z: UNCOUNTED ZONE ← NIL;
--source of dynamically allocated objects
```

```
{Nodes are automatically allocated from a
system heap}
```

Variables for pointer examples

```
cand1, cand2, cand3, cand4: Person;
preswinner, presloser, vpwinner,
vploser: personPtr;
p, rootNode: link;
```

```
cand1, cand2, cand3, cand4: Person;
preswinner, presloser, vpwinner
vploser: personPtr;
p, rootNode: link;
```

Procedure declarations

```
Fact: PROCEDURE[n: LONG INTEGER]
  RETURNS[LONG INTEGER] =
  BEGIN
    RETURN[IF n = 0 THEN 1
           ELSE n*Fact[n - 1]]
  END;
--Mesa does not differentiate between
--FUNCTION and PROCEDURE.
```

```
FUNCTION Fact(n: INTEGER): INTEGER;
  BEGIN
    IF n = 0 THEN Fact := 1
    ELSE Fact := n*Fact(n - 1)
  END;    {Fact}

{Pascal FUNCTIONS can only return "simple" TYPES,
i.e., CHAR, INTEGER, and REAL.}
```

```
Swap: PROCEDURE[iptr, jptr:
  LONG POINTER TO INTEGER] =
  {temp: INTEGER;
   temp ← iptr↑;
   iptr↑ ← jptr↑;
   jptr↑ ← temp};
```

```
PROCEDURE Swap(var i, j: INTEGER);
  VAR t: INTEGER;
BEGIN
  t := i;
  i := j;
  j := t
END;
```

```
--All arguments are passed by value in Mesa:
--i.e., the value of an argument, not its address
--is assigned to the parameter. Of course, this
--value itself can be an address.

--In Mesa, a block can be delimited either by
--BEGIN ... END or by { ... }
```

Mesa

```
a[1] ← 3.8E6;
mxy[2][3] ← 7;


IF b THEN PROCEDURE1[];

IF i # j / 2
   THEN PROCEDURE1[]
   ELSE PROCEDURE2[];

a[1] ← IF boolvar1
   THEN 4.56
   ELSE 8.71;
--An IF expression

--control: [1..15];
SELECT control FROM
   1, IN [7..10] = > statement1;
   2, 5, >10     = > statement2;
   ENDCASE       = > statement3;


SELECT TRUE FROM
   boolvar1 = > statement1;
   boolvar2 = > statement2;
   . . .
   boolvarn = > statementn;
ENDCASE;


a[1] ← SELECT control FROM
   1, IN [7..10] = > 1.12;
   2, 5, >10     = > -4.856;
   ENDCASE       = > 73.2;
--A SELECT expression

i: INTEGER ← 1;
WHILE i < 10
   DO ... i ← i + 1; ...ENDLOOP;
```

Pascal

Statements

```
   a[1] : = 3.8E6;
   mxy[2][3] : = 7;
   altmxy[2,3] : = 7;

IF b THEN PROCEDURE1;

IF i < > j div 2
   THEN PROCEDURE1
   ELSE PROCEDURE2;

IF boolvar1
   THEN a[1] : = 4.56
   ELSE a[1] : = 8.71;


{control: 1..15;}
CASE control OF
   1, 7, 8, 9, 10: statement1;
   2, 5, 11, 12, 13, 14, 15: statement2;
   3, 4, 6: statement3
END;

IF boolvar1 THEN
   statement1
ELSE IF boolvar2 THEN
   statement2
. . .
ELSE IF boolvarn THEN
   statementn;

CASE control OF
   1, 7, 8, 9, 10: a[1] : = 1.12;
   2, 5, 11, 12, 13, 14, 15: a[1] : = -4.856;
   3, 4, 6: a[1] : = 73.2
END;

i : = 1;          {assume i defined earlier}
WHILE i < 10 DO
   BEGIN ... i : = i + 1; ...END;
```

Mesa

Pascal

Statements continued

```
i: INTEGER ← 1;
DO
  ...i ← i + 1; ...
  IF i > = 10 THEN EXIT;
ENDLOOP;
```

```
i : = 1;
REPEAT ... i: = i + 1; ...
UNTIL i ≥ 10;
```

```
--The Mesa construct
--
--UNTIL condition DO
--   {StatementSeries};
--ENDLOOP;
--
--is similar to that of Pascal except that the
--condition is tested at the "top "of the LOOP
--and, if false, the LOOP is not executed. REPEAT
--is a Mesa reserved word whose semantics are
--not the same as Pascal REPEAT.
```

```
{In the Pascal construct

  REPEAT StatementSeries
  UNTIL condition;

the condition is tested only after the StatementSeries
has been executed once, i.e., the test is at the "bottom"
of the LOOP.}
```

```
FOR i: INTEGER IN [1..n) DO
 ... sum ← sum + a[i]; ...
ENDLOOP;
```

```
{i: INTEGER; defined earlier}
FOR i : = 1 to n - 1 DO
BEGIN ...sum ← sum + a[i]; ...END;
```

Unbound variant record initialization

```
figure.figureName ← ['a, 'r, 'b, 'i, 't, 'r, 'a, 'r, 'y];
WITH f: figure SELECT FROM
  point = > f.position ← [-1.37, 14];
  line = > {f.xCoef ← 2.81,
            f.yCoef ← 4.2,
            f.slope ← -.7};
  circle = > {f.center ← [0.00,3.00],
              f.radius ← 5.00};
ENDCASE;

--the variable figure must be renamed
--within the WITH statement
```

```
figure.figureName[0] : = 'a';
figure.figureName[1] : = 'r'; ...
WITH figure DO
  CASE tag OF
    point: WITH position DO
      BEGIN horizontal : = -1.37;
            vertical : = 14;
      END;
    line: BEGIN
          xCoef : = 2.81;
          yCoef : = 4.2;
          slope : = -.7;
        END;
    circle: WITH center DO
      BEGIN horizontal : = 0.00;
            vertical : = 3.00;
            radius : = 5.00;
      END
  END;
```

Bound variant record initialization

```
pointFigure.figureName ← ['p, 'o, 'i, 'n, 't, ' ,'1, ' ,' ];
pointFigure.point← [-1.37, 14];
```

```
{Pascal has no notion of bound variants}
```

Mesa

Pascal

Some pointer examples

```
cand1 ← Person[
  name : Name['R, 'e, 'a, 'g, 'a, 'n, ' , ' , ' ],
  age : 72,
  sex : male,
  party : GOP,
  contribution : 0];
```

```
WITH cand1 DO
  BEGIN
    name[0] : = 'R'; name[1] : = 'e'; . . .
    age : = 72;
    sex : = male;
    party : = GOP;
    contribution : = 0;
  END;
```

```
--Similarly initialize cand2 to MondaleData,
--cand3 to BushData, and cand4 to FerraroData.
```

```
z← Heap.Create[initial:1];
--Initialize source FOR dynamically
--allocated objects
```

```
{Pascal allocation will be from an anonymous
system heap.}
```

```
preswinner ← z.NEW[Person ← cand1];
presloser ← z.NEW[Person ← cand2];
vpwinner ← z.NEW[Person ← cand3];
vploser ← z.NEW[Person ← cand4];
```

```
NEW(preswinner); preswinner ↑ : = cand1;
NEW(presloser);  presloser ↑ : = cand2;
NEW(vpwinner);   vpwinner ↑ : = cand3;
NEW(vploser); vploser ↑ : = cand4;
```

```
preswinner ← presloser;
--preswinner and presloser both point to
--the same RECORD (initialized to MondaleData).
--No access path remains to the RECORD initialized
--with ReaganData.
```

```
preswinner : = presloser;
```

```
vpwinner ↑ ← vploser ↑ ;
--vp winner and vploser point to distinct
--RECORDS, each initialized to FerraroData.
```

```
vpwinner ↑ : = vploser ↑ ;
```

```
FOR p: LONG POINTER TO Node ←
rootNode, p.next UNTIL p.next = NIL DO
  IF p.voter.contribution > 100
    THEN AskForMoney[p.voter.name]
ENDLOOP;
```

```
p : = rootNode;
WHILE p < > NIL DO
  BEGIN
    IF p ↑ .voter.contribution > 100
      THEN AskForMoney[p ↑ .voter.name];
    p : = p.next
  END;
```

```
--When applied to a pointer, the operation
--of selection implies dereferencing. In Mesa,
--this type of dereferencing is done
--automatically. Thus, it is not necessary to
--write p ↑ .voter.contribution or
--p ↑ .voter.name.
```

1.3 Mesa extensions of Pascal

1.3.1 Modules and interfaces

Mesa programs look quite similar to Pascal programs when viewed in the small. However, Mesa provides and enforces a modularization capability that is far more powerful than that of Pascal. In Mesa, you build large systems from a collection of smaller, separately compiled components called modules. The Mesa *binder* (the binder is similar to a linking loader in Pascal) enforces *strong type checking* among the modules that make up a system. In Pascal, you must make a choice when developing a large system. Either you construct a monolithic program to ensure type correctness, or you link separately complied program units without any guarantee that the type of variable X in one unit matches the type of variable X in another unit. In the latter case, type mismatches are discovered only at run-time.

Type checking across module boundaries in Mesa is only part of its modularization power. There are two categories of module in Mesa. *Definitions* (or interface) modules declare types, constants, and procedure headers of procedures that manipulate values of types declared in the module. An interface defines an abstraction by collecting all operations on a class of objects into a single module. An interface module contains no executable code; it only contains enough information to allow the compiler to type check other modules that use the declared symbols. The body of a procedure declared in an interface is not part of the interface. Interface modules compile into symbol tables.

The second category of module is the *Program* module. A program module acts as an *implementor* of an interface if it contains code that implements procedures declared in an interface module. A program module acts as a *client* of an interface if it calls procedures defined in that interface module.

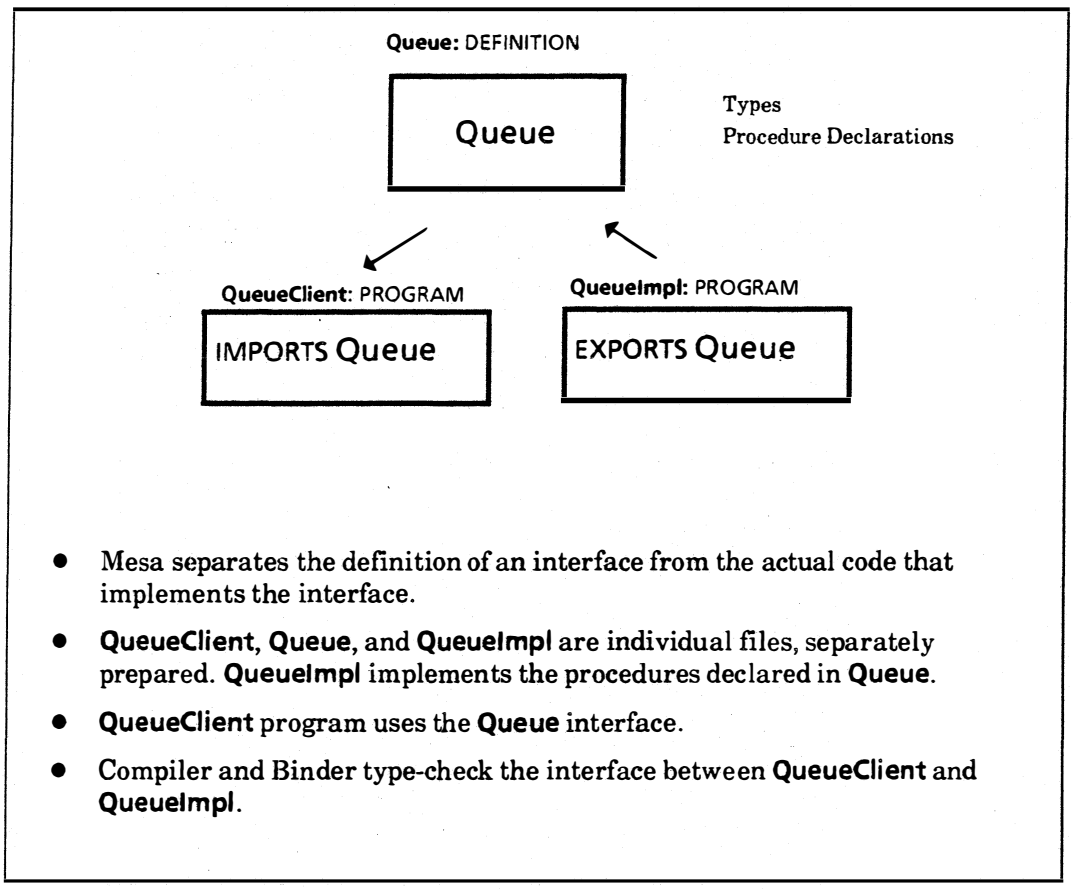
An interface is a contract between client and implementor: the interface specifies items that are available for clients to use, but doesn't say how they will be provided; the implementing module determines the details of the implementation.

There are several advantages of interfaces:

- Once an interface has been agreed upon, construction of the implementor and client can proceed independently. Thus interfaces and implementations are decoupled. This facilitates information hiding and permits changes to implementing modules without requiring a change to a client. Once an abstraction has been defined in a **DEFINITIONS** module (the interface) and implemented in one or more **PROGRAM** modules, an arbitrary (client) **PROGRAM** module can access the services advertised in the interface.

- Interfaces enforce consistency in the connections among modules. Operations upon a class of objects are collected into a single interface, not defined individually and in potentially incompatible ways.

- Nearly all of the work required for type-checking interfaces is done by the compiler.



Mesa modularity

1.3.2 Exceptions: signals and errors

Mesa provides *signals* to indicate exception conditions. Signals provide an orderly means for dealing with exceptions that is inexpensive if they occur infrequently. Examples of exceptions are invalid inputs, the inability of an abstractions to respond (e.g.,an allocator out of space), or any unusual or "impossible" event.

A Mesa **SIGNAL** can be thought of as the association of a procedure with an exceptional condition. "Raising" a signal when the exception occurs is similar to invoking the associated procedure except that the code to be executed is determined dynamically and is found in a "handler". The binding to a handler is determined by searching *catch phrases* (that contain handlers) in the call stack of the process in which the exception is raised; the dynamically innermost catch phrase that accepts the signal (by having a handler prepared to deal with the signal) is selected and executed. Often, parameters are passed when the signal is raised to help a handler determine what went wrong. Catch phrases are written in a distinctive syntax that clearly identifies them as the location of handlers containing code to respond to signals.

The cost of raising a signal is significantly higher than the cost of calling a procedure, but exceptions are events that should not happen very often. The system guarantees that all exceptions are handled at some level; those that the program fails to catch are accepted by the debugger. The debugger keeps intact the state of the program that raises a signal.

1.3.3 Processes, monitors, and condition variables

Mesa provides efficient mechanisms for concurrent execution of multiple processes within a single system. This allows programs that are inherently parallel in nature to be clearly expressed.

Example

```
GetInput: PROCEDURE[buffer: LONG POINTER TO Buffer]
    RETURNS [bytesRead: CARDINAL] =
BEGIN
p: PROCESS RETURNS [CARDINAL];
...

p ← FORK ReadLine[buffer];
...
<< concurrent computation >>
...
bytesRead ← JOIN p;
END;
```

FORK makes it possible to start the execution of another procedure concurrently with the program that started it. **FORK** returns a process, which may either be detached to proceed independently, or saved for a future **JOIN**. A process type is declared similarly to a procedure type, except that only the type of the result is specified.

All processes execute in the same address space. Consequently, they are not protected from each other (certainly acceptable in a single-user system) but process creation and switching between processes is cheap (about the same as a procedure call).

Mesa provides facilities for synchronizing processes by means of entry to monitors and waiting on condition variables. A monitor has shared data in its global frame, and its own procedures for accessing it. To prevent two processes from executing the the same monitor at the same time, a *monitor lock* is used for mutual exclusion. Calling one of a monitor's **ENTRY** procedures automatically acquires the monitor lock (**WAIT**ing if necessary), and a return releases it. The monitor lock serves to guarantee the integrity of the global data, which is expressed as the monitor invariant, an assertion defining what constitutes a "good state" of the data for that particular monitor. It is the responsibility of every entry procedure to restore the monitor invariant before returning.

Example

```
StorageAllocator: MONITOR =
  BEGIN
  StorageAvailable: CONDITION;
  Block: TYPE = RECORD[...];
  ListPtr: TYPE = LONG POINTER TO ListElmt;
  ListElmt: TYPE = RECORD[block: Block, next: ListPtr];
  FreeList: ListPtr;

  Allocate: ENTRY PROCEDURE RETURNS [p: ListPtr] =
    BEGIN
    WHILE FreeList = NIL DO
      WAIT StorageAvailable
      ENDLOOP;
    p ← FreeList; FreeList ← p.next;
    END;

  Free: ENTRY PROCEDURE[p: ListPtr] =
    BEGIN
    p.next ← FreeList; FreeList ← p;
    NOTIFY StorageAvailable
    END;

  END.
```

It may happen that one process enters the monitor, finds the monitor data in a valid state, but cannot continue until some other process enters the monitor and alters the state (for example, a process may find that there is no storage available). The **WAIT** operation allows the first process to release the monitor lock and await the desired condition. The **WAIT** is performed on a condition variable associated by agreement with the actual condition required. When another process makes that condition true, it will perform a **NOTIFY** on the condition variable, and the waiting process will continue from where it left off (after reacquiring the lock) and testing the condition again.

1.3.4 New data types

In Mesa, the predefined type **LONG STRING** is really "**LONG POINTER TO** *Stringbody*"; a *StringBody* contains a packed array of characters, a maxlength field giving the length of that array, and a length field indicating how many of the characters are currently significant. Each program contains the following predeclarations:

Example

```
LONG STRING: TYPE = LONG POINTER TO StringBody;
StringBody: TYPE = MACHINE DEPENDENT RECORD[
  length: CARDINAL,
  maxlength: --readonly-- CARDINAL,
  text: PACKED ARRAY[0..0) OF CHARACTER];

whatWasThat: LONG STRING = "Eh?";   --constant STRING
answer: LONG STRING ← [256];        --allocate a StringBody with maxlength 256
```

A sequence is an indexable collection of items, all of which have the same type. In this respect a sequence resembles an array; however, the length of the sequence is not part of its type. The (maximum) length of a sequence is specified when the object containing that sequence is created, and it subsequently cannot be changed. It is the responsibility of the programmer to keep track of the number of items in the sequence at any time. Sequences are declared as the last field in a record.

Example

```
lptscr: TYPE = LONG POINTER TO SequenceContainingRecord;
finger: lptscr ← NIL;
SequenceContainingRecord TYPE = RECORD[
  a: BOOLEAN,
  b: BOOLEAN,
  seq: SEQUENCE length:CARDINAL OF LONG INTEGER];

...
finger ← Heap.systemZone.NEW[SequenceContainingRecord[10]];
--SequenceContainingRECORD[10] is a TYPE specification describing a RECORD with a
--sequence part, seq, containing 10 LONG INTEGERS. The effect of the call is to allocate
--enough storage to hold two BOOLEANS and 10 LONG INTEGERS and return a long
--pointer to this storage.
```

Dynamic variables in Mesa are allocated in *zones*. Zones are not necessarily associated with fixed areas of storage; rather they are objects characterized by procedures for allocation and deallocation. There is a standard system zone, **systemZone**, but programs that allocate substantial numbers of similar dynamic variables can often improve performance by segregating each kind into its own zone. **NEW** is used to allocate a dynamic variable from a zone, and **FREE** to release it.

Mesa allows a default initial value to be associated with a type. Default values for arguments can simplify procedure applications; default initial values are useful to ensure that the corresponding storage is always well-formed, even before the variable has been used by the program.

1.3.5 Mesa extensions of Pascal constructs

This section mentions a number of areas where Mesa provides "convenience" extensions or conceptually small changes.

SELECT statements generalize Pascal's CASE construct by allowing several ways to specify how one statement is to be chosen for execution from an ordered list. The most common form is based on the relation between the value of a given expression and those of expressions associated with each selectable statement. The relation may be equality (the default), any relational operator appropriate to the types of the values involved, or containment in a subrange. A single selection may be prefixed by several selectors and an optional **ENDCASE** statement is selected only if none of the others are. *Discriminating* selection is used to branch on the type of a variant record value. **SELECT** expressions are analogous, but choose from an ordered list of expressions.

Examples

```
--control: [1..15];
SELECT control FROM
  1, IN [7..10] = > statement1;
  2, 5, >10     = > statement2;
  ENDCASE       = > statement3;
```

```
a[1] ← SELECT control FROM
    1, IN [7..10] = > 1.12;
    2, 5, >10     = > -4.856;
ENDCASE   = > 73.2;
    --A SELECT expression
```

```
Shape: TYPE = {point, line, circle};

Figure TYPE = RECORD[
  figureName: Name,
  specificFigure: SELECT fieldID: Shape FROM
    point = > [position: Coordinate]
    line = > [xCoef, yCoef, slope: REAL],
    circle = > [center: Coordinate,
                radius: REAL];
  ENDCASE];
```

Iteration is provided by loop statements in which several different kinds of control can be freely intermixed. A loop has a *control clause* and a *body*. The control clause may specify a logical condition for normal termination, possibly combined with a range or a sequence of assignments for a *controlled variable*. In addition to ordinary statements, the body may contain **EXIT** or **GOTO** statements to explicitly terminate its execution, and may be followed by a **REPEAT** clause that acts like a selection on the **GOTO** used to terminate the loop. (**GOTO** cannot be used to synthesize arbitrary control structures. It is much like a "local" exception.)

Examples

```
i ← 1;
UNTIL i > = 10                    --UNTIL i> = 10 is the loop control
  DO ... i ← i + 1 ; ...ENDLOOP;
Next-Statement;
```

The following example is equivalent to the one above.

```
i ← 1;
DO
IF i > = 10 THEN GOTO quit;       --first statement in the body
... i ← i + 1 ; ...
REPEAT --REPEAT doesn't mean repeat, it means "location of exits options".
  quit = > NULL;
ENDLOOP;
Next-Statement;
```

An example of linked list traversal:

```
NodeLink: TYPE = LONG POINTER TO Node;
node, headOfList: NodeLink;
Node: TYPE = RECORD[
   listValue: SomeTYPE,
   next: NodeLink];

FOR node ← headOfList, node.next UNTIL node = NIL
   DO ... ENDLOOP;
```

The *loop control* variable is **node**. Its *initial value*, **headOfList**, is assigned prior to the first iteration. Before each subsequent iteration the *next expression*, **node.next**, is reevaluated and assigned to the control variable. The user must either use a **GOTO** to terminate the loop or include a *condition test*. The condition test **UNTIL node = NIL** was used in the above example.

The **LOOP** statement is used when there is nothing more to do in the iteration, and the programmer wishes to go on to the next repetition, if any.

```
stuff: ARRAY[0..100) of PotentiallyInterestingData;
Interesting: PROCEDURE[PotentiallyInterestingData] RETURNS[BOOLEAN];
i: CARDINAL;

FOR i IN [0..100) DO
   ---some PROCessing FOR each value of i
   ...
   IF ~Interesting[stuff[i]] THEN LOOP;
   --PROCess stuff[i];
   ...
ENDLOOP;
```

In Pascal, procedure execution must proceed somehow to the end of the body before terminating; in Mesa, it can be terminated anywhere by executing a **RETURN** statement. If the procedure's type includes results, the **RETURN** statement may supply the values to be returned - otherwise they are taken from the result variables named in the type. Each procedure body is followed by an implicit return.

Examples

```
ReturnExample1: PROCEDURE[option: [1..4]] RETURNS[a, b, c: INTEGER] =
  BEGIN
  a ← b ← c ← 0;
  SELECT option FROM
    1 => RETURN [a:1, b:2, c:3];          --keyword parameter list
    2 => RETURN [1, 2, 3];                -- position version of option 1
    3 => RETURN;                          -- a = b = c = 0
    ENDCASE => b ← 4;
  c ← 9;
  END; -- implicit return; a = 0, b = 4, c = 9

ReturnExample2: PROCEDURE[g: INTEGER] RETURNS[INTEGER ← 3, INTEGER ← 4] =
  BEGIN
  SELECT g FROM
    0 => RETURN [ , 2];                   -- RETURNS [3,2]
    1 => RETURN [8, ];                    --RETURNS [8,4]
    2 => RETURN [ , ];                    --RETURNS [3,4]
    3 => RETURN [5];                      --RETURNS [5,4]
    4 => RETURN [ ];                      --RETURNS [3,4]
    ENDCASE =>
  END;                                    --implicit return: [3,4]
```

Pascal procedures are not values that may be assigned to variables; Mesa procedures are.

Example

```
InverseTrigValue: REAL;
InverseTrigFunction: TYPE = PROCEDURE [x: REAL] RETURNS [REAL];

ArcSin: InverseTrigFunction = BEGIN --PROCEDURE body-- ...END; --PROCEDURE constant
ArcCos: InverseTrigFunction = BEGIN --PROCEDURE body-- ...END; --PROCEDURE constant
ArcTan: InverseTrigFunction = BEGIN --PROCEDURE body-- ...END; --PROCEDURE constant
InverseTrigFunctionVariable: InverseTrigFunction;      --PROCEDURE variable
...
InverseTrigFunctionVariable ← ArcSin;
InverseTrigValue ← InverseTrigFunctionVariable[3.1415/4];
```

1.3.6 Input and output in Mesa

The Mesa language definition omits many of the features commonly expected in programming languages, such as input/output and string manipulation operations. These facilities are available to Mesa programmers, but they are provided by interfaces written in the language itself. Standard interfaces are documented in the *Mesa Programmer's Manual.*

1.4 References

The definitive reference for the language is the *Mesa Language Manual,* version 11.0. The remaining chapters in the Mesa Course will guide your reading of the Mesa Language Manual and will discuss in detail all of the topics mentioned only briefly in this chapter.

1.5 Exercises

1. Convert the following Pascal program fragment to Mesa.

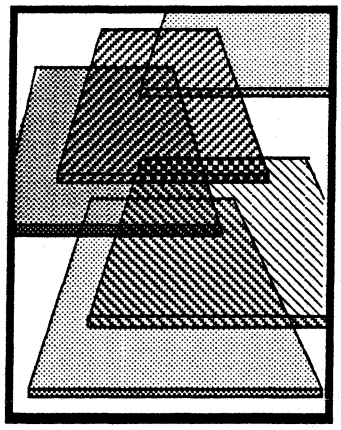
```
CONST
  maxlength = 1000;
TYPE
  index = 1..maxlength;
  rowType = ARRAY [index] OF integer;
VAR
  inrow : rowType;
  ix: index;

PROCEDURE shellsort (VAR row : rowType; length : index);
  VAR
      jump, m, n : index;
      temp : integer;
      alldone : boolean;

  BEGIN
      jump : = length;
      WHILE jump > 1 DO
          BEGIN
              jump : = jump DIV 2;
              REPEAT
                  alldone : = true;
                  FOR m : = 1 TO length - jump DO
                      BEGIN
                          n : = m + jump;
                          IF row[m] > row[n]
                              THEN
                                  BEGIN
                                      temp : = row[m];
                                      row[m] : = row[n];
                                      row[n] : = temp;
                                      alldone : = false
                                  END
                      END { FOR}
              UNTIL alldone
      END {while}
  END; {sort}
```

2. Convert the following Pascal program fragment to Mesa.

```
{straight list insertion}
TYPE
  ref = ↑word;
  word = RECORD
           key : integer;
           count : integer;
           next : ref
         END;
VAR
  root: ref;

PROCEDURE search (x: integer; VAR root: ref);
  VAR
    w : ref;
    b : boolean;
  BEGIN
    w : = root;
    b : = true;
    WHILE ( w < > nil) AND b DO
      IF w ↑ .key = x THEN b : = false ELSE w : = w ↑ .next;
    IF b THEN
      BEGIN {NEW ENTRY}
        w : = root;
        NEW(root);
        WITH root ↑ DO
          BEGIN
            key : = x;
            count : = 1;
            next : = w
          END
      END
    ELSE
      w ↑ .count : = w ↑ .count + 1
  END; {search}
```

Notes:

2



Interfaces

As mentioned in the last chapter, the chief differences between Pascal and Mesa lie not in the syntax of the language, but rather in how modules interact to share information, and how individual modules are combined together into systems. Mesa's structured modularization allows modules to be created and tested individually, and then later integrated with complete type safety. Thus, Mesa effectively reduces the problems of programming in the large down to the problems of programming in the small. This chapter illustrates how Mesa's interfaces allow individual programs to share information; the next chapter discusses how interfaces are used in large-scale system building.

2.1 Preliminary readings

Skim the first five chapters in the *Mesa Language Manual* to get acquainted with the common Mesa constructs and syntax. You will need these chapters as a reference as you read this chapter and do the exercises.

Read Appendix B of the *Mesa Language Manual*, Programming Conventions, before you start to write your own programs.

2.2 Definition of terms

Client

A *client* is a program (as opposed to a person) that uses the services of another program or system.

Interface

An *interface* is a formal contract between pieces of a system that describes the services to be provided. A provider of these services is said to *implement* the interface; a consumer of them is called a *client* of the interface.

Interface module

An *interface* or **DEFINITIONS** *module* defines types, variables, constants, procedures, and signals, thus specifying the services to be provided by its implementation modules.

Implementation module

An *implementation* or **PROGRAM** *module* is a program that codes (*implements*) and makes available to clients (*exports*) items in an interface. One implementation module can export all or part of one or several interfaces, and an

	interface can be implemented by several implementation modules jointly.
Load	*Loading* a module allocates memory space for its code and data, and links it to other modules that are already loaded, but does not start it.
Symbol	A *symbol* is any user-defined name in a program, such as a constant, type, variable, or procedure.

2.3 Discussion

There are two kinds of modules in Mesa: **DEFINITIONS** and **PROGRAM**. **DEFINITIONS** modules are also called *interface modules*, or just *interfaces* for short. You can think of an interface or **DEFINITIONS** module as a catalog containing a precise description of each item offered. The purpose of an interface is only to *define* procedures and variables that will be available to other programs; the interface does not contain the actual code for those procedures.

All executable code is contained in the second kind of module, called a **PROGRAM** module. A program module can act as a manufacturer of an interface (creating the items in the catalog), or as a customer (ordering items from the catalog). In Mesa, the "manufacturers" are called *implementors*, and the "customers" are called *clients*. Thus, program modules communicate via interfaces: a shared symbol is defined in an interface module, implemented by a program module, and used by other program modules. The interface is the link between the two program modules; there is no direct communication between client and implementation.

One advantage of this approach is information hiding; the client knows nothing of the implementation, and thus cannot take advantage of specific details of that implementation. Another important advantage is that the implementation is decoupled from the client; as long as the declaration in the interface remains the same, the implementation can be changed without affecting the client.

The rest of this chapter discusses the mechanics of linking together the three basic pieces of the interface mechanism, which are:

(1) an interface or **DEFINITIONS** module,
(2) an implementor of that interface, which is a **PROGRAM** module, and
(3) a client, which is also a **PROGRAM** module.

2.3.1 CompareImplA, which uses no interfaces

You can write Mesa code without using interfaces at all. CompareImplA.mesa is a simple example of a self-contained **PROGRAM** module. Take a look at the code:

```
CompareImplA: PROGRAM =
  BEGIN
  Compare: PROCEDURE [x,y: CARDINAL] RETURNS [same: BOOLEAN] =
    BEGIN
    IF x = y THEN RETURN[same←TRUE]
    ELSE RETURN[same←FALSE] ;
    END; --of procedure Compare
  END.
```

CompareImplA consists of one procedure, **Compare**, which takes two numbers as arguments, compares them, and returns a result of either **TRUE** (the numbers are the same) or **FALSE** (the numbers are not the same). However, there is no mainline code to call **Compare**, nor are there any I/O calls to get input or print results. Obviously, this program is of little use by itself. One way to make it useful is to "publish" it so that other programs can call our **Compare** procedure. This is called *exporting* the procedure.

2.3.2 Exporting

Exporting describes the relationship between an interface and its implementation. If you want to make a procedure available to the outside world, you define that procedure in an interface, implement it in a program module, and *export* the implementation to the interface. Client programs can then access the procedure directly from the interface. This process is called *exporting an interface.*

To use the earlier analogy, we want to publish a catalog from which clients can order a compare procedure, and we want to sign up as the manufacturer of the compare procedure advertised in the catalog. To do this, we have to write the interface and upgrade **CompareImplA** so that it exports **Compare**.

2.3.2.1 The interface

Here is the interface, which we have called **InterfaceB**:

```
InterfaceB: DEFINITIONS =      --keyword DEFINITIONS declares this to be an interface
  BEGIN
  Compare: PROCEDURE [x,y:CARDINAL] RETURNS[result:BOOLEAN];
  END.
```

This module is an interface; it defines procedures that are available to others. This particular interface contains only one definition, that of the procedure **Compare**. **InterfaceB** provides enough information about **Compare** so that the compiler can type-check client programs, but it does not contain the actual executable code for **Compare**. The actual code for **Compare** is in our implementation, which is a **PROGRAM** module.

2.3.2.2 The implementation

Here is **CompareImplB**, the implementation module:

```
DIRECTORY
  InterfaceB;
CompareImplB: PROGRAM EXPORTS InterfaceB =
  BEGIN
  Compare: PUBLIC PROCEDURE [x,y:CARDINAL] RETURNS[result:BOOLEAN] =
    BEGIN
    IF x = y THEN RETURN[result ← TRUE]
    ELSE RETURN[result ← FALSE] ;
    END; --of procedure Compare
  END.
```

This module is an upgraded version of **CompareImplA**; the code for the procedure is the same, but this time we are exporting the code to the interface. To export all or part of an interface, you need to do three things You need to specify that you are referencing other

modules, you need to list the interfaces that you are exporting, and you need to list the specific procedures that you are exporting.

The **DIRECTORY** clause in **CompareImplB** accomplishes the first of these three; it tells the compiler which interfaces will be referenced during this compilation. If you want to use information from an interface, you must include that interface in your **DIRECTORY** clause. In this case, the compiler needs to reference **InterfaceB** to verify that the procedure declaration in the implementation matches the procedure declaration in the interface.

The **EXPORTS** clause accomplishes the second objective; it lists the interfaces that are being implemented, at least in part, by this module. An exporting module need not implement all the symbols in an interface; the implementation of an interface is often the cooperative effort of several modules. A **PROGRAM** module can also export more than one interface.

The third objective is achieved by declaring **Compare** to be a **PUBLIC** procedure. Symbols can be declared as being **PUBLIC** or **PRIVATE**. **PUBLIC** symbols can be exported to an interface, but **PRIVATE** symbols cannot. In **PROGRAM** modules, the default is **PRIVATE**: all symbols are assumed to be **PRIVATE** unless specifically declared **PUBLIC**. Thus, the word **PUBLIC** indicates that **Compare** is an implementation that is being exported to an interface. The compiler verifies that the declaration matches the declaration in the interface exactly, except for the word **PUBLIC**.

Figure 2.1 summarizes the communication between an interface and its implementation.

Interface

```
InterfaceName: DEFINITIONS =
  BEGIN
  ProcedureName: PROCEDURE...;
  END.
```

Implementor

```
DIRECTORY
  InterfaceName;
InterfaceImpl: PROGRAM
 EXPORTS InterfaceName =
  BEGIN
  ProcedureName: PUBLIC PROCEDURE ... =
    BEGIN
    .....
    END; -- of procedure
  END. -- of implementation module
```

Figure 2.1

2.3.3 Importing

Now that we have exported **Compare**, other programs can use it. Conveniently, we have a willing client, **CompareClient**, eagerly waiting on the sidelines to *import* our code.

Importing describes the relationship between a client program and an interface. A client that wishes to use a particular procedure only needs to know the definition of the procedure and the name of the interface from which to access it. It knows nothing about

the actual implementation. Thus, in our example, **CompareImplB** exported **Compare** to the interface **InterfaceB**, and now **CompareClient** can import **Compare** from **InterfaceB**. There is no direct communication between **CompareImplB** and **CompareClient**.

2.3.3.1 Importing a procedure

Here is the skeleton of **CompareClient**:

```
DIRECTORY
  InterfaceB USING [Compare] ;
CompareClient: PROGRAM IMPORTS InterfaceB =
  BEGIN
  .....
  f ← InterfaceB.Compare[a, b] ;
  .....
  END ;
```

There are three steps to importing a procedure, which correspond to the three steps of exporting a procedure. First, you must list the interface in the **DIRECTORY** statement, just as in the exporting example. This tells the compiler that your module references **InterfaceB**. In this example, the **DIRECTORY** clause is further restricted by a **USING** clause, which lists the specific symbols that you will be using from that interface. Thus, **CompareClient** can use **Compare** from **InterfaceB**, but cannot use any other symbols from that interface. You do not have to have a **USING** clause, but it is a very good idea.

Second, you need to list **InterfaceB** in the **IMPORTS** list; *this specifies the interfaces for which implementations must be provided at run-time.*

Finally, you need to indicate that the procedure is imported by referring to it as **InterfaceB.Compare**, and not just **Compare**. You must always fully qualify the name of an imported symbol so that the compiler will know that it is coming from another interface.

2.3.3.2 Template for importing a procedure

Figure 2.2 diagrams the communication between an interface and a client that **IMPORTS** a procedure.

Interface

```
InterfaceName: DEFINITIONS =
  BEGIN
  ProcedureName: PROCEDURE...;
  END.
```

Client

```
DIRECTORY
  InterfaceName USING [ProcedureName];
ClientName: PROGRAM
  IMPORTS InterfaceName =
BEGIN
  ... InterfaceName.ProcedureName[...] ;...
  END.
```

Figure 2.2

2.3.3.3 Importing a constant

In the last section, we discussed how to import a procedure from an interface.However, not all information in an interface requires an implementation. Some of the symbols in an interface, such as variables, types, and constants, are *compile-time* symbols. Such symbols are available directly from the interface; no implementation is necessary. *Run-time* symbols, on the other hand, are symbols (such as procedures) for which code must be supplied at run-time. If you use only compile-time symbols from an interface, and not run-time symbols, you do not need to import the interface. For example, here is an interface:

```
IncrementDefs: DEFINITIONS =
  BEGIN
    inputTooBig: CARDINAL = LAST[CARDINAL]     --LAST returns largest value
  END.
```

and here is the module **IncrementImpl**, which imports **inputTooBig** from **IncrementDefs**.

```
DIRECTORY
  IncrementDefs USING [inputTooBig];        -- note interface and constant name
  IncrementImpl: PROGRAM =
    BEGIN
    Increment: PROCEDURE [x: CARDINAL] RETURNS [y:CARDINAL, error:BOOLEAN] =
      BEGIN
      IF x < IncrementDefs.inputTooBig THEN          -- note fully-qualified name
        RETURN [y ← x + 1, error ← FALSE]
      ELSE RETURN[y ← x, error ← TRUE];
      END;
    END.
```

Thus, importing compile-time information is just like importing run-time information, except that you do not need to include the interface in the **IMPORTS** list. The **IMPORTS** list includes only those interfaces for which run-time implementations are needed.

2.3.3.4 Template for importing a constant

Figure 2.3 diagrams the communication between an interface and a client that is importing a constant from that interface.

Interface

```
InterfaceName: DEFINITIONS =
  BEGIN
  ConstantName: CARDINAL = ...;
  END.
```

Client using a constant

```
DIRECTORY
  InterfaceName USING [ConstantName];
InterfaceImpl: PROGRAM =
  BEGIN
  ...InterfaceName.ConstantName...;
  END.
```

Figure 2.3

2.3.4 Compiling and running your programs

As discussed above, a module's **DIRECTORY** clause lists all the interfaces referenced by that module. When you compile a module, the compiler needs to be able to read all the interfaces listed in the **DIRECTORY** clause so that it can type-check your program. This means that if you list an interface in your **DIRECTORY** clause, you must have the compiled version of that interface on your local disk when you compile your program, or you will get a compilation error. Thus, an interface must always be compiled *before* program modules that reference that interface.

Another important thing to remember is that when you recompile an interface, you will have to recompile all of its clients and implementors as well. The reason for this is that all Mesa object modules (.bcd files) contain a time stamp as part of their identification. When clients and implementors of an interface are compiled, the time stamp of the interface is noted and retained in both the client and implementation object code file identification. When you try to combine the client and the implementation into a larger system, the time stamps are checked against one another. If the client and the implementation do not reference the same version of the interface, a version mismatch will occur, which prevents the system from running.

Once you have compiled all the modules that make up a system, you can run the system. In the next chapter, you will learn how to use the *binder* to help you group your modules together, but for now you will have to load them all manually from CommandCentral. (All modules listed on the **Run** line of CommandCentral will be loaded.) You need to load all the program modules (your client, plus the implementations for any procedures that you have imported), but not the interfaces (since they don't contain executable code.) Implementation modules must be loaded before client modules, so that the implementation is ready when the client needs it.

Thus, to execute the Compare system, you would have to set up Command Central like this, and invoke **Go!**. You can run Compare now, if you like. (Note: **CompareClient** references some interfaces that you may not have on your local disk, so we have provided a compiled version of this module. Normally you would have to compile **CompareClient**.)

Compile: InterfaceB CompareImplB
Bind:
Run: CompareImplB CompareClient

2.3.5 Importing and exporting

In the previous example, each program module was either a client or an implementor. Generally speaking, however, a **PROGRAM** module can be a client, an implementor, or both. Most commonly, a given **PROGRAM** module is both client and implementor. The module can import and export the same interface, or it can export one or more interfaces and import another (or several others.) The terms *client* and *implementor* refer more to the function of a module than to the module itself; there is nothing to prevent a client module from also being an implementor, or vice versa.

Figure 2.4 is a diagram of the communication between an interface and another module, which is both an implementor and a client of the interface. This diagram is merely a composite of the client/interface and the implementor/interface diagrams.

```
Interface
InterfaceName: DEFINITIONS =
  BEGIN
  ConstantName: CARDINAL = ...;
  ExportedProcedureName: PROCEDURE...;
  ImportedProcedureName: PROCEDURE...;
  END.
```

```
Implementor and Client
DIRECTORY
  InterfaceName USING [ConstantName, ImportedProcedureName]
InterfaceImpl: PROGRAM
  IMPORTS InterfaceName
  EXPORTS InterfaceName =
  BEGIN
  ExportedProcedureName: PUBLIC PROCEDURE... = BEGIN...END ;
  ...InterfaceName.ConstantName...;
  InterfaceName.ImportedProcedureName[] ;
  END.
```

Figure 2.4

2.3.6 System interfaces

System interfaces are general purpose interfaces that define comprehensive facilities for building everything from tools to whole systems. System interfaces serve as the entry point to an extensive library of procedures, variables, and data types, that saves you from reinventing and reimplementing utilities. Examples of system interface are **String**, which performs common string operations, and **Exec**, which handles communication with the Executive window.

System interfaces are nice because they provide so many useful utilities, but they have the attendant disadvantage that you must learn what interfaces are available, and what routines they implement. System interfaces that are part of Pilot (the operating system) are documented in the *Pilot Programmer's Manual*; interfaces that are part of the tools environment are documented in the *Mesa Programmer's Manual*.

You use symbols from a system interface just like private interfaces; you need to include the interface in the **DIRECTORY** clause and in the **IMPORTS** list, and refer to the symbol as **InterfaceName.Symbol**. In fact, system interfaces are just like all other interfaces except for one thing: the compiled versions of implementations of system interfaces are included in the XDE system bootfile. Thus, since the implementations are provided in the bootfile, you do not have to explicitly load implementation modules for system interfaces.

Recall from section 2.3.4 that when you use symbols from any interface, system or private, you must have the compiled version of the interface (not the implementation) on your local disk. If, for example, you want to use some procedures from the **Heap** interface (a system interface), you must make sure that Heap.bcd is on your local disk before you compile your program. Compiled versions of system interfaces are stored on a special directory, called the *release directory*; when you need to use a system interface, you will have to ask

someone where the release directory is and retrieve the appropriate object file for that interface from that directory.

Thus, to summarize: if you want to use procedures defined in the system interface **String**, you must import that interface and you must have the file String.bcd on your local disk when you compile your program (which is thus a *client* of the **String** interface), but you do not have to explicitly run the file that implements those procedures. In fact, you will not normally even know the name of the implementation file; remember, an interface is the link between programs, and the client need know nothing about the implementation.

2.3.6.1 An example of using system interfaces

To see an example, take another look at CompareClient.mesa, which uses procedures from several system interfaces. Here is the beginning of that program:

```
DIRECTORY
  FormSW USING [
    AllocateItemDescriptor, ClientItemsProcType, CommandItem, line0, line1,
    NumberItem, ProcType],
  Heap USING [systemZone],
  InterfaceB USING [Compare],
  Put USING [Line],
  Tool USING [Create, MakeFileSW, MakeFormSW, MakeMsgSW, MakeSWsProc,
    UnusedLogName],
  ToolWindow USING [TransitionProcType],
  Window USING [Handle];

CompareClient: PROGRAM IMPORTS FormSW, Heap, Put, Tool, InterfaceB =
...
```

CompareClient uses procedures from seven interfaces: six system interfaces and one private interface (**InterfaceB**). As you can see, the **USING** clause is a good way to document the exact symbols that this progam uses. Also notice that two of the interfaces are in the **DIRECTORY**, but not in the **IMPORTS** list. As discussed in section 2.3.3, this means that the symbols being used from that interface are compile-time values, and not run-time values.

2.4 Summary

Mesa's interfaces provide a formalized mechanism to allow individual modules to share types, constants, variables, and procedures. You can define your own interface, implement procedures declared in that interface, or use procedures implemented elsewhere. Interfaces thus encourage data abstraction and information hiding. As a quick review:

To *implement* a symbol defined in an interface you must:

- include the interface in your module's **DIRECTORY** clause;
- include the interface in your module's **EXPORTS** list;
- declare the symbol with the same name and type as appears in the interface;
- declare the symbol to be **PUBLIC**; and
- compile your module after the interface.

To *be a client* (use symbols defined in an interface), you must:

- include the interface name in the **DIRECTORY** clause;
- include the symbol in a **USING** clause
  (you do not have to have a **USING** clause, but it is a good programming habit);
- include the interface name in the **IMPORTS** list;
- use the symbol with its interface's name prefixed, as **Interface.Symbol**;
- compile the module after the interface has been compiled; and
- make sure the module that the implementation is available at run-time (loaded).

If you only use compile-time symbols, you do not need to **IMPORT** the interface.

Figure 2.5 on the next page summarizes the communication between an interface and its implementation and between an interface and its client. Implementations and clients are both **PROGRAM** modules, and a single module can function in both ways (although this is not shown in the figure.)

2.5 Questions

1) In what order must the following six modules be compiled? In what order must they be run?

   a) **Program1** is an implementation module that imports procedures from **Interface1** and **Interface2**. One of the procedures that it imports is implemented by **Program2**. **Program1** also exports a procedure to **Interface3**.

   b) **Interface1** is a definitions module.

   c) **Program2** is an implementation module that uses types from **Interface1** and exports a procedure to **Interface2**.

   d) **Interface2** is a definitions module that uses types from **Interface1**.

   e) **Program3** is a module that imports procedures from all three interfaces.

   f) **Interface3** is a definitions module

2.6 References

Chapter 7 of the *Mesa Language Manual* is essentially a denser statement of the information in this chapter and the next chapter.

Appendix A of the *Mesa Language Manual*, Pronouncing Mesa, tells you how to pronounce Mesa symbols.

Client

```
DIRECTORY
   InterfaceName USING [ProcedureName, ConstantName] ;
ClientName: PROGRAM
   IMPORTS InterfaceName =
   BEGIN...InterfaceName.ProcedureName[]; ... InterfaceName.ConstantName...END.
```

Notes:

1) This is a client module because it **IMPORTS** an interface.
2) The client can call procedures and use constants defined in the interface.
3) The interface must be listed in the **DIRECTORY**.
4) The procedures and constants must be in a **USING** clause.
5) The implementations of the procedures are bound at run-time, not at compile-time. The interface must be **IMPORT**ed.
6) The constants are bound at compile-time. The interface need not be **IMPORT**ed just to access them.

Interface

```
InterfaceName: DEFINITIONS =
   BEGIN
   ConstantName: CARDINAL = ...;
   ProcedureName: PROCEDURE ...;
   END.
```

Notes:

1) This is a interface module, as shown by the key word **DEFINITIONS**.
2) Interfaces can define constants that are available directly from the interface.
3) Interfaces can define procedures that are implemented by an implementation module.

Implementor

```
DIRECTORY
   InterfaceName ;
InterfaceImpl: PROGRAM
   EXPORTS InterfaceName =
   BEGIN
   ProcedureName: PUBLIC PROCEDURE ... = BEGIN ...END ;
   END.
```

Notes:

1) This is an implementation module because it **EXPORTS** an interface.
2) The ***InterfaceName*** must appear in the **DIRECTORY**.
3) The procedures being exported are declared as **PUBLIC**.
4) The **EXPORTS** list causes public procedures in this Implementation to be exported to the interface .
5) The module that implements interface *X* is conventionally called *X*Impl.
6) An implementation can also be a client provided the correct **DIRECTORY** ... **USING** clause is included. (see Figure 2.4.)

Figure 2.5

2.7 Exercises

Before beginning these exercises you should read Appendices A and B of this manual, which address Mesa syntax errors and debugger basics, respectively. Do the debugger exercises of Appendix B to start becoming familiar with the debugger.

2.7.1 Exercise in importing a procedure

Your assignment is to write a client program. We have provided an interface (**ReverseLettersDefs**) that defines a procedure, and an implementation module (**ReverseLettersImpl**) that supplies that procedure. The client module, which you should call ReverseLetters.mesa, will call the procedure **ReverseProc** from **ReverseLettersDefs**. **ReverseProc** in turn calls procedures that accept a character string from the user and output the string with the letters reversed.

Use the client template from Figure 2.5 to help you with this exercise. Once you have written your client program, compile the following modules (remember, an interface must be compiled before any modules that use it):

- ReverseLettersDefs.mesa -- the interface that defines **ReverseProc**
- ReverseLetters.mesa -- your client module
- ReverseLettersImpl.mesa -- the module that implements **ReverseProc**,.
- BasicIOImpl .mesa -- contains I/O procedures used by **ReverseLettersImpl**

Run the following modules

Run: BasicIOImpl ReverseLettersImpl ReverseLetters

BasicIOImpl implements procedures that are imported by **ReverseLettersImpl**, imported so it must be loaded before **ReverseLettersImpl**. When Tajo is ready, bring up the Tajo Executive window and type:

> ReverseLetters.~ hello *-- you type this*

The reversed letters are: olleh *-- the program returns this*

Experiment with reversing strings of letters and spaces.

2.7.2 Exercise in exporting a procedure

Now it's your turn to write an implementation module. You will write a procedure called **GetAverage** that computes the average of the integers passed to it. (You can do the average computation by any method, or do something else with the numbers, as long as you pass out an integer.) To keep the I/O simple, the average passed out of your procedure will be an integer value, and thus will be rounded up or down.

Your procedure will receive an array containing up to ten integers, and the actual number of integers to average. You will export your procedure **GetAverage** to the interface AverageDefs.mesa, which we provide. We also supply a client program to call your procedure and do the I/O.

After you have written your implementation module, compile the following modules:

- AverageClient.mesa -- this client program gets up to ten integers from the user, counts them, imports the interface **AverageDefs** to get your procedure, calls your procedure to compute the average of the numbers, and outputs the result.
- AverageDefs.mesa -- this is the interface that contains the definition of your procedure.
- AverageImpl.mesa (or whatever you called your implementation module).

Run the following files:

Run: AverageImpl AverageClient

Invoking **Run!** will put you into Tajo. Bring up the Executive and type:

>Average 2 4 -- *you type this*

The average is: 3 -- *the program returns this*

2.7.3 Exercise in importing and exporting using one interface

This exercise demonstrates importing and exporting using a single interface. First, you will import the interface **CombineDefs**. This imported interface provides the factorial routine **Fact**, which computes the factorial of a number for you. **CombineDefs** also contains some types and constants that you will need.

Your job is to write a procedure to compute a combinatorics problem, using the imported **Fact**. You will then export your procedure to the interface **CombineDefs** for a client to use. The client, which is provided for you, will create a tool window for you to enter data, and will use your code to compute a solution and display the result.

The first step is to write a procedure to calculate the following: Given a group of people of size **"baseSize"**, how many ways can you combine them into groups of size **"groupingSize"** ? The formula for this problem is

$$\frac{\text{baseSize!}}{\text{groupingSize! (baseSize - groupingSize)!}}$$

These variable names must be exact, and capitalization IS relevant. The name of your procedure will be **Combine**, and its type is **CombineDefs.CombineType**. You will find its definition in the interface **CombineDefs**. You will need to import **CombineType**, and the procedure **Fact** to perform the factorials from the interface **CombineDefs**. You will then export your procedure **Combine** to the interface **CombineDefs**.

Using CommandCentral, compile the following 5 modules:

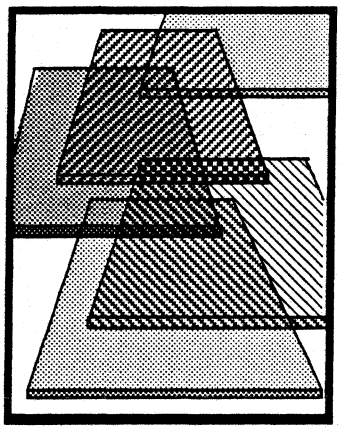
- CombineDefs.mesa -- the interface
- CombineImpl (or whatever you called it) -- the implementation module for **Combine**
- FactorialImpl.mesa -- supplies the factorial procedure for **Fact**
- CombinatoricsToolImpl.mesa -- supplies the user interface tool for the client
- CombineClient.mesa -- the client module

Run! the four implementation modules:

Run: CombineImpl FactorialImpl CombinatoricsToolImpl CombineClient

When you arrive in Tajo, you will see a tool window, which was produced by **CombinatoricsToolImpl**. Fill in the fields for **baseSize** and **groupingSize** and invoke **Combine!**. The answer will appear in the lower subwindow.

3



Binding

In the last chapter, we discussed how individual modules can use interfaces to share information. In this chapter, we will focus on how separately compiled modules are *bound* together into larger units.

3.1 Definition of terms

Configuration	A *configuration* is the bound code of one or more individual modules.
Configuration file	A *configuration file* is the file that contains the names of the modules that are to be bound together and describes how they are to be bound.
System interface	A *system interface* is an interface whose implementation is exported by the system bootfile.

3.2 Discussion

In the last chapter, you had to run several modules in a specific order to ensure that the implementation of an interface was available when a client program tried to reference it. This process is inconvenient, but manageable when there are few modules involved. When you are working on a large system, however, the job of keeping track of the necessary modules and their loading order becomes more difficult.

To help simplify things, the Mesa *binder* creates a logical structure called a *configuration* for the modules comprising a large system. This is analogous to the grouping of employees within a company. Groups of employees are organized into departments, with each department having certain duties. While the employees in a department do the actual work, the department itself can be thought of as doing the work, thus simplifying the world's view of things. Similarly, each configuration can be thought of as one logical entity that performs a certain task, although the task is actually performed by the modules within the configuration.

The binder processes a special file called a *configuration file*. This file contains a list of modules, which may be program modules or other configurations, and describes how they

are to be combined and initialized. The binder matches the import requests and export requests of the listed modules and creates an object module containing information about imported and exported items, object code for each module in the configuration, the names and versions of each module, and the interfaces referenced by those modules. This object module, the configuration, is also called a binary configuration description or "bcd" file.

There are several advantages to using a configuration instead of loading each module individually. One advantage is simplicity: after you have bound the modules together, you can type just the name of the configuration to run your program or system. Additionally, if other programmers want to use your system, they only need to obtain one module, the bound configuration, instead of finding and retrieving each individual module.

Another advantage of using the binder is version control. Every program module and definitions module has an associated time-stamp. This time-stamp can be thought of as an extension of the module's name; thus different versions of a module are different modules. For example, CompareImpl.bcd of Oct 14, 1984 1:15 p.m. is a different module from CompareImpl.bcd of Oct 15, 1984 10:12 a.m. When creating a configuration, the binder insures that all clients and implementors of an interface are referring to the *same version* of that interface; this effectively extends Mesa's strict type-checking across module boundaries.

3.2.1 A configuration file

The input to the binder is a *configuration file*, which contains a list of the modules to be bound, a list of imports and exports, and the order in which the modules are to be loaded. Here is Average.config, a configuration file for the program that you wrote in chapter 2:

```
Average: CONFIGURATION
IMPORTS Exec, String, Format, Heap
CONTROL AverageClient =
BEGIN
AverageImpl ;
AverageClient ;
END.
```

3.2.1.1 Reading a configuration file

Although **Average** looks much like a Mesa program, it is actually written in C/Mesa (configuration Mesa). There are five parts to a C/Mesa file:

(1) declaration (***Name*: CONFIGURATION**),
(2) **IMPORTS** list
(3) **EXPORTS** list
(4) **CONTROL** list
(5) **BEGIN-END** block

- The ***Name*** of the configuration file is the name that you will type to run your program after you have bound it.
- The **IMPORTS** list contains any interfaces that need to be imported from outside of the configuration; this is covered more fully in section 3.2.1.3.

- The EXPORTS list names all the interfaces for which this configuration exports an implementation. In this case, nothing is exported so there is no exports list. Exporting from a configuration is covered more fully in section 3.2.1.4 .
- The CONTROL list states which bound components are to be started and in which order. In most simple applications, only one component need be started explicitly. This is usually the component that contains mainline code. The other components are started implicitly when procedures in them are called.
- The BEGIN-END block itemizes the modules and configurations that are going to be bound together in the output configuration. This list corresponds to the list that you typed on the **Run:** line in the last chapter. In this case, the binder will use the information given in Average.config to bind together the files AverageClient.bcd and AverageImpl.bcd, and the resulting configuration will be stored in the file Average.bcd. The module names in the BEGIN-END block do not have to be listed in any particular order.

When you run the configuration **Average**, it will execute just as the individually loaded modules **AverageImpl** and **AverageClient** did in the chapter 2 exercise. If you want to try it, set up Command Central as follows and invoke **Go!:**

Compile:
Bind: Average
Run: Average

3.2.1.2 Importing into a configuration

The IMPORTS list of a configuration file is not simply a list of the imports of its components. It is a list of interfaces that need to be imported from *outside* the configuration. Interfaces that are imported by one module of the configuration and exported by another module in the same configuration are referred to as "self-contained" within the configuration, or "resolved." Such interfaces do not need to be imported by the configuration, but you must make sure that their implementation modules are listed in the configuration file.

The module **AverageClient** imports **GetAverage** from the interface **AverageDefs**, and the module **AverageImpl** supplies **GetAverage**. Thus, all the necessary information is available; **GetAverage** need not be imported into the configuration. The implementations for **Exec, String, Format**, and **Heap**, however, are not supplied by either of the modules being bound together, and must thus be imported into the configuration. (Recall from the last chapter that implementations for system interfaces are part of the bootfile, and are thus already loaded.)

3.2.1.3 Exporting from a configuration

Like the IMPORTS list, the EXPORTS list is not just a list of items exported by the components of the configuration. Putting an interface in the EXPORTS list of a configuration makes its symbols available to the world outside the configuration, just as putting an interface in the EXPORTS list of a module makes its symbols available outside the module. You can think of the bound configuration as a large module, composed of other, smaller modules. You get to choose which symbols you will make available to the outside world, and which you will

keep local to your configuration. You might want to keep all of your symbols local to your configuration, in which case you wouldn't even have an **EXPORTS** list.

One of the side effects of exporting an interface from a configuration is that the interface's implementation will remain loaded. (It thus has the same status as a system interface.) This means that the next configuration that imports the interface won't have to load the implementation module by listing it in the configuration file. Figure 3.1 illustrates exporting an interface from a configuration.

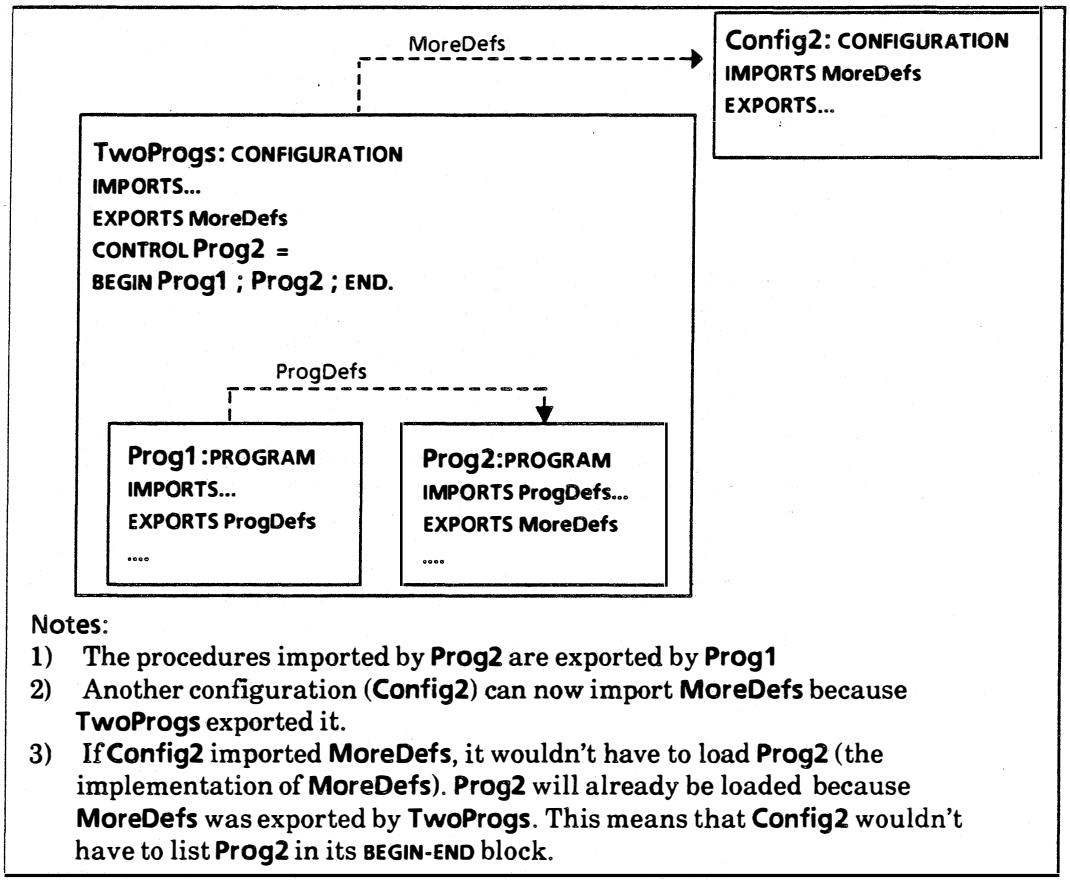


Figure 3.1 Exporting from a configuration

3.2.1.4 Template for a configuration file

Figure 3.2 is a general template for a configuration file.

Configuration
ConfigName*: CONFIGURATION** IMPORTS ***InterfaceA, InterfaceB,... EXPORTS ***InterfaceX, InterfaceY, InterfaceZ,...*** CONTROL ***Module1, ... =*** BEGIN ***Module1; Module2; ...*** END.
Notes: 1) This is a configuration because of the key word **CONFIGURATION**. The name of the source file should be ConfigName.config. 2) The configuration contains **Module1**, **Module2**, etc. **ModuleK** can be a program or a configuration. Order of module names within the **BEGIN**...**END** block is not important. 3) The **CONTROL** statement specifies the module that is to receive control when the configuration is started. (Also list there any modules that require explicit starting, but this is rarely necessary.) 4) ***ConfigName*** will import the interfaces listed in the **IMPORTS** statement. These interfaces should be all those imported within any **ModuleM** and not exported by another **ModuleN**. 5) ***ConfigName*** will export the interfaces listed in the **EXPORTS** statement. These interfaces must be exported by some **ModuleJ**. (You never have to export anything from a configuration, unless you want to make it available to others.)

Figure 3.2 Template for a configuration file

3.2.2 Unbound procedures

In XDE, a configuration can be run even if some of the procedures are not available, as when the exporting module has not yet been loaded. If a missing procedure is not called, everything runs without incident. However, when a missing procedure is called, a software interrupt named **UnboundProcedure** is generated. The program will not be able to continue and control will transfer to the debugger. If this happens, you should make sure that *all* of the modules necessary to run your program are listed in your configuration file, and add them if they're not there. Such errors are generally easy to debug.

3.2.3 Naming conventions

The *file name* is the name of the file in which you store modules, as in XYZ.mesa. The *module name* is the name that appears before the word **PROGRAM**, **DEFINITIONS**, or **CONFIGURATION**. It is *highly* recommended that you keep the file name the same as the module name (and remember that capitalization is significant.)

The name of a configuration file should be different from the names of the modules that it binds together. The reason is this: if you compile a module called XYZ.mesa, you get an object file called XYZ.bcd. If you bind this module to other modules using a configuration

file called XYZ.config, you get a bound configuration called XYZ.bcd, which overwrites the old XYZ.bcd. Consequently, you lose your compiled implementation of XYZ.mesa. By convention, implementation modules should have the suffix Impl, as in XYZImpl.mesa, to avoid this problem. Figure 3.3 illustrates this problem and its solution.

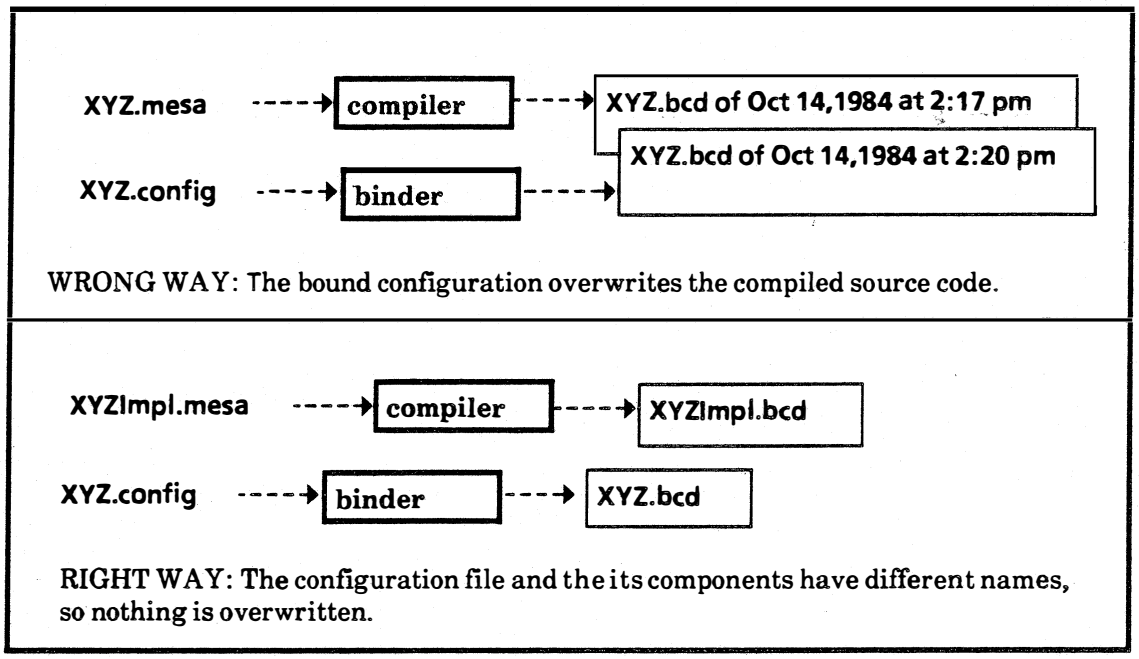


Figure 3.3 Naming conventions

3.2.4 System interfaces

As discussed in the last chapter, system interfaces are interfaces whose implementations are included in the bootfile. Thus, when you import a system interface, you do not have to include its implementation in your config file. The implementation is already bound into the bootfile, and will be available when you run your program. You do have to import the interface, but you do not have to include its implementation in your configuration, and you do need to have the copmiled version of the interface on your local disk.

3.3 Summary

This chapter discussed using the binder to produce bound configurations from a list of object modules. From the information in the "config" file and in each "bcd" file being bound, the binder can:

(1) resolve requests from modules for imported items
(2) combine a group of object modules into one larger object module
(3) control which interfaces are to be exported.
(4) determine which module is to be started first.
(5) maintain version control

Figure 3.4 gives a summary of the source file used by the binder, and its relationship to the modules that it binds together. This diagram also includes the use of system interfaces in program modules and in the configuration file.

Implementation Module

```
--this text stored in a file called ProgramNameImpl.mesa
DIRECTORY
 InterfaceName ;
ProgramNameImpl: PROGRAM
 EXPORTS InterfaceName =
 BEGIN
  ProcedureName: PROCEDURE ... = BEGIN ... END.
 END
```

Client Module

```
--this text stored in a file called ClientName.mesa
DIRECTORY
 InterfaceName USING [ProcedureName] ,
  SystemInterfaceName USING [SystemProcedure];
ClientName: PROGRAM
 IMPORTS InterfaceName, SystemInterfaceName =
 BEGIN ...
  InterfaceName.ProcedureName[] ;
  SystemInterfaceName.SystemProcedure[] ...
 END
```

Notes:
1) System interfaces are imported just like any other interface.
2) The module name should be the same as the program name, but not the same as any of the procedure names.

Configuration File

```
--this text stored in a file called ProgramName.config
ProgramName:CONFIGURATION
IMPORTS SystemInterfaceName
CONTROL ClientName =
BEGIN
 ProgramNameImpl ;
 ClientName ;
END.
```

Notes:
1) The name of the configuration file is not the same as the name of any of the modules that it binds together.
2) Implementation modules for the system interfaces are not listed.
3) There are no imports other than system interfaces because all of the imported interfaces are implemented by modules within the configuration.
4) Control goes to the module that has the mainline code, generally the client module.

Figure 3.4 Configuration file and Naming Conventions

3.4 References

Chapter 7 of the *Mesa Language Manual*, Modules, Programs, and Configurations, discusses configuration files and C/Mesa.

Chapter 17 of the *Xerox Development Environment User's Guide* discusses the binder and how to use it. This chapter also describes the binder's switches and error messages.

The *Mesa Programmer's Manual* and the *Pilot Programmer's Manual* give the details of the various system interfaces.

3.5 Exercises

3.5.1 Writing a configuration file and binding

For your first exercise, we have supplied a client program and two interfaces. Your job is to write a configuration file to bind the client with the implementations of the interfaces.

You will need the following files:

- ReverseWordsImpl.mesa -- the client program. It takes a string of input words (separated by spaces) from the user and reverses the order of the words.
- PrivateStorage mesa -- an interface defining storage allocation procedures
- BasicIODefs.mesa -- another interface
- BasicIOImpl.mesa -- the implementation for some of the procedures defined in the interfaces **BasicIODefs** and **PrivateStorage**.

The scenario looks like this: **ReverseWordsImpl** gets the definitions of the procedures it needs from the interfaces **PrivateStorage** and **BasicIODefs**. These interfaces in turn get the actual code for the procedures from the implementation module **BasicIOImpl**. Therefore, you need to write a configuration file that binds together the client program and the implementation module. The name of your configuration file should be Reverser.config. You will then run the entire program under the name "**Reverser**".

Remember, if you are binding two modules together and one of them exports the symbols that the other imports, you don't need to list the interface in the **IMPORTS** or **EXPORTS** list of the configuration file. You only need to list interfaces that are **IMPORT**ed from outside the configuration file (such as system interfaces).

3.5.2 Writing an interface

We're going to re-visit the combinatorics exercise. This time, instead of using **CombineDefs** to export **Combine**, you will write your own interface to define this procedure. Modify your implementation of **Combine** so that it exports the interface **MoreCombineDefs**, and write this interface so that it defines **Combine**.

You still need to import **CombineDefs** to use **Fact** and **CombineType**. However, you should now export **Combine** to **MoreCombineDefs**.

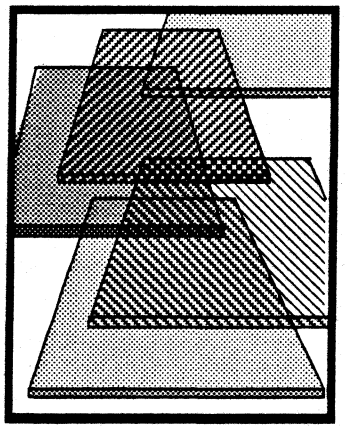
You must also modify the client module to import **Combine** from **MoreCombineDefs**.

Compile the following 3 modules:

- your interface (**MoreCombineDefs**)
- the modified client module (**CombineClient**)
- your modified implementation module (**CombineImpl**)

Write a configuration file, bind the necessary modules together, and run your configuration. Remember, you need all the same implementation modules that you needed last time you ran this program.

Notes:



Pointers

This chapter is an introduction to using pointers in Mesa. It covers what pointers are, how to perform common operations such as initialization and assignment on them, and how to pass them as procedure parameters. The next chapter, Dynamic Allocation, discusses how to allocate storage for the data that pointers reference.

There are a number of graphs throughout this chapter. They depict the memory in a hypothetical machine by representing each location in memory as a box. The number above the box is the memory location. The number in the box is the value stored in the location. The name below the box is the symbol in the example that has the associated value stored in the memory location.

4.1 Definition of terms

Pointer

A *pointer* is a reference to the location of a value. Mesa has *pointer types*, for pointers to specific types of values, and *pointer variables*, which contain the addresses of values rather than the values themselves. In Figure 4.1 below, **c** is a variable of type **INTEGER** containing the value 5. The variable **b**, a **LONG POINTER**, contains the address of **c**, and therefore **b** is a pointer to **c** and is said to reference **c**.

@

@ is the prefix "address of" operator. @x generates a reference to the expression x. In Figure 4.1, **b** contains the value **@c**, and so **b** is a pointer to **c**. Similarly, **a** contains **@b**, and so is a pointer to **b**.

Dereference

To *dereference* a pointer is to follow the pointer through one level of indirection toward the value it is referencing. Dereferencing a variable is the opposite of generating a reference to a variable. In other words, if **b** is a pointer to **c** then dereferencing **b** produces **c**. In Figure 4.1, dereferencing **a** once produces **b**, and dereferencing **a** twice produces **c**.

↑

In Mesa, ↑ is the postfix dereferencing operator. ↑ is the inverse of @, and is found at the opposite end of the expression. In Figure 4.1, **a** is **@b**, while **a**↑ is **b**, and **a**↑↑ is the same as **b**↑, which is **c**.

Dangling pointer

A *dangling pointer* is a pointer to an invalid memory location. A dangling pointer is usually caused by deallocating storage while a

pointer to it remains. Dereferencing a dangling pointer leads to unpredictable results.

Address fault

An *address fault* occurs when an attempt is made to reference an illegal address. For example, suppose that pointer **b** were not initialized to point to **c**, but instead left to be whatever value was in that location when **b** was allocated. If the value in the location is not a legal address, then dereferencing **b** causes an address fault. If, on the other hand, the address is legal, then you will *not* get an address fault. Rather, your pointer will be referencing some arbitrary location in memory, and you will be working with invalid data.

Frame

A *frame* is a Mesa processor data structure allocated while a module or procedure is executing to contain the variables and internal data structures for that module or procedure. Program frames are called *global frames*, and procedure frames are called *local frames*. Since Mesa supports recursion, there may be several frames for a particular program or procedure.

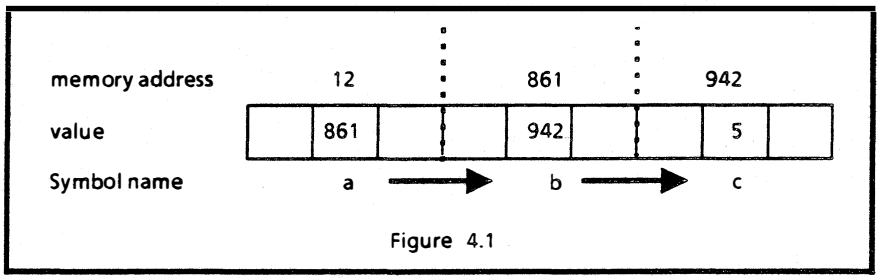


Figure 4.1

4.2 Discussion

Pointers are essential for good programming.

4.2.1 Declaring pointers

The Mesa architecture defines a uniform, paged virtual memory of 16-bit words. (A page is 256 words.) The entire virtual memory can be accessed by **LONG POINTER**s, which are two words long and can therefore address all 2^{32} locations.

Within this uniform virtual memory there is a distinguished region called the Main Data Space (**MDS**). Within the **MDS**, words may be addressed by **POINTER**s, which are one word long. The **MDS** is used internally to hold global and local frames. Therefore, all the pointers to storage that you allocate should be **LONG POINTER**s.

Pointers in Mesa are declared as references to types so that the Compiler can type-check their usage. The following example declares a pointer to an object of type **INTEGER**:

```
intPtr: LONG POINTER TO INTEGER;
```

4.2.2 Initializing pointers

Pointers allow indirect access to objects. In order for a pointer to be meaningful, the object it points to must exist. This means that storage has been allocated for the object, and has

been appropriately initialized. In the exercises in this chapter, the storage is allocated from the program's frame. Once an object is allocated and initialized, the @ operator is used to generate the pointer.

You can also allocate storage dynamically using the system's storage allocator; we will discuss this in the next chapter.

To initialize a pointer called **intPtr** to point to an **INTEGER** variable whose value is 5 you would write:

```
int: INTEGER ← 5;
intPtr: LONG POINTER TO INTEGER ← @int;
```

The first line allocates a space in the global frame and initializes it to 5. The second line initializes the pointer to the address of the storage location that contains the integer, as depicted in Figure 4.2 below.

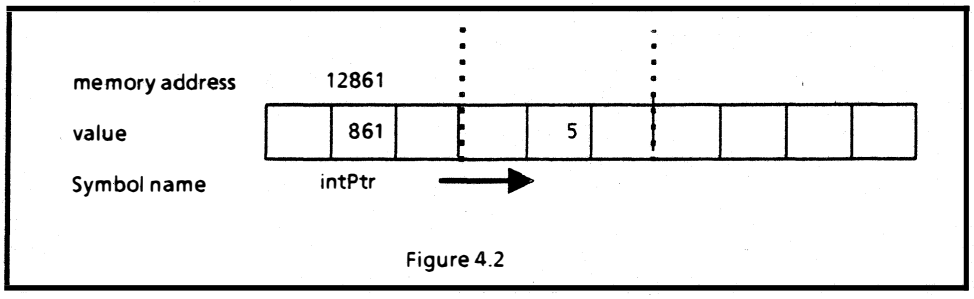


Figure 4.2

What if **intPtr** were initialized and **int** were not? As shown in Figure 4.3, the value for **int** would be meaningless, even though **int** is allocated. Pointing **intptr** to this location is valid, but not very useful.

```
int: INTEGER;
intPtr: LONG POINTER TO INTEGER ← @int;
```

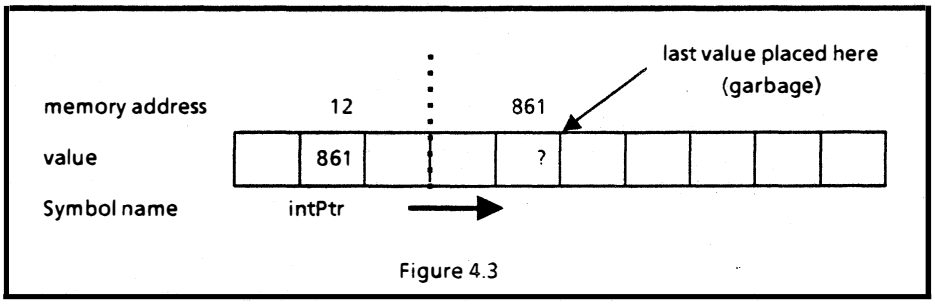


Figure 4.3

It is a good idea to avoid having pointers to uninitialized objects, lest you forget that the object is uninitialized and try to use the pointer. This would cause strange errors that are hard to debug. Instead, keep a pointer "uninitialized" until the object it will point to is initialized. Consider:

```
int: INTEGER;
intPtr: LONG POINTER TO INTEGER;
```

This recoding is one way of keeping your pointer uninitialized, but it suffers from the same problem as before. Now there are two uninitialized variables instead of just one, as illustrated in Figure 4.4.

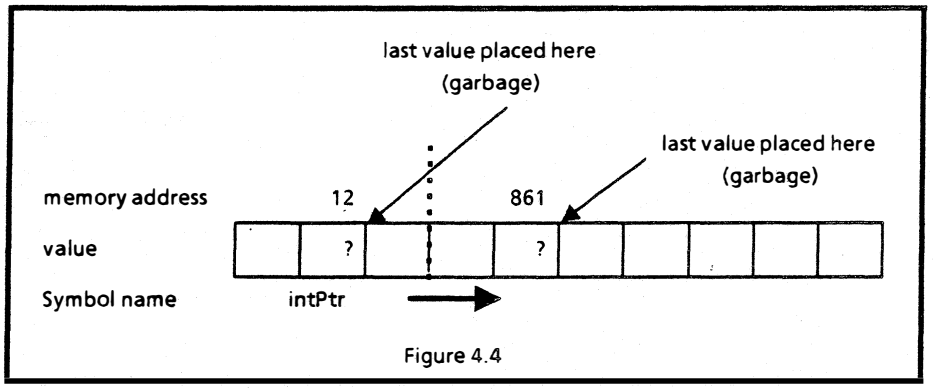


Figure 4.4

We have already discussed what might happen if you have a pointer to an uninitialized variable (such as **int**). If you try to dereference an uninitialized pointer, on the other hand, the value stored in the pointer's location would be interpreted as the address of a location. As shown in Figure 4.5 this pointer's value might point to a valid memory location in the address space. Dereferencing **intPtr** would therefore yield the garbage value 212 stored in memory location 942.

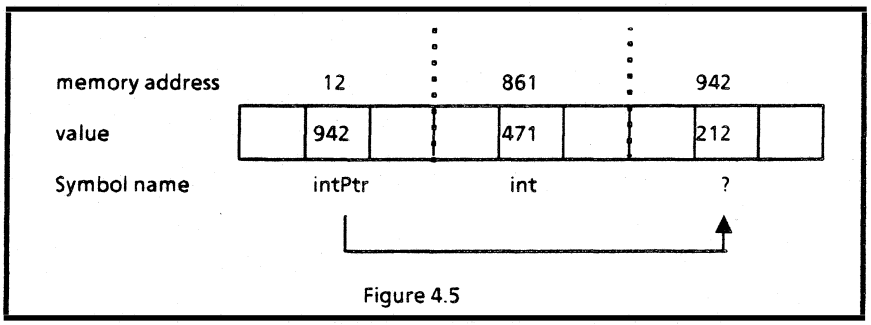


Figure 4.5

If, on the other hand, the value of **intPtr** pointed outside of the address space, to unavailable memory, then your program would address fault and the debugger would be called. In an environment that uses real memory addresses in code, this means that any address that points beyond the end of available memory would cause an address fault. However, the Pilot environment provides virtual memory. Addresses (that appear in code) are virtual and must be dynamically translated into real memory address at runtime.

During address translation, Pilot determines whether the page containing the reference is in real memory. If it is not, a page fault occurs and the page is swapped in from its backing file using available mapping information. An address fault occurs if the page to be swapped in is not mapped (has no associated backing store). Thus, in a virtual memory system, addresses that lie in the address space of a process can still cause address faults if they reference sections of the address space that are not mapped, as shown in figure 4.6.

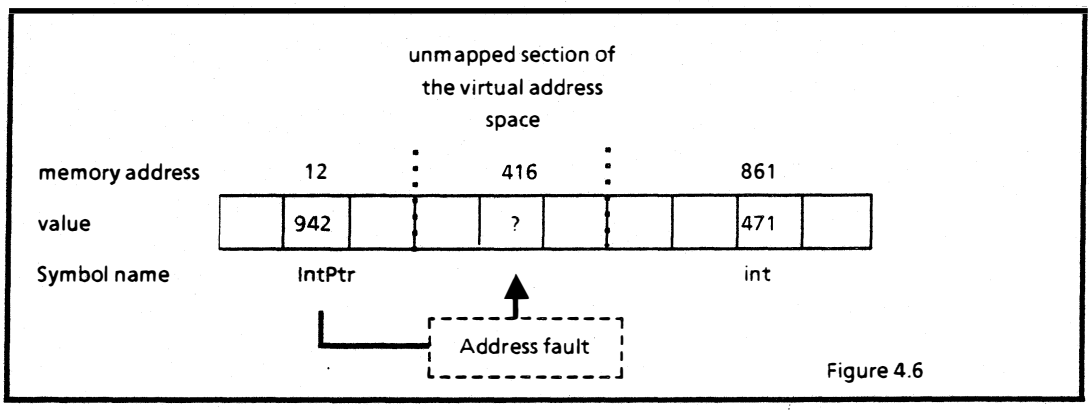


Figure 4.6

It is important to initialize all pointers, even those that have no referent. Mesa provides the special value **NIL** for this purpose. **NIL** signifies that a pointer does not point to anything valid and should not be dereferenced. Dereferencing a **NIL** pointer is undefined and will cause an address fault. When you are debugging, getting an immediate address fault is far better than having your program continue to execute with invalid data. In the latter case, your program may not malfunction until far from the scene of the crime.

```
int: INTEGER;
intPtr: LONG POINTER TO INTEGER ← NIL;
```

4.2.3 Assigning pointers

There are two common uses of pointers in assignment statements: assigning the address of a location to a pointer, as in the initialization of **intPtr**; and changing the contents of one pointers's referent to be a copy of another pointer's referent.

4.2.3.1 Assigning pointer values

In Mesa, pointers are type checked to the object they reference. This means that only pointers pointing to the same type of object can be assigned, as in this example:

```
int: INTEGER ← 5;
intPtr: LONG POINTER TO INTEGER ← @int;
anotherPtr: LONG POINTER TO INTEGER ← NIL;
anotherPtr ← intPtr;
```

The assignment of **intPtr** to **anotherPtr** is valid because they both point to an object of type **INTEGER**. After the assignment is complete, both **intPtr** and **anotherPtr** point to the same memory location. This has the same effect as if both pointers were individually assigned the address of **int**, like this:

```
int: INTEGER ← 5;
intPtr: LONG POINTER TO INTEGER ← @int;
anotherPtr: LONG POINTER TO INTEGER ← @int;
```

Figure 4.7 shows a before-and-after view of this assignment.

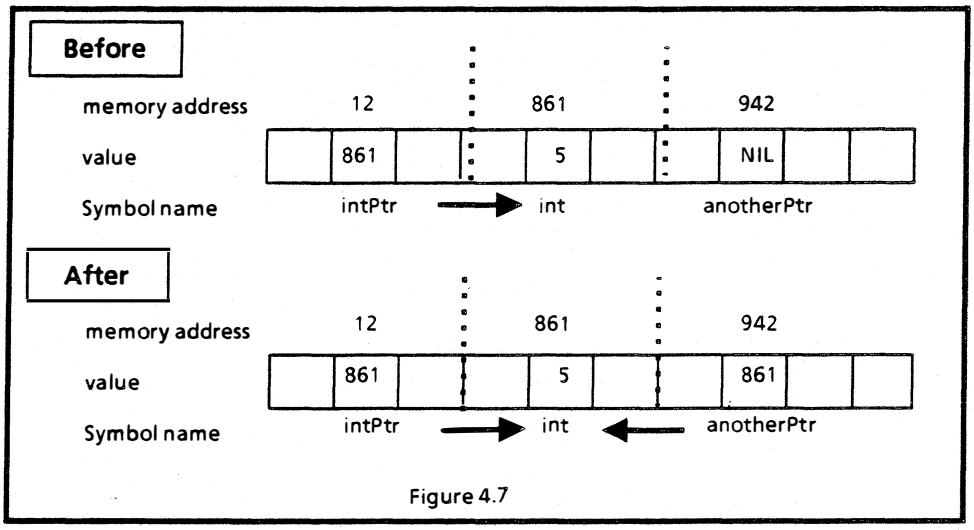


Figure 4.7

Now both **intPtr** and **anotherPtr** reference **int**. When **int**'s value changes, dereferencing either pointer will yield the changed value.

4.2.3.2 Assigning the contents of pointer references

Often, you do not want to share the value of an object, but you want to have two pointers that reference identical copies of one object. To do this, you dereference the pointers in the assignment statement:

```
int: INTEGER ← 5;
anotherInt: INTEGER ← 0;
intPtr: LONG POINTER TO INTEGER ← @int;
anotherPtr: LONG POINTER TO INTEGER ← @anotherInt;
anotherPtr↑ ← intPtr↑;
```

This assignment copies the value referenced by **intPtr** into the memory location referenced by **anotherPtr**. Changing the value in either of these two locations has no effect on the value pointed to by the other pointer. Figure 4.8 shows this situation.

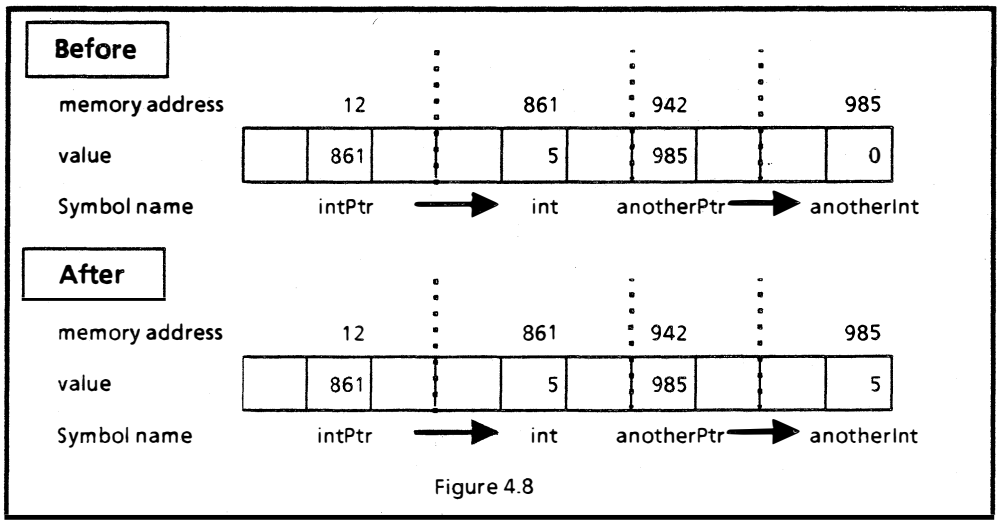


Figure 4.8

When you use pointers, be sure to think about the type of assignments you want your program to perform. If you accidentally share data between two or more pointers when you intend to copy the values, you will undoubtedly find some surprises when one pointer's referent is unexpectedly changed through another pointer. Conversely, copying data when you intend to share it will result in expected changes not taking effect.

4.2.4 Using pointers for parameter passing

There are two basic techniques of parameter passing: *call by reference* and *call by value*. In Mesa, all parameter passing is done as call by value. In other words, the variables passed as parameters to a procedure are not changed by what happens inside that procedure's body. For example, consider the procedure **DoNothing**:

```
DoNothing: PROCEDURE [a: INTEGER] =
  BEGIN a ← a + 1; END;
```

Assume that an **INTEGER int** has the value 5. When a program calls **DoNothing [int]**, the *value* of **int** is copied into **DoNothing**'s local variable **a**. When **DoNothing** changes the value of **a**, nothing happens to the value of **int**. Once **int**'s value has been copied into **a**, **int** is isolated from whatever goes on inside of **DoNothing**. Upon exit from **DoNothing**, **a** has the value 6 but **int** still has the value 5, as illustrated in Figure 4.9.

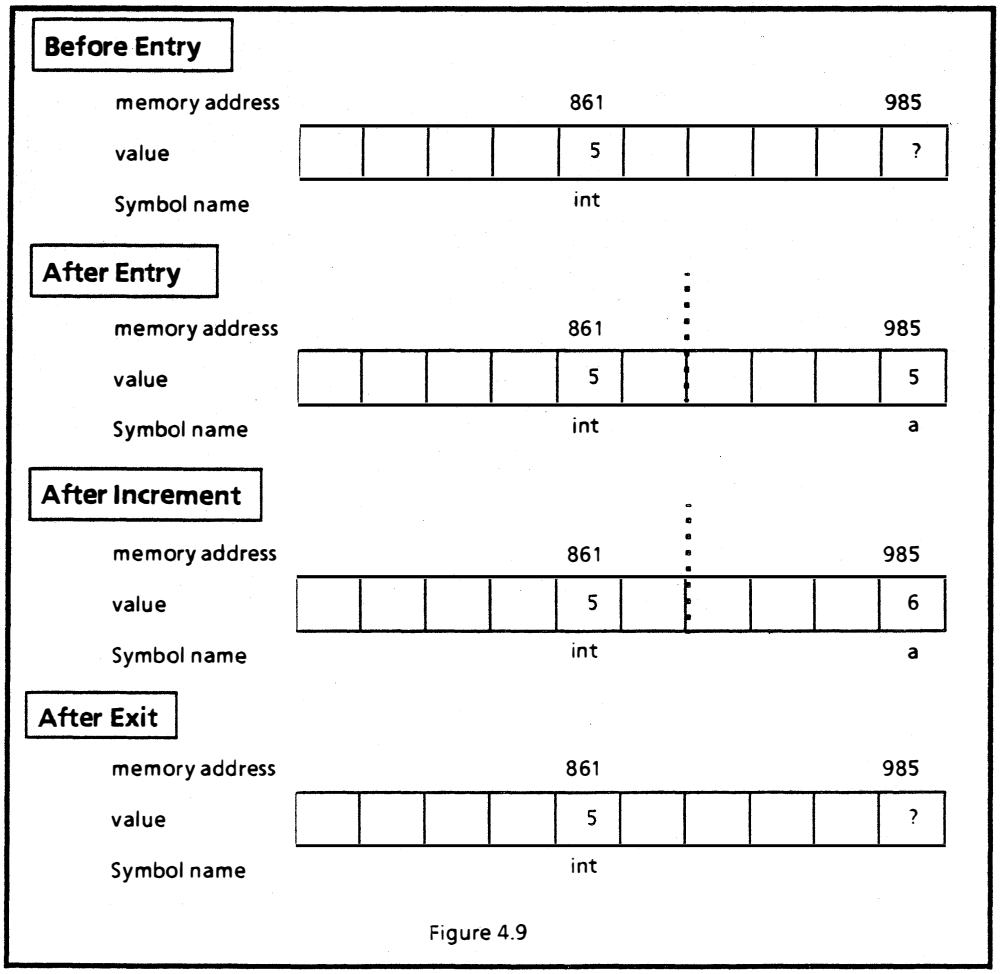


Figure 4.9

If Mesa did support call by reference and **DoNothing** was called so that its parameter, **a**, was a reference to the actual parameter, **int**, then **DoNothing** would have the desired effect of incrementing **int**. This manner of programming, where an argument to a procedure is changed as a side effect of the call, is considered bad form and discouraged in favor of having the procedure return the new value, as in:

```
DoSomething: PROCEDURE [a: INTEGER] RETURNS [INTEGER] =
  BEGIN RETURN [a + 1]; END;
```

Nevertheless, it is sometimes desirable for a procedure to modify one of its arguments. For example, a procedure may be called with a large array, several components of which need to be changed. If the array is so large that returning a copy of it would consume significant processor time and memory, then efficiency considerations may outweigh model programming, and the procedure might be designed to accomplish its end through side effects on its input.

When a procedure needs to have a side effect on one of its input variables, it takes as an argument not the variable itself but a pointer to that variable. After all, a pointer is a reference to where the value of the variable is stored. Given this reference (the address of the variable), a procedure can freely manipulate the contents of a variable by storing values into the location in memory where the variable's value resides. For example, a procedure **Increment** could look like this in Mesa:

```
Increment: PROCEDURE [a: LONG POINTER TO INTEGER] =
  BEGIN a↑ ← a↑ + 1; END;
```

To change the value of **int** by calling **Increment**, a program has to pass the procedure a pointer to **int**. When it makes the call **Increment[@int]**, the program makes the local variable **a** inside **Increment** point to **int**. Given such a call, **Increment** can change the value of the variable **int** by dereferencing the pointer **a**. Figure 4.10 illustrates the situation upon entry to the **Increment** procedure. The local variable **a** contains the address of the global variable **int**. When the assignment statement **a↑ ← a↑ + 1** is executed inside of **Increment**, the value of **int** is incremented. If **int** held the value 5 before the call **Increment[@int]**, then it will contain the value 6 immediately after the statement **a↑ ← a↑ + 1** is executed, as illustrated in Figure 4.10.

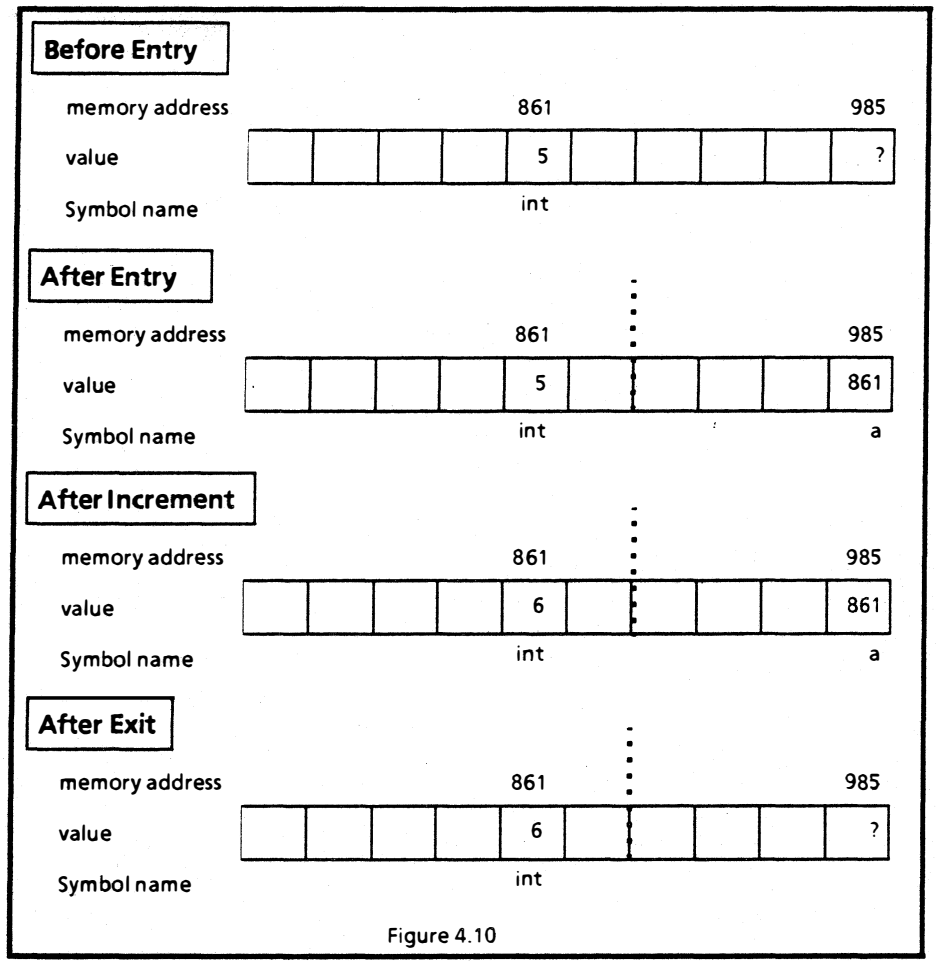


Figure 4.10

4.2.5 A common mistake: dangling pointers to local storage

When you asssign pointers to local values in procedures, you must not reference these values after exiting the procedure. Dereferencing a dangling pointer that used to point to a value allocated in a local procedure is undefined. The following example illustrates this.

`SimplePointer1.mesa` contains an instance of the **Increment** procedure discussed above. This program, when run, will work perfectly. Take a look at the code:

```
SimplePointer1: PROGRAM =
  BEGIN
  c: CARDINAL ← 0;
  worked: BOOLEAN ← FALSE;

  Increment: PROCEDURE [a: LONG POINTER TO CARDINAL] =
    BEGIN a↑ ← a↑ + 1; END; --Increment

  Unity: PROCEDURE RETURNS [b: CARDINAL] = BEGIN b ← 1; END; --Unity

  --Mainline Code
  c ← Unity[];
  Increment[@c];
  worked ← c = 2;
  END.
```

`SimplePointer2.mesa` tries to accomplish the same thing as **SimplePointer1**, but it takes a more devious approach. The code for **SimplePointer2** is slightly confusing, but looks like it will work when run. Unfortunately, the code is faulty. See if you can find the problem:

```
SimplePointer2: PROGRAM =
  BEGIN
  c: CARDINAL ← 0;
  worked: BOOLEAN ← FALSE;

  Increment: PROCEDURE [a: LONG POINTER TO CARDINAL] =
    BEGIN a↑ ← a↑ + 1; END; --Increment

  PointerToUnity: PROCEDURE RETURNS [b: LONG POINTER TO CARDINAL] =
    BEGIN d: CARDINAL ← 1; RETURN[@d]; END; --Unity

  --Mainline Code
  c ← PointerToUnity[]↑;
  Increment[@c];
  worked ← c = 2;
  END.
```

Look at the first assignment statement in the main body of **SimplePointer2**, the line: **c ← PointerToUnity[]↑;**. The intent is to dereference the pointer returned by the call to **PointerToUnity** in order to get the value 1. While **PointerToUnity** is executing, the situation is as depicted in the "Before Exit" part of Figure 4.11. The pointer **b** to be returned by **PointerToUnity** contains the address of the variable **d**, a variable local to **PointerToUnity**.

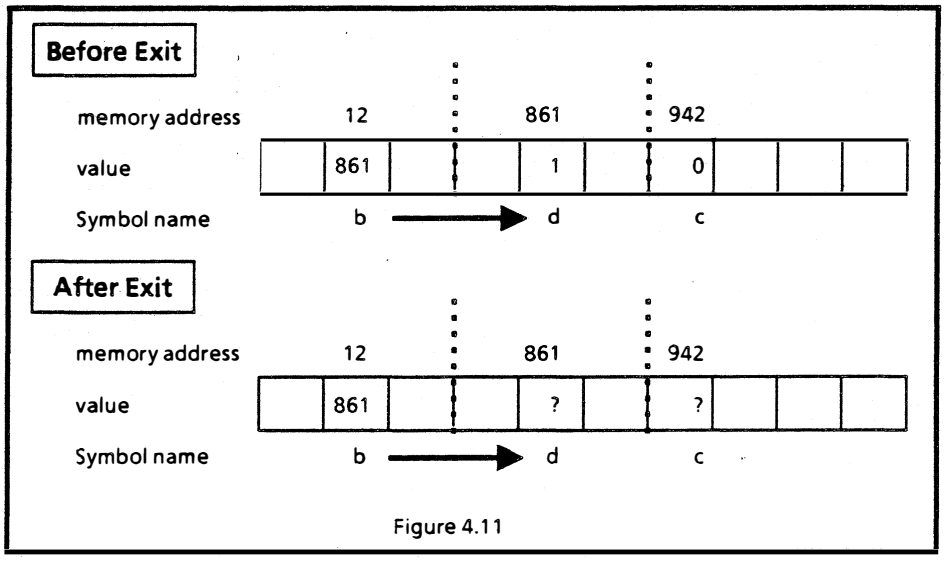


Figure 4.11

"After Exit" shows the situation after returning from **PointerToUnity**. The variable **c** should be assigned the value contained in the variable pointed to by **b**. But, now that **PointerToUnity** has been exited, the space used by **PointerToUnity** is considered by the system to be free space, ready to be overwritten as space is needed. Since **d** is local to **PointerToUnity**, it may already be overwritten now that **PointerToUnity** has been exited. The pointer returned by **PointerToUnity** points to where the value of **d** used to be. But **d** may be overwritten now, and so the pointer is worthless. When the program tries to assign

the value **@d↑** to **c**, it will be assigning a value that *might not* be the value that **d** had when **PointerToUnity** finished execution.

This procedure demonstrates the mistake of returning a dangling pointer to a local variable. When assigning pointers to values in local frames, be sure that the referents will still exist after the procedure has returned. One way to ensure this is to dynamically allocate space that outlives the local frame; this is the subject of the next chapter.

4.3 Summary

This chapter briefly discussed how pointers are used in Mesa programs. It presented a set of do's and don't's to keep in mind when programming with pointers, most notably:

- Do declare pointers as pointers to objects. This keeps you inside of the Mesa type checking system, which will go a long way in preventing pointer errors.
- Do initialize all variables including pointers. Having initialized variables will save you the trouble of worrying about whether or not a variable's value is valid. When you cannot initialize a pointer to an allocated and initialized piece of storage, signify this by initializing the pointer to **NIL**.
- Do be aware, when using pointers in assignment statements, whether you want the value shared between the two pointers (and therefore alterable by either pointer), or copied. To share the value between two pointers, assign the pointers (**ptr2 ← ptr1**); to copy the value, assign the dereferenced pointers (**ptr2↑ ← ptr1↑**).
- Do use pointers as arguments to procedures when you want the value of the caller's variable changed by the called procedure.
- Do **not** return pointers that point to a procedure's local variables.

4.4 References

Sections 3.3 and 3.4 of the *Mesa Language Manual* cover the syntax of record and pointer declarations, as well as detailing the operations that can be performed on pointers and records.

4.5 Questions

1) Assume that you are calling a procedure from an interface in order to get the next piece of input data from a file of **CARDINAL**s. Let's say that the **DataIn** interface contains three procedures, declared as follows, that can each get the next **CARDINAL** from the file.

```
GetNextValue1: PROCEDURE [nextValue: CARDINAL];
GetNextValue2: PROCEDURE [nextValue: LONG POINTER TO CARDINAL];
GetNextValue3: PROCEDURE RETURNS [nextValue: CARDINAL];
```

From looking at those declarations, determine which of the following calls will actually get the next piece of data from the file, and decide which call would be the best one to use in a Mesa program from a stylistic point of view.

```
i: CARDINAL ← 0;
DataIn.GetNextValue1[@i];
DataIn.GetNextValue1[i];
DataIn.GetNextValue2[@i];
DataIn.GetNextValue2[i];
@i ← DataIn.GetNextValue3[];
i ← DataIn.GetNextValue3[];
```

2) Given the type declarations below, explain what the differences between calling **AverageData1** and **AverageData2** are.

```
DataHandle: TYPE = LONG POINTER TO Data;
Data: TYPE = RECORD [
  interval, scale, length, maxlength: CARDINAL,
  data: ARRAY [0..0) OF CARDINAL];

AverageData1: PROCEDURE [dataToAverage: Data] =
  BEGIN
  FOR i: CARDINAL IN [0..dataToAverage.length - 1) DO
    BEGIN
    dataToAverage.data[i] ← (dataToAverage.data[i] + dataToAverage.data[i + 1] )/2;
    END;
  END;

AverageData2: PROCEDURE [dataToAverage: DataHandle] =
  BEGIN
  FOR i: CARDINAL IN [0..dataToAverage.length - 1) DO
    BEGIN
    dataToAverage.data[i] ← (dataToAverage.data[i] + dataToAverage.data[i + 1] )/2;
    END;
  END;
```

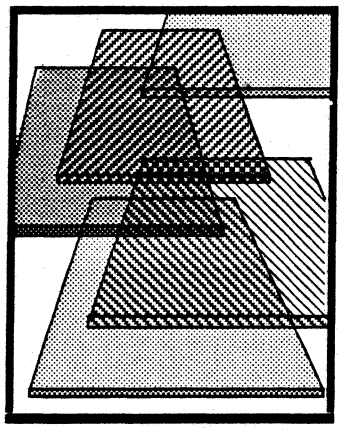
4.6 Exercises

1) Study Appendix D, which appears at the end of this course. It discusses how to debug address faults.

Write two procedures: **Compare**, which compares the values referenced by two pointers, and **Exchange**, which exchanges the value referenced by two pointers. You should declare your procedures to be of type **PointerDefs.CompareProcType** and **PointerDefs.ExchangeProcType**. Store your procedures in a file called CompareAndExchangeImpl.mesa.

To test your procedures, have your program call **PointerDefs.CreateCompareAndExchangeTool** passing the names of the two procedures. We have provided a config file (CompareAndExchangeTool.config) and the implementation for the tool (MesaCourseImplForCompareAndExchangeTool.bcd). Thus, you need to write your implementation, bind the config file, and run CompareAndExchangeTool.bcd.

5



Dynamic storage allocation and management

After reading the last chapter, you undoubtedly realized that pointers were not invented to point at just **INTEGER**s, when there're so many more interesting data structures in the world. Pointers can point at just about anything, including objects of undeterminable size at compile-time. Of course, constructs such as **CARDINAL**s, with their fixed known length at compile-time, can reside in a local or global frame, but what about a dynamic array or a string of characters? To allocate storage for constructs whose length or usage is not known at compile-time, you need dynamic allocation.

This chapter discusses how you allocate and deallocate storage dynamically, and suggests some ways for managing that storage effectively. We also discuss heaps, which are the storage allocators used for dynamic allocation.

5.1 Preliminary readings

Read the Pilot Memory Managment section (§ 4.6) in the *Pilot Programmer's Manual 11.0*. This section discusses zones and heaps.

Read § 6.6 in the *Mesa Language Manual 11.0*, entitled "Dynamic Storage Allocation." It discusses the Mesa operators **NEW** and **FREE**, which are used to allocate and deallocate storage.

5.2 Definition of terms

Dynamic allocation — *Dynamic allocation* acquires storage during program execution.

Dynamic deallocation — *Dynamic deallocation* releases space acquired through dynamic allocation.

Node — A *storage node*, or *node* for short, is a block of allocated storage, often with a record structure.

Storage Leak — A *storage leak* occurs when a program neglects to free all the storage nodes it has allocated, thus reducing the total amount of space available for the system. Leaked storage

degrades the system performance and in extreme cases can cause the system to crash.

Heap — A *heap* is a system-designated area of virtual memory used for dynamic allocation of storage. Heaps, which provide more automatic management of storage than zones, are designed to support the Mesa language operators **NEW** and **FREE**, which allocate and deallocate storage dynamically.

Valid memory location — A location is *valid* if it is currently allocated. A location that has been freed is invalid and should not be referenced.

Zone — A *zone* is a client-designated area of virtual memory used to acquire and manage arbitrarily sized storage nodes.

5.3 Discussion

Heaps are the primary storage allocators in Mesa. They are designed to allocate and free blocks of storage (nodes) of arbitrary size. A heap begins as one large free (unallocated) node somewhere in virtual memory. When a program requests storage, a node is allocated and a pointer to its location is returned to the requesting program. The program then moves values in and out of this node by indirect reference through the pointer. When the program no longer needs the storage, it returns the node to the heap's pool of available (free) nodes.

Clients interact directly with a heap by using Mesa's **NEW** and **FREE** operators and the facilities of the **Heap** interface. Clients use the **Heap** interface to obtain a heap (by either creating one or using one provided by the system) and to destroy a heap. Clients allocate storage from a heap with the **NEW** operator, and return storage to the heap when it is no longer needed with the **FREE** operator.

5.3.1 The system heap

Tajo provides a system-wide heap, called the **systemZone**, for all programs to share. If you need to share storage with other programs, the system heap is a good place to allocate the common storage. You should also use the system heap for programs that only allocate a small amount of storage. You will see an example of using the **systemZone** a little later in the chapter.

You access the **systemZone** through the **Heap** interface. For a program to allocate and deallocate nodes from the **systemZone**, it must **IMPORT** it from the **Heap** interface. Take a look at Section 4.6.2 of the *Pilot Programmer's Manual*, which describes this interface. **Heap.systemZone** is declared as an **UNCOUNTED ZONE**. (Think of this name as historic, not mnemonic.) The size of the **systemZone**, initially 40 pages, is bounded only by the amount of available virtual memory; it expands automatically when a request for storage is larger than the largest free node. The **systemZone** is created when a volume is booted and not destroyed unless the volume is rebooted. Misuse of this heap can be costly, since there is no garbage collection mechanism to free nodes that are no longer in use.

5.3.2 Private heaps

A program can create a private heap. Private heaps exist separately from the system heap, and only programs that have access to a private heap can allocate nodes from it. Like the system heap, private heaps can be grown to unlimited size, although they are typically bounded at 64K pages. The growth of an unbounded heap is limited only by available virtual memory.

Heap.Create is declared as follows:

```
Heap.Create: PROCEDURE[initial: Space.PageCount,
  maxSize: Space.PageCount ← Heap.unlimitedSize,
  increment: Space.PageCount ← 4,
  swapUnit: Heap.SwapUnitSize ← Heap.defaultSwapUnitSize
  threshold: NWords ← Heap.minimumNodeSize,
  largeNodeThreshold: NWords ← Space.wordsPerPage/2,
  ownerChecking: BOOLEAN ← FALSE, checking: BOOLEAN ← FALSE]
  RETURNS [UNCOUNTED ZONE];
```

Except for **initial**, the parameters have default values, which you will not (at this point) need to change. **initial** specifies the initial size of the heap, in pages. The system will automatically grow the heap as needed, in steps of **increment** up to **maxSize**.

You should destroy a private heap when you are finished with it. To destroy a private heap, call **Delete**, passing the zone returned by **Create**, like this:

```
Heap.Delete: PROCEDURE[z: UNCOUNTED ZONE, checkEmpty: BOOLEAN ← FALSE];
```

Delete has a second parameter to check if all the allocated nodes have been deallocated. This parameter, defaulted to false, prevents the accidental deletion of a heap still in use.

Space leaks are not as important in private heaps as they are in the **systemZone**, since deleting a private heap frees the entire space occupied by the heap and thereby reclaims any unfreed nodes. Any space leaks would be a potential problem only during the life of the private heap.

5.3.3 Allocating nodes: Using the NEW operator

A conventional way to allocate a node is to determine the amount of storage needed, and then ask the heap for a chunk of that size. The **NEW** operator does this, but it adds the protection of type checking for the allocated node by taking the type of the object as a parameter. It determines the size of the node that needs to be allocated, allocates it, and then returns a pointer to the allocated node.

Mesa enforces type checking on the returned value (the pointer). For example, if you were allocating a record of 3 **CARDINAL**s, your code would look something like this:

```
ptrToRecord: LONG POINTER TO Record ← NIL;
Record : TYPE = [ a: CARDINAL ← 0,
   b: CARDINAL ← 1,
   c: CARDINAL ← 2];
   . . .
ptrToRecord ← Heap.systemZone.NEW[Record];
```

The node allocated by the **NEW** operator (from **Heap.systemZone**) is of type **Record**. The pointer returned by **NEW** is thus a **LONG POINTER TO Record**. The variable on the left side of this assignment statement must conform to that type.

You can also initialize a node while allocating it with the **NEW** operator. To get the default initialization for **Record**, you could change the assignment to be:

```
ptrToRecord ← Heap.systemZone.NEW[Record ← []];
```

To override the default values, to set **c ← 10**, for example, you could write:

```
ptrToRecord ← Heap.systemZone.NEW[Record ← [c:10]];
```

5.3.4 Deallocating nodes: Using the FREE operator

The **FREE** operator takes a pointer to a node pointer as its parameter It frees the node and sets the value of the node pointer to **NIL**, as in

```
Heap.systemZone.FREE[@ptrToRecord];
```

Setting the pointer to **NIL** reduces the chances of creating a dangling reference. Figure 5.1 illustrates how **FREE** works. Without the extra level of indirection in **@ptrToRecord**, the system would not be able to change the value in **ptrToRecord** to **NIL**.

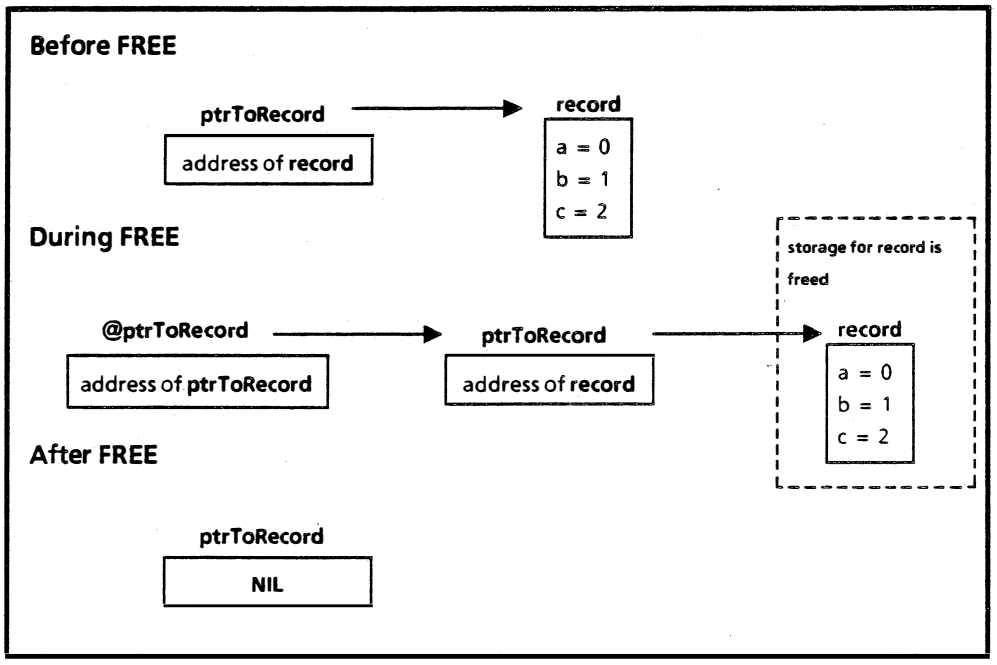


Figure 5.1 Using **FREE**

5.3.5 The systemMDSZone

The Mesa environment also provides a second system-wide heap. This second heap is called the **systemMDSZone**, and is used for allocating storage pointed to by **POINTER**s (whereas the **systemZone** is used for allocating storage pointed to by **LONG POINTER**s). The **systemMDSZone** exists inside a 256-page space called the Main Data Space (MDS), and is limited to that size. Since you will not ordinarily be using the **systemMDSZone**, this chapter discussed only the **systemZone**. However, the two heaps are functionally identical, and all observations about the **systemZone** apply also to the **systemMDSZone**.

5.4 Basic rules for storage management

So far, you've learned the definition of dynamic storage allocation and the procedures to manipulate storage dynamically. However, we haven't covered the best ways to supervise and manipulate space allocation and deallocation. If you had an infinite amount of resources (time and space), then management of those resources would be unnecessary, but since resources are limited and therefore considered to be precious, taking the time to understand storage management can improve your program's (and system's) performance. The following list represents general guidelines for efficient storage management. The rest of this chapter will discuss each item on the list in detail.

1. Hold onto storage only while you are using it.
2. Minimize the number of times you allocate any one item.
3. Keep global frames small.
4. Allocate temporary variables from local frames.
5. Avoid allocating string literals from the global frame.
6. Pass a pointer to an object as an argument rather than the object itself.
7. Use the **systemZone** when the total amount of allocated storage is small, and when use is over a short period of time.
8. Use a private heap when your program (or set of programs) require a lot of storage.
9. Avoid allocation from the **systemMDSZone**.

5.4.1 Hold onto storage only while you are using it

The actual space taken up by dynamically allocated objects is a precious resource, so you should only use it when absolutely necessary. Avoid allocating storage until you need it, and release that storage when you are no longer using it.

5.4.2 Minimize the number of times you allocate any one item

This rule really asks you to think about how a particular item is to be used in your program. When you learn about **SEQUENCE**s in the next chapter, you'll find that a dynamic array is implemented by copying different-sized arrays back and forth and changing the pointers to create the illusion of a dynamic array. The problem is that repeated allocations and deallocations take time and cause fragmentation within the heap. If you can determine the approximate use of the **SEQUENCE** in the program, then you can allocate a **SEQUENCE** that is, for example, four elements larger than what is currently needed, because you know that the **SEQUENCE** will need space for four more elements in the near future.

You might have noticed that this rule can conflict with the first rule of holding onto storage only while you are using it. You walk a fine line between the time issue and the

space issue and must make tradeoffs between the two to "optimize" your program. When making decisions about tradeoffs, keep in mind such issues as the size of the allocations, the use of the allocated space, and the length of use of the space.

5.4.3 Keep global frames small

Again, you are trying to conserve a precious resource. Global frames reside in the Main Data Space (MDS), a 256-page segment of virtual memory that can be directly addressed by short (16-bit) POINTERs. The MDS is heavily used by the run-time system, so you should avoid placing non-essential demands on it. As you may know, once a program is loaded it stays loaded until it is explicitly unloaded or until the system is rebooted. As a result, many global frames can exist in the MDS; thus the amount of free pages available for other programs to use decreases. Keeping global frames small helps to free the MDS for other tasks.

5.4.4 Allocate temporary variables from local frames

Besides the global frame, you can allocate space from a local frame and from heaps. Storage for local frames also comes from the MDS (see above). The difference between local and global frames (in terms of their burden on the MDS) is that a local frame remains allocated only as long as it is executing. When the procedure returns, the space for the local frame is released. Therefore, when you have fixed-size variables that are not needed for the life of the program, you should allocate them from local frames.

5.4.5 Avoid allocating string literals from the global frame

Suppose you need a string literal in the mainline code. If you allocate a string literal in the mainline code (with or without the **L** suffix), that literal will take up space in your global frame for the life of the program. To work around this problem, you should have the mainline code call a procedure that includes the code using the string literal. That way, the space for the string literal is released when the procedure finishes.

5.4.6 Pass a pointer to an object as an argument rather than the object itself

In Mesa, procedures pass arguments by value. In a procedure call, the parameters are copied into the local frame of the called procedure. Thus, passing a large object wastes both space and time. Avoid copying large objects in procedure calls by passing a pointer to an object instead.

5.4.7 Use the systemZone when the total amount of allocated storage is small, and when use is over a short period of time

The **systemZone** is created when the system is booted; a private heap, however, is created when your program makes a call to **Heap.Create**. The time needed to make this call can be significant when all you need is a small block of storage for a short period of time. For transient storage, the low overhead of using the **systemZone** is quite attractive.

5.4.8 Use a private heap when your program (or set of programs) requires a lot of storage

Private heaps have several advantages over public heaps. You can restrict the number of clients using a private heap, allowing faster access and minimizing fragmentation. You have potentially faster access because requests for storage must be monitored; thus, the fewer the clients, the less you have to wait in line for storage. Having a small number of clients reduces the amount that allocated nodes are spread around the heap. Since you have no control over where a block of storage is allocated from, the degree of dispersion of nodes wlll be large if the heap is large. The result of this is that a large heap will have very little of it mapped into real memory at any one time, and accessing the blocks of storage will cause more swapping than if they were allocated within a smaller heap.

5.4.9 Avoid allocation from the systemMDSZone

Since the **systemMDSZone** is contained within the MDS, allocations from this public heap compete with local and global frames for the bounded 256-page resource. The **systemZone** and private heaps, by comparison, are bigger and less congested.

5.5 Summary

This chapter discussed why you need dynamic allocation, and introduced heaps as the most common storage allocator for dynamically allocating nodes. To access the heap facility, you use the **Heap** interface (described in the *Pilot Programmer's Manual*). This interface provides two system heaps, as well as the mechanisms for creating and deleting private heaps.

You use the **NEW** operator to allocate nodes from a heap. When using **NEW**, you specify the heap the node should be allocated from and the type of the node to be allocated. The **NEW** operator calculates the size of storage needed, causes the allocation to occur, and returns a pointer to the node.

When your program is through with a node it must return the storage to the storage allocator. You do this with the **FREE** operator, passing a pointer to the pointer to the node. **FREE** deallocates the node and sets your pointer to **NIL**.

This chapter also presented some guidelines to help you manage storage allocation in a manner that will help your programs' performance. Most of the guidelines are common sense maxims that will help you use the system's time and space efficiently. The guidelines can be boiled down to two basic themes: don't waste time and space, and make a careful tradeoff when time and space issues conflict.

5.6 Questions

Assume that you are using an interface named **Node** that has procedures to allocate and free nodes of type **NodeType**, as defined below:

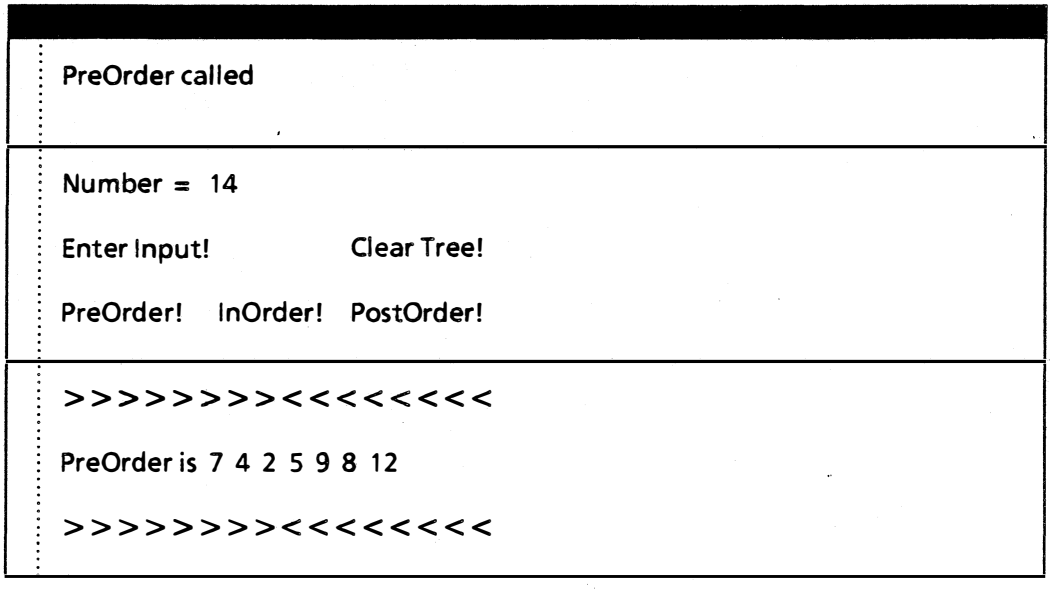
```
NodePtr: TYPE = LONG POINTER TO NodeType;
NodeType: TYPE = RECORD [
   start, end, size: LONG CARDINAL,
   duration: CARDINAL];
AllocateNode: PROCEDURE RETURNS [newNode: NodePtr];
FreeNode: PROCEDURE [nodeToFree: NodePtr];
```

Because the **FreeNode** procedure does not return **NIL**, you must set the **NodePtr**s to **NIL** with an assignment statment after you call **FreeNode**. Since the code frees nodes in many places, the following procedure was written to help free nodes. Does this procedure work as intended?

```
OurFreeNode: PROCEDURE [nodeToFree: NodePtr] =
   BEGIN
   Node.FreeNode[nodeToFree];
   nodeToFree ← NIL;
   END;
```

5.7 Exercises

The Tree Traversal Tool allows you to enter numbers into a sorted binary tree. At any point, you can make a preorder, inorder, or postorder traversal of the tree,with the order of traversal displayed in the tool. Your assignment is to complete the tool by writing the procedures **Init**, **EnterNumber**, and **ClearTree** in the module TreeTraversalProblem.mesa. The comments in this

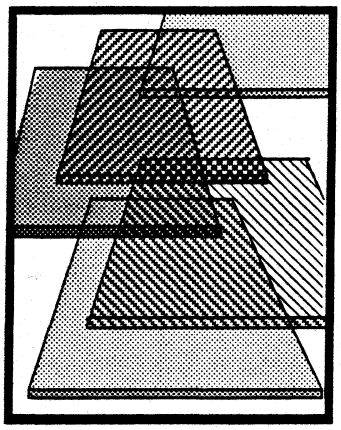


Tree Traversal Tool

module provide a more complete explanation of the procedures that you are expected to write.

You will also need the modules TreeProblem.config, TreeTraversalTool.mesa, and TreeTraversalDefs.mesa.

6



Sequences

Now that you know about heaps, it's time to look at one of the most common heap-dependent Mesa constructs: **SEQUENCE**s, the Mesa implementation of dynamic arrays. This construct allows you to defer specifying the size of an array until run-time. Because you don't know the size of a sequence until run-time, you have to allocate that sequence from a heap rather than in a local or global frame. This chapter discusses how to allocate, deallocate, and use sequences.

6.1 Discussion

One of the main advantages of using a dynamic array rather than a static array is that you don't have to commit your program to consuming storage before it uses that storage. A program does not allocate storage until it is actually ready to use that storage. You can also change the size of a dynamic array after it allocating it; this comes in handy when you find out sometime in the middle of your program that your sequence is too short. However, a corresponding drawback of using dynamic arrays is the amount of time it takes to allocate a dynamic array during run-time. Static arrays avoid this overhead since they're allocated when the program is loaded.

6.1.1 Declaring a Sequence

Sequences are always declared as the last field in a record. For example, the following declares a record structure that contains a sequence of **LONG INTEGER**s:

```
ptrToRecord: LONG POINTER TO Record ← NIL;
Record : TYPE = RECORD[
   a: BOOLEAN ← TRUE,
   b: BOOLEAN ← FALSE,
   C: BOOLEAN ← TRUE,
   seq: SEQUENCE length: CARDINAL OF LONG INTEGER];
```

The declaration of a sequence has a *variant tag* part (the **length: CARDINAL**) and an *element type* part (the **LONG INTEGER**). The type specification in the variant part determines the type of the indices used to select a sequence element. The range of valid indices is not specified when the sequence is declared but will be computed by the **FIRST** and **SUCC** functions when the sequence is allocated. This computation requires that the variant tag specify a valid

IndexType, as defined in the *Mesa Language Manual*. The element type defines the type of object that is being sorted in the sequence, thereby making sequences type-safe.

6.1.2 Allocating a Sequence

To allocate the record to contain a sequence of 10 elements, you could encode:

```
ptrToRecord ← Heap.systemZone.NEW[Record[10]];
```

Record[10] is a type specification describing a **RECORD** with a sequence part, **seq**, containing 10 **LONG INTEGER**s. The effect of **Heap.systemZone.NEW[Record[10]]** is to allocate **SIZE[Record[10]]** words of storage from the **systemZone** and return a **LONG POINTER TO Record** to this storage. All fields in the common part of the **RECORD** (the **BOOLEAN** fields **a**,**b**, and **c** in the example) are initialized to their default values if default values have been specified (**TRUE**, **FALSE**, and **TRUE** in the example). The sequence tag field, **length**, is set to 10, a value computed automatically using the formula:

```
length ← SUCC^10 [FIRST[CARDINAL]]
```

If the variant tag type uses an enumerated type or a subrange type whose first element is not 0, the value of **length** would still be the value of the tenth successor of the first element of the index set.

The index will range over **[0..10)**, a set of values computed using the formula:

```
[FIRST[CARDINAL]..SUCC^10 [CARDINAL] )
```

The elements of the sequence part are not initialized when the sequence is allocated. Initializing the sequence is your responsibility. However, you can use a constructor of type **Record** in the call to **NEW** to provide different initial values for the common part of the **RECORD** , as in:

```
ptrToRecord ← Heap.systemZone.NEW[Record[10] ← [a: FALSE]];
```

6.1.3 Using a Sequence

You can index individual elements of a sequence directly. For example, if **var** is of type **LONG INTEGER** ,then all of the following are equivalent:

```
var ← ptrToRecord ↑ .seq[3];
var ← ptrToRecord.seq[3];
var ← ptrToRecord[3];
```

Once you have allocated a sequence, you can use it as you would an array:

```
IF ptrToRecord.length > 5 THEN ptrToRecord[5] ← 13;
```

6.1.4 Deallocating a Sequence

You deallocate the record containing the sequence as you would any other node, by using the **FREE** operator:

```
Heap.systemZone.FREE[@ptrToRecord];
```

6.1.5 VowelSeparatorWithPublicHeap

VowelSeparatorWithPublicHeap is an example of dynamically allocating records with sequences in them. The program, which runs from the Executive, separates user input into vowels and consonants. A sample input would be:

```
VowelSeparator.~ separate the letters in these words by vowels and
consonants
```

Try running the program now.

6.1.5.1 TextSeqBody: the data structure used for storing text

The input is stored in the **TextSeqBody** data structure, which is defined in the **SequenceDefs** interface as:

```
TextSeqBody: TYPE = RECORD [
  length: CARDINAL,
  text: SEQUENCE maxlength: CARDINAL OF CHARACTER];
```

The **length** field specifies the number of elements currently stored in the sequence. The **text** field defines the sequence of characters where the input is stored. The **maxlength** tag field specifies the maximum number of characters that can be stored in the sequence.

TextSeq is a pointer type to this record object, defined as:

```
TextSeq: TYPE = LONG POINTER TO TextSeqBody;
```

6.1.5.2 The procedure Main

In **VowelSeparatorWithPublicHeapImpl**, the procedure **Main** controls translating the input into a **TextSeqBody** and separating the characters into vowels and consonants. However, since the program runs from the Executive, no call to **Main** appears in the program. Instead, the mainline code calls **Init**, which subsequently calls **InitializeVowelSeparator** (from the **SequenceDefs** interface). **InitializeVowelSeparator** registers the program with the Executive, telling it that **Main** is the procedure to call when a user types the `VowelSeparator.~` command. It is important to remember that the procedure, not the whole program, is executed when the command is invoked.

Let's assume a user types into the Executive

```
VowelSeparator.~ separate the characters in these words
```

The Executive recognizes the command and calls **Main**. **Main** declares three variables, **input**, **vowels**, and **consonants**, of type **TextSeq**. These variables will point to **TextSeqBodys** containing the input, the vowels in the input and the consonants in the input. The variables **vowels** and **consonants** are initialized to **NIL**.

SequenceDefs.GetText stores the user's input in **input** and then translates it into a **TextSeqBody**. Because **GetText** must allocate the **TextSeqBody**, we pass the **systemZone** as a parameter to **GetText**. Passing the zone ensures that all nodes are allocated from the same heap. Figure 6.1 depicts the situation at this point.

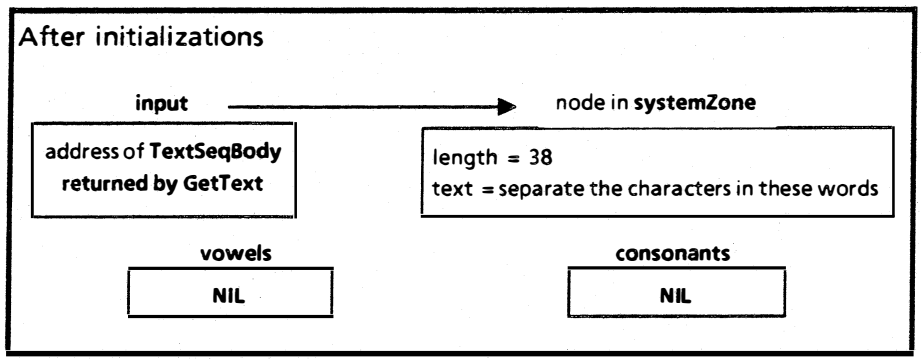


Figure 6.1

Following these initializations, **Main** calls **Separate** to sort the input line into vowels and consonants. **Separate** creates (allocates) two **TextSeqBody**s and returns a pointer to each of these **TextSeqBody**s. Figure 6.2 represents the situation after **Separate** has returned.

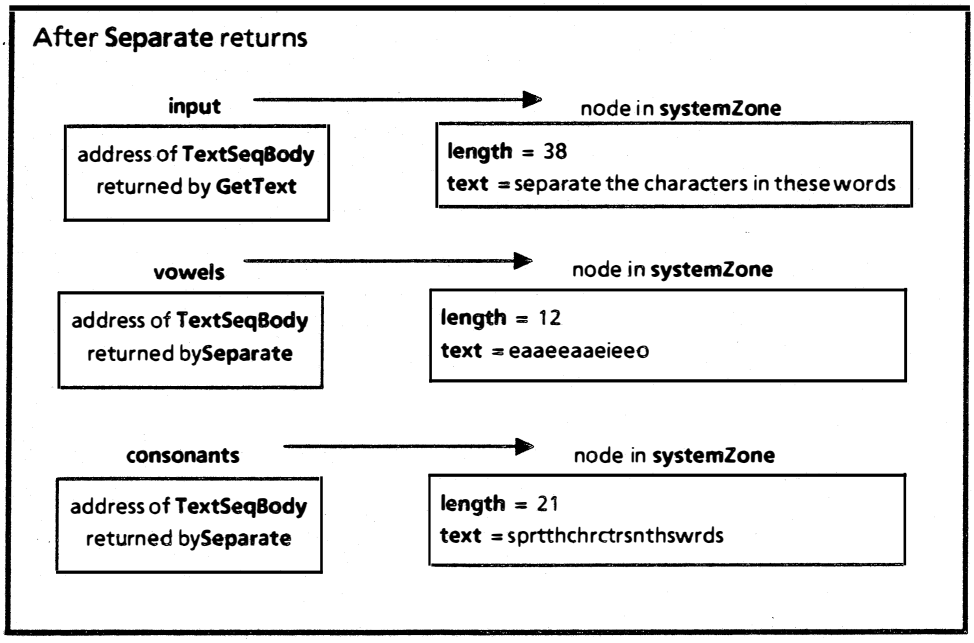


Figure 6.2

Main now outputs the separated characters, first checking to see if there is anything to print. It uses **SequenceDefs.PutComments** and **SequenceDefs.PutText** to print to the Executive. (**PutComments** outputs string literals; **PutText** outputs a **TextSeqBody**.)

Next, **Main** frees the **TextSeqBody**s that were allocated and passed to it:

```
FreeTextSeq[@input];
FreeTextSeq[@vowels];
FreeTextSeq[@consonants];
```

Figure 6.3 shows that all allocated storage is freed before **Main** returns.

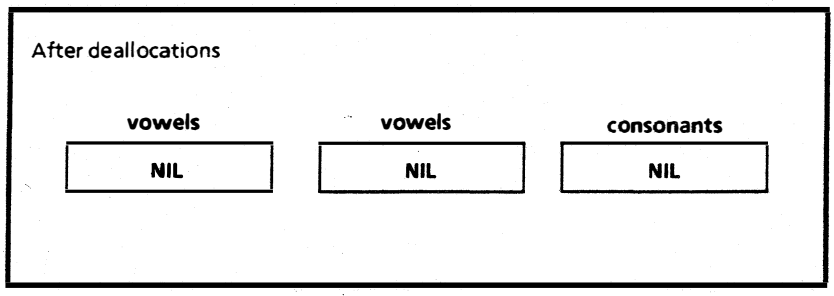


Figure 6.3

Note: Use the information presented in the last chapter (Dynamic Storage Allocation and Management) to figure out the reason for freeing the **TextSeqBody** nodes in this procedure as well as in **AppendChar**

6.1.5.3 How the input is separated

Separate and **AppendChar** are the procedures primarily responsible for separating the characters. **Separate** defines the algorithm for separating the characters; **AppendChar** adds a character into a **TextBodySeq** object.

Separate takes a parameter of type **TextSeq** and separates the characters into two sequences, one containing vowels and the other containing consonants, and returns pointers to each of these **TextSeqBody**s. We use the following algorithm: check if the next character in the input line is alphabetic; if it is, check the alphabetic character to see if it is a vowel. If the character is a vowel, we append it to the **vowels TextSeqBody**. Otherwise, we append it to the **consonants TextSeqBody**.

Note: In the implementation of this algorithm, **Separate** allocates storage for **vowels** and **consonants** from a reasonable guess of vowel and consonant distribution. We did this to minimize the number of allocations done by **AppendChar**.

AppendChar builds the vowel and consonant sequences by adding a character to the end of a **text** sequence. If the **text** sequence is not full (i.e., **length** is less than **maxLength**), then the character can just be appended (by entering it as the next element in the sequence and incrementing **length**).

However, if the **text** sequence is full, the situation is more complicated. **AppendChar** cannot add the next element because there is no room left in **text**. Trying to store into the sequence will cause a run-time error if you compiled with the b switch (bounds checking). If there is no bounds checking, the append will be done, but the element will not be stored into a properly allocated memory location. Instead, it will be stored just beyond the end of the allocated storage. This location could be undefined (causing an address fault), currently allocated for another node (smashing memory by writing over other data), or unallocated (with no assurances on how long the location will stay unallocated and its contents unchanged).

To avoid this situation, you must allocate a new **TextSeqBody** when the sequence is full. (This is how to "grow" a sequence.) You must then copy the contents from the old sequence into the new one. This is what **AppendChar** does; take a look at the code for this procedure.

The series of graphs in Figure 6.4 illustrates the expansion of the sequence when **AppendChar** is asked to append the letter **e** to a full **TextSeqBody**.

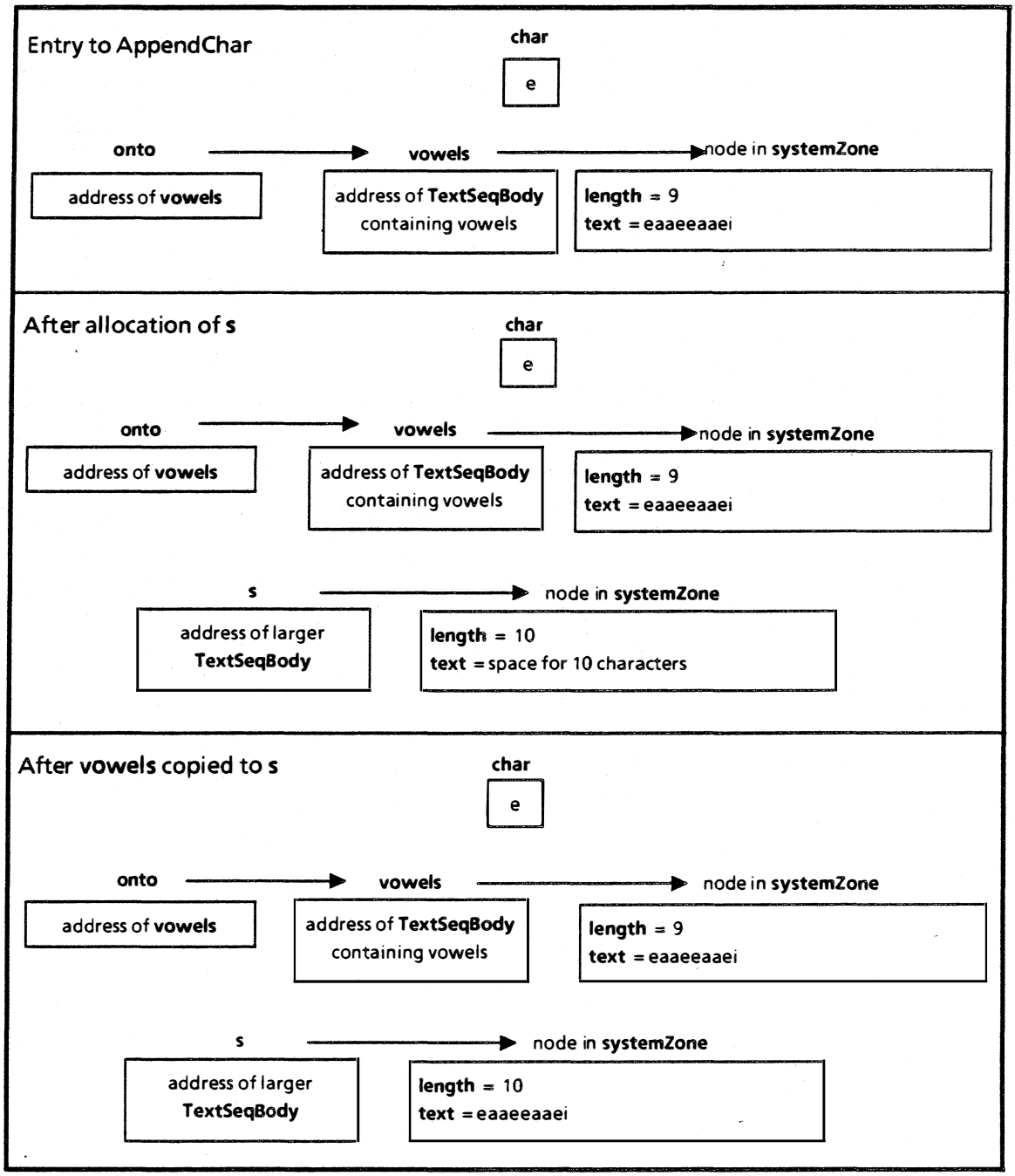


Figure 6.4

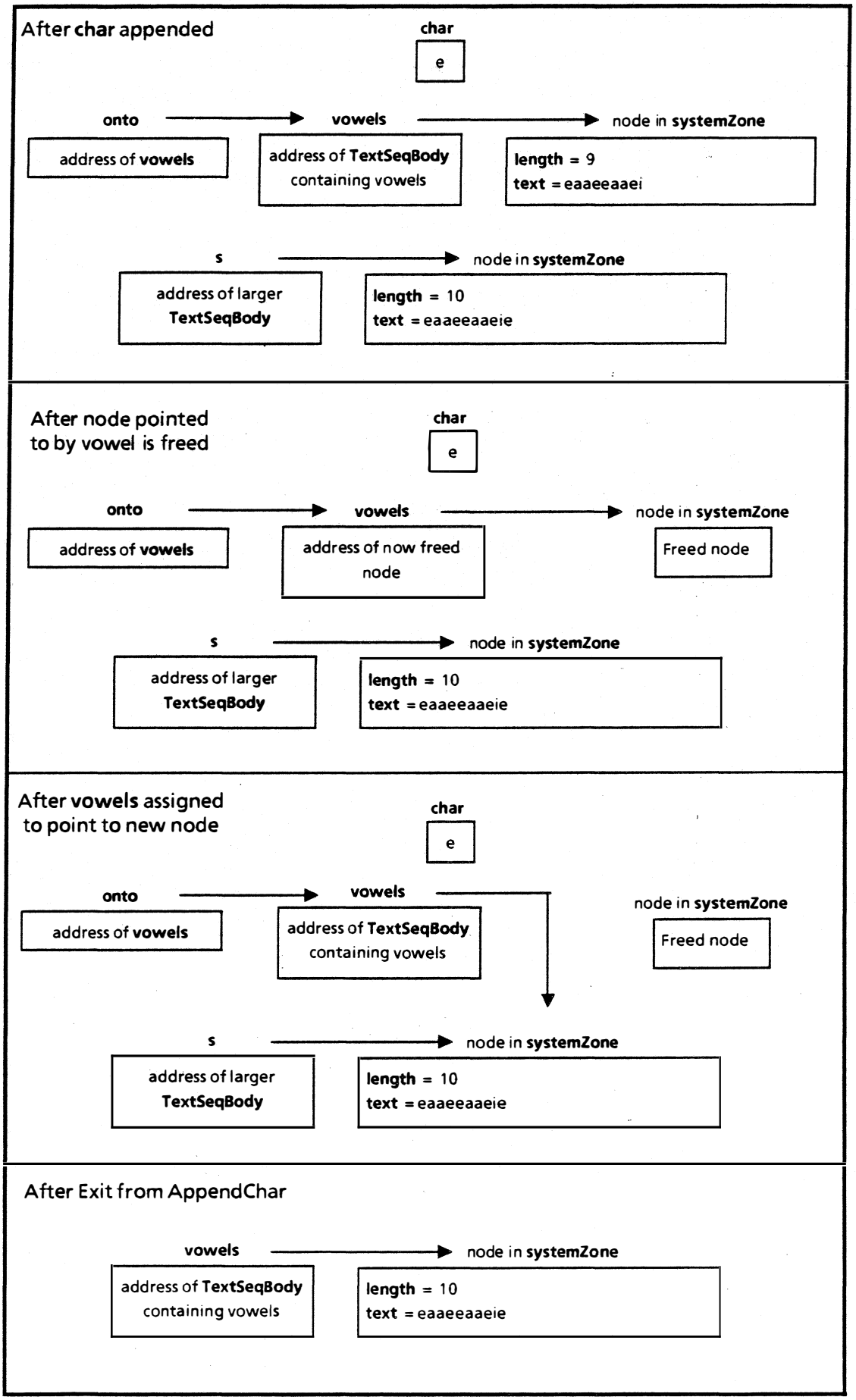

After char appended
char
e
onto
vowels
node in systemZone
address of vowels
address of TextSeqBody containing vowels
length = 9
text = eaaeeaaei
s
node in systemZone
address of larger TextSeqBody
length = 10
text = eaaeeaaeie
After node pointed to by vowel is freed
char
e
onto
vowels
node in systemZone
address of vowels
address of now freed node
Freed node
s
node in systemZone
address of larger TextSeqBody
length = 10
text = eaaeeaaeie
After vowels assigned to point to new node
char
e
onto
vowels
node in systemZone
address of vowels
address of TextSeqBody containing vowels
Freed node
s
node in systemZone
address of larger TextSeqBody
length = 10
text = eaaeeaaeie
After Exit from AppendChar
vowels
node in systemZone
address of TextSeqBody containing vowels
length = 10
text = eaaeeaaeie

Figure 6.5

6.1.6 VowelSeparatorWithPrivateHeap

VowelSeparatorWithPrivateHeapImpl differs from **VowelSeparatorWithPublicHeapImpl** only in that it uses a private heap instead of the **systemZone** to allocate **TextSeqBody**. This module is part of the configuration called `VowelSeparatorWithPrivate-Heap.bcd`. It runs from the Executive command `VowelSeparator.~`. Run the program to verify that it acts like **VowelSeparatorWithPublicHeap**, and then study `VowelSeparatorWithPrivateHeapImpl.mesa`. Pay particular attention to the creation and deletion of the private heap, and to the allocation and deallocation of nodes.

6.2 Summary

A sequence appears as the last field in a record. It contains a variant index field in its declaration, which becomes fixed at the time of allocation. To enlarge a sequence, therefore, you must:

1) allocate a new, larger one,
2) copy the data from the full sequence into the new one,
3) free the old sequence , and
4) adjust the pointers so the new sequence is referenced by the pointer that referenced the original sequence.

6.3 Reference

The *Mesa Language Manual 11.0* section entitled "Sequences" is a thorough reference.

6.4 Exercises

Complete a program that takes a string of characters as input and stores the characters alphabetically in queues according to the number of queues that the user specifies. For example, if the input were ***James! Where are you?!***, and the user wanted four groups of characters, the result would look like this:

For Group 0 (A-G):
a e e e a e

For Group 1 (H-N):
J m h

For Group 2 (O-T):
s r r o

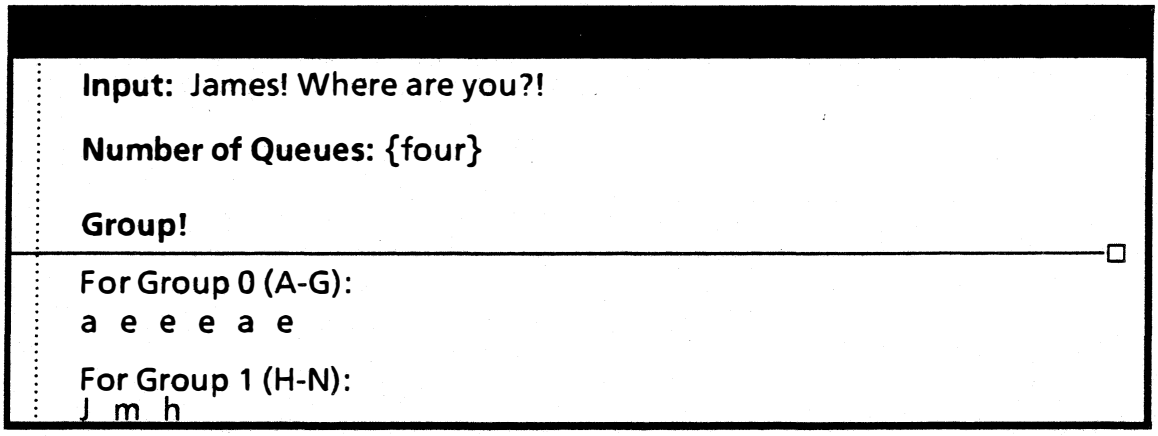
For Group 3 (U-Z):
W y u

For Last Group (non-alphabetic characters):
! SP SP SP ? !

Done.

The program runs from a tool, which consists of the following modules:

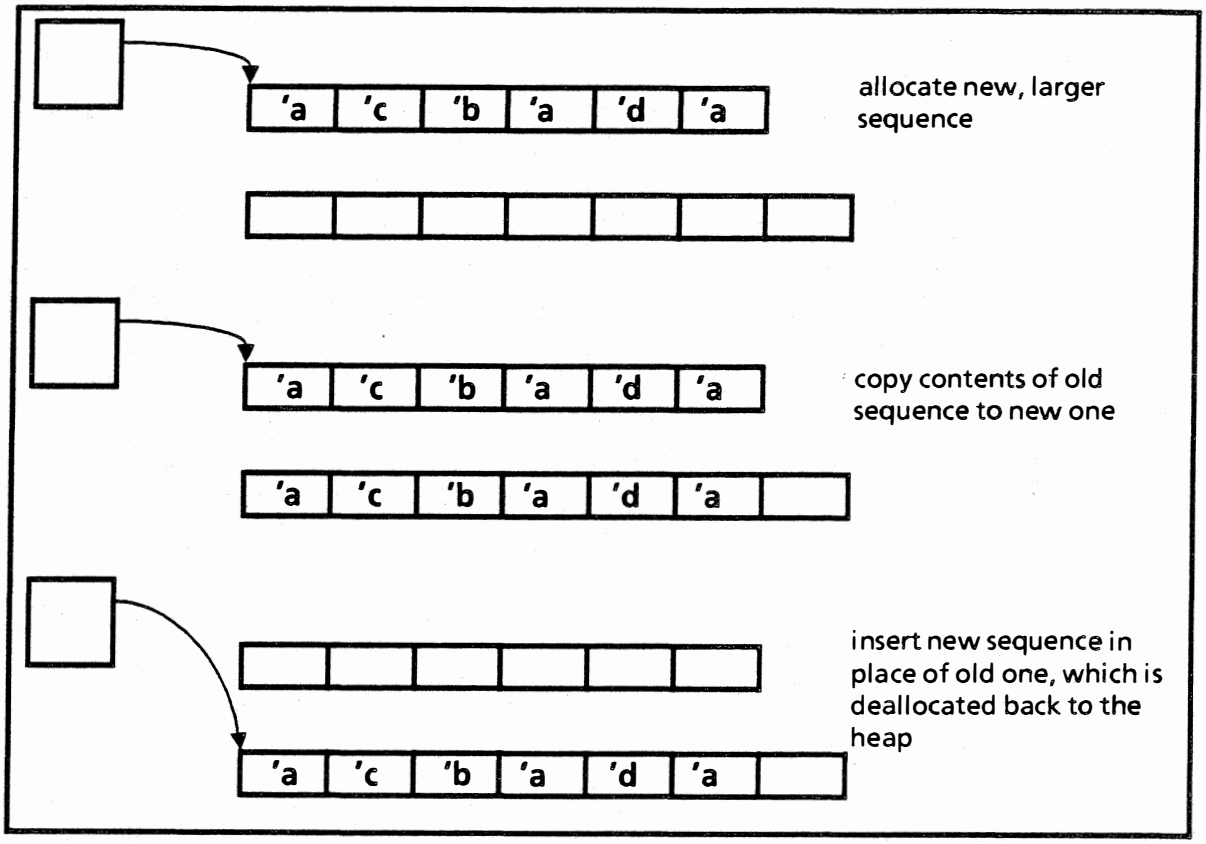
LetterTool.mesa: contains tool-related code (I/O);
LetterImpl.mesa: contains the implementation code that actually processes the input;
LetterDefs.mesa: is the interface for these modules;
LetterConfig.config: is the configuration module for the above.



The tool as it appears when *LetterConfig.bcd* is executed.

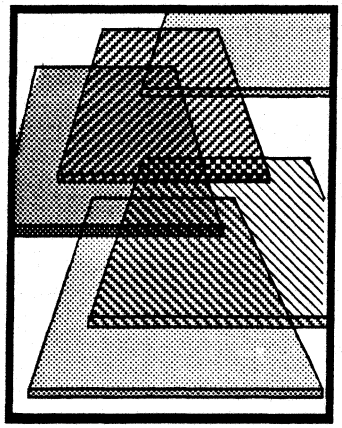
When **Group!** is invoked, the CommandItem procedure **Group** (in **LetterTool**) passes the input string and the number of desired queues to procedure **ProcessInput** (in **LetterImpl**). **ProcessInput** calls **InitQueues** to create and initialize the queues. It then calls **CutUpAlphabet** to determine which characters each queue will handle. **ProcessInput** then calls **StoreLetters** to actually put the characters into the queues. Finally, **PrintResults** (in **LetterTool**) is called to display the results of the user-requested action.

There are two instances where you must consider dynamic storage allocation. First, there is the initial allocation from a heap, where two factors are variable: the number of queues and the size of each queue. Secondly, there is the expansion of a queue when the sequence that represents the queue is full. The "expansion" really consists of allocating a new sequence that is larger than the original one, copying over the original sequence into the new one, inserting the new sequence in place of the original one, and freeing the space that the original sequence occupied (see diagram on next page).



"Expansion" of a sequence

7



Strings

In this chapter we introduce Mesa strings. Although you may not have realized it, the classic implementation of a string as an array of characters with an associated length actually involves a pointer. In languages such as Pascal, these string pointers are hidden from you. Mesa, on the other hand, makes this string pointer explicit and puts it under program control.

This chapter will show how string pointers differ from standard pointers, and how string use is facilitated by using public interfaces.

7.1 Definition of terms

String A *string* is conceptually a sequence of characters, such as "that". A string is represented in Mesa as a pointer to a record that contains an array of characters and a length

7.2 Discussion

The structure of a **STRING** is very similar to the structure of the **TextSeqBody** in the last chapter. As described in the *Mesa Language Manual* (§6.1), the type **LONG STRING** is:

```
LONG STRING: TYPE = LONG POINTER TO StringBody;
StringBody: TYPE = MACHINE DEPENDENT RECORD [
  length: CARDINAL,
  maxlength: CARDINAL,
  text: PACKED ARRAY[0..0) OF CHARACTER];
```

The **length** field of the string is, by convention, the current length of the string in the **text** array. The **maxlength** field specifies the maximum length of the string. This field is read-only because the size of a string is fixed when it is allocated.

The **text** field is a special form of array, which used to be the primary way for providing dynamic arrays in Mesa, before **SEQUENCE**s were added to the language. It declares an array (as the last field in a record) to have an undetermined length (indices from **[0..0)**). The compiler, however, interprets this field as an array with zero length. This has interesting

effects on string pointer manipulations in assignment and comparisons, as discussed below.

7.2.1 Allocating a STRING

There are four ways to allocate a **STRING**:

- Allocate fixed-sized storage from the local or global frame of a program.
- Assign a string literal to a string variable. String literals are automatically allocated in the local or global frames of your program.
- Use the **NEW** operator to allocate storage from a heap.
- Use procedures provided by the **String** interface (discussed in the *Pilot Programmer's Manual*, §7.3) to allocate storage from a heap.

STRINGs are the only Mesa construct that can be allocated by an explicit request for space from a local or global frame. For example, the following declares a variable **string** and allocates space for up to 256 characters from the same local or global frame as the statement itself:

```
string: LONG STRING ← [256];
```

Sometimes, however, you may want to use known text as a string, for example, to print a message, prompt the user for input, or explain how to use the program Mesa provides string literals for these uses, such as:

```
globalString: LONG STRING ← "Hi There";
localString: LONG STRING ← "Hi There"L;
```

Both of these strings are initialized to point to a record whose **length** and **maxlength** fields are 8 and whose **text** field contains the characters H, i, , T, h, e, r, e. **globalString** is allocated out of the program's global frame; **localString** is allocated from the local frame (denoted by the suffixed **L**.)

When a string literal is inappropriate, you will often allocate the string from a heap (or it will be allocated for you). As a pointer, a **STRING** is well suited for the **NEW** and **FREE** operators. The following example accomplishes what our first example did, except it gets its storage from the heap instead of the local or global frame of the program. It declares a **LONG STRING** and initializes it to **NIL**. When space is needed, it uses the **NEW** operator on the **StringBody** type to allocate a space for 256 characters:

```
string: LONG STRING ← NIL;
...
string ← Heap.systemZone.NEW[StringBody[256]];
```

To deallocate the string, you use the **FREE** operation:

```
Heap.systemZone.FREE[@string];
```

Because strings are very common in Mesa programs, there is a system interface (called **String**) that implements primitive string operations such as allocating, copying, and

comparing strings. The **MakeString** and **FreeString** procedures in this interface work much like **NEW** and **FREE** for allocating and deallocating a string.

String.MakeString takes two parameters: the heap from which the node is to be allocated, and the maximum size of the string:

```
String.MakeString: PROCEDURE[z: UNCOUNTED ZONE, maxlength: CARDINAL];
```

To allocate a string of the same size and from the same heap as the last example, you could code:

```
string: LONG STRING ← NIL;
...
string ←String.MakeString[z: Heap.systemZone, maxlength: 256];
```

String.FreeString takes as its arguments the heap from which the string was allocated and a pointer to the string. It frees the space pointed to by the string and sets the string to **NIL**:

```
String.FreeString[z: Heap.systemZone, s: string ];
```

7.2.2 Caveats in using strings

Besides the usual pointer considerations, there are a few peculiarities related to the structure of strings that you should be aware of. The following examples demonstrate common **STRING** misuse. Try to figure out the effect of each group (and the error) before looking at the explanations.

7.2.2.1 Initializing strings from the current frame

```
string1, string2: LONG STRING ← [256];
```

This is analogous to

```
number: CARDINAL ← 5;
ptrToNumber1, ptrToNumber2: LONG POINTER TO CARDINAL ← @number;
```

It points both strings to the same 256-character space, which is most likely not what was intended. To point each string to its own space of 256 characters, you would code:

```
string1: LONG STRING ← [256];
string2: LONG STRING ← [256];
```

7.2.2.2 Comparing strings

Consider the following attempts to compare **string1** and **string2**:

```
string1: LONG STRING = "Hi There"L;
string2: LONG STRING = "Hi There"L;
  1) IF string1 = string2 THEN ...
  2) IF string1 ↑ = string2 ↑ THEN ...
  3) IF string1.text = string2.text THEN...
```

All three string comparisons are incorrect. The first compares the value of the pointers, and not the objects which these pointers reference. This comparison asks if the two

pointers point to the same object, not if the two objects pointed to are equal. For this example, the result is **FALSE**, even though the two strings contain the same text.

The second comparison seems like it should work: it compares the objects referenced by the two pointers. Unfortunately, when the compiler generates code for the comparison, it treats strings as having text fields with zero length without taking run-time sizes into account. Since the sizes are zero, the statement only compares the **length** and **maxlength** fields of the two strings (equivalent to **string1.length = string2.length AND string1.maxlength = string2.maxlength**). For this example, the result is **TRUE**. However, this comparison does not really compare the two strings.

The final statement fails for the same reason as the second comparison. When the compiler generates the comparison code, it treats the text field as an empty array [0..0). The compiler thinks it is comparing two empty objects. (The result of this is left for you to determine. The value is definitely a constant, but is it **TRUE** or **FALSE**?)

To compare two strings properly, you need to compare each element in their arrays. This is simple to encode, and you may want to try it as a short exercise. However, the **String** interface provides **String.Equal** and **String.Compare** to perform these primitive **STRING** operations; take a look at their descriptions in the **String** section of the *Pilot Programmer's Manual.*

7.2.2.3 Assigning strings

```
string1: LONG STRING ← [256];
string1 ← "Copy this into the string, please"L;
```

This set of statements does not, in fact, copy the string literal into the space allocated from the current frame. The first statement declares the variable **string1** and initializes it to point at a **StringBody** with a 256-character text field. The second statement assigns **string1** to point to a new **StringBody**, one which contains the literal "Copy this into the string, please", making the original 256-character text field leaked storage that can no longer be referenced.

To correctly copy this literal into **string1** you could use either **AppendString** or **Copy** from the **String** interface.

7.2.3 Using the String interface.

The **String** interface provides routines for doing common string operations: comparing, appending, copying, and allocating. A number of the appending and copying routines also involve allocation. You will need to be familiar with these routines to complete the exercises at the end of this chapter.

7.3 Summary

This chapter has not really presented anything new. All string use involves pointers, and you have already learned the intricacies of pointer usage. However, **STRING**s do cause problems, often because programmers are used to strings as arrays of characters. Just remember that in Mesa, the pointer has been put under program control. The structure of Mesa **STRING**s is another potential source of difficulty. Because the **text** field is seen by the

compiler as having zero length, comparisons among **StringBodies** are not as straightforward as among other pointer objects. However, the **String** interface supplies most common string routines, so you will not have to worry about writing them yourself.

7.4 References

Section 6.1 of the *Mesa Language Manual* briefly describes the record structure of a **STRING** and discusses how to declare and use string variables.

Section 7.3 of the *Pilot Programmer's Manual* describes the **String** interface, including many procedures for manipulating **STRING**s.

7.5 Exercises

In this exercise, you will modify a line editor that runs in a tool window. The line editor currently calls several string manipulation procedures defined in the **String** interface. These procedures allocate and deallocate strings from a heap, free strings, copy strings, and replace strings. In addition, the tool implements some more advanced string features such as substring operations. Your assignment is to implement the same procedures through another interface called **String2**. You will write the implementations to this new interface and bind the modules together into a configuration.

You will need the following modules for this assignment:

EditorDefs.mesa
EditorImpl.mesa
EditorTool.mesa
String2.mesa
Editor2.config

Notice that none of the modules currently use **String2**. You should:

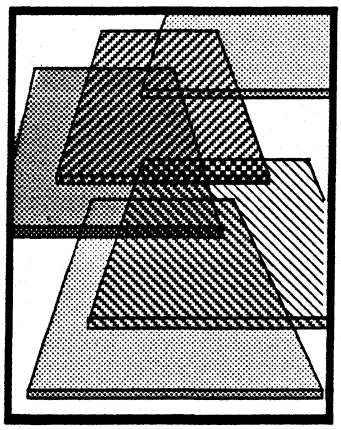
1) Change all **String** references in the module EditorImpl to **String2**.
2) Create an implementation module for **String2**. (Name it String2Impl.mesa.)
3) Move the procedure **InsertString** from the module EditorImpl to String2Impl.mesa.
4) Change all **InsertString** references to **String2.InsertString**.
5) Write implementations for the procedures listed in **String2**.
6) Change the configuration Editor2.config to reflect the new program modules.

All of the procedures in **String2** are taken directly from the Pilot **String** interface. You should take a look at the **String** documentation in the *Pilot Programmer's Manual* to get an idea of what each of these procedures is supposed to do.

This might also be a good time for you to familiarize yourself with a tool called DebugHeap. This tool allows you to check for storage leaks in your programs. To find out how to use this tool, check your *XDE User's Guide*.

Notes:

8



Signals

Signals are a software interrupt facility used when exceptional conditions occur during the execution of a program. Mesa's signal mechanism is more flexible and powerful than the exception handling facilities provided by most other languages or systems.

This chapter provides several examples that illustrate how to suspend program execution to handle an exception, how to provide code to handle the exception, and how to continue program execution afterwards. At the end of the chapter, you will apply your understanding of signals to write a program that both generates and handles signals.

8.1 Definition of terms

Exception — An *exception* is an unusual event that programs must be prepared to handle, such as end-of-file or an invalid input.

Signal — A *signal* is a Mesa language construct used to help handle exceptional conditions encountered during program execution. Signals are like procedures except that the code to be executed for a signal call is determined at run-time.

Error — An *error* is a Mesa language construct similar to a signal, except that program execution can be resumed after a signal, but not after an error. The word "signal" is used to refer to both signals and errors, except where explicitly noted.

Catch Phrase — A *catch phrase* is a Mesa construct that establishes code to catch one or more signals. The catch phrase contains the code to be executed when the exception occurs.

Signaller — The *Signaller* is the program that receives control when a signal is raised, attempts to find an associated catch phrase, and executes the code in the catch phrase.

Call Stack — The *call stack* is a Mesa processor data structure containing a frame for each procedure invocation that has not yet returned. The call stack is ordered by most recent invocation, and is referred to as growing

downward. Therefore, going "up" the call stack means going from the most recently called procedure record toward the oldest.

Raise — To *raise* a signal is to instruct the Signaller to look in each procedure on the call stack until it finds a procedure with a catch phrase for that signal. The Signaller searches up the call stack.

Reject — A catch phrase *rejects* a signal when it is not prepared to handle it (the Signaller continues searching up the call stack for another catch phrase for the same signal). A catch phrase rejects a signal either by explicitly placing a **REJECT** statement in the code or by not specifying how to resolve the signal.

Resume — To *resume* a signal is to tell the Signaller to resume program execution immediately after the statement that raised the signal. As when returning from a procedure call, any values returned by the signal are passed back to the statement that raised the signal. An **ERROR** cannot be resumed.

Continue — To *continue* a signal is to tell the Signaller to resume program execution at the statement following the one to which the catch phrase belongs. Thus, control is resumed in the procedure where the signal was caught, not the procedure that raised the signal.

Retry — To *retry* a signal is to tell the Signaller to re-execute the statement to which the catch phrase belongs.

Goto, Exit, Loop — These are Mesa statements that can be used, in addition to **REJECT**, **RESUME**, **CONTINUE**, and **RETRY** to indicate where execution is to occur after the signal handling mechanism is finished.

Unwind — *Unwind* is a special signal raised by the Signaller to allow procedures about to be deleted from the call stack to clean up their data structures (e.g. deallocate storage and close files). When there is an unconditional branch out of the catch phrase (**GOTO**, **EXIT**, **LOOP**, **CONTINUE**, **RETRY**) the Signaller raises the unwind signal at the point where the original signal was raised.

8.2 Discussion

Generally speaking, there are two methods for detecting an event at which you are not present. You can continuously *poll* an observer or participant of the event, or you can have the observer or participant *notify* you. If the event you are checking for is reasonably predictable and you have time, polling may be convenient. However, if the event is unlikely to occur or happens intermittently, notification may be more convenient. The choice of method always involves a trade-off between the inefficiency of polling when nothing has happened and the inconvenience of being interrupted for notification.

Most computer languages do not implement a notification system for errors or exceptions. Since computers execute so quickly, the inefficiency of polling can often be tolerated, particularly when compared with the expense of providing a notification capability.

However, there are cases, such as device time-out, when notification is an easier, more logical, and more efficient way to communicate the information that an exception has occurred. For example, while you are transferring files from a file server, it is a rare event for the connection to time out, and notification is preferable to polling. Mesa provides the *signal* facility for cases such as this.

Signals also make it easier for someone who is reading a program to see the exceptions that are being handled and to identify the code that handles them. A signal always indicates the occurrence of a rare event. Status polling doesn't have this feature: since it is usually implemented by boolean checking, it is not always obvious which of the two is the rare case.

8.2.1 How signals work

The declaration of a signal is similar to that of a procedure: there may be a parameter list and a returns list. But instead of being initialized to an actual body of code, a signal is initialized by the symbol **CODE**. Here's a sample signal declaration:

```
StringBoundsFault: SIGNAL[s: LONG STRING]
  RETURNS [ns: LONG STRING] = CODE;
```

A signal is raised when a **SIGNAL** (or **ERROR**) statement is executed, as in:

```
SIGNAL StringBoundsFault [string];
```

The body of code to be executed for a signal is determined at run-time (dynamic binding). When a signal is raised, normal execution is suspended and control is passed to the Signaller, which is part of Mesa's run-time support. It is the Signaller's responsibility to find and execute the bodies of code to handle the signal.

These bodies of code are called *catch phrases*. Each catch phrase can have code for one or more signals, in a structure similar to a **SELECT** statement. For example:

```
StringBoundsFault => BEGIN
  ns ← AllocNewString [s: length + 10];
  CopyString [from: s, to: ns];
  DeallocateString [s];
  RESUME [ns];
  END;
String2 => BEGIN...END;
```

A catch phrase can occur in one of two places: explicitly on a procedure call (denoted by "!"), or after the word **ENABLE** in a **BEGIN-END** block. A !-defined catch phrase will catch a signal raised while the called procedure is executing, or while procedures called by that procedure are executing. An **ENABLE**-defined catch phrase does the same thing for every procedure call in the surrounding **BEGIN-END** block, and in addition will catch any signal raised directly in the **BEGIN-END** block. In the code fragment below, **Signal1** would be caught only if it is raised while **Procedure1** is executing. **Signal2**, on the other hand, would be caught if it is raised through **Procedure1**, through another procedure call in the block, or directly, as in the **SIGNAL Signal2** statement.

```
BEGIN
ENABLE Signal2 => BEGIN ... END;
...
Procedure1[...!Signal1 => BEGIN ... END];
SIGNAL Signal2;
...
END;
```

Catch phrases form a dynamic list that is ordered by the call stack, and by **BEGIN-END** blocks within each procedure call. In the example above, the catch phrase for **Signal1** in the call to **Procedure1** is nested below the **ENABLE**-defined catch phrase for **Signal2**. These two catch phrases are followed by any **ENABLE**-defined catch phrases in enclosing **BEGIN-END** blocks and then any catch phrase on the procedure one higher on the call stack, etc. This list of catch phrases is terminated at the root of the call stack, where there is an implicit catch phrase that catches any signal that has not been otherwise dealt with and raises the error **UncaughtSignal**.

When a signal is raised, the Signaller goes up the program's call stack looking in the **BEGIN-END** blocks of each procedure on the stack for a catch phrase that recognizes the signal. When an appropriate catch phrase is found, the Signaller executes a call to it. The parameters (if any) are passed and the catch phrase is entered. As with procedures, the signal's parameters can be referenced inside the body of the catch phrase. (The signal's parameters have precedence over any other symbols of the same name. Within a **StringBoundsFault** catch phrase, for example, **s** and **ns** refer to the signal's parameters.)

After the catch phrase is entered one of three things can happen:

- **Resume** A **RESUME** statement tells the Signaller to conclude processing of this signal and resume execution of the program at the point where the signal was raised. Its syntax is just like **RETURN**, and the signal can return values if it is defined that way. **RESUME** is not legal if the signal is an **ERROR**.
- **Exit** **EXIT**, **CONTINUE**, **RETRY**, **LOOP**, and **GOTO** are the statements used to conclude processing a signal by jumping to a point outside the catch phrase. When a jump occurs, the Signaller raises the special signal **UNWIND** to inform procedures more deeply nested on the call stack that they are about to be deleted. (**UNWIND** is discussed in §8.2.5.)
- **Reject** This tells the Signaller to continue processing this signal and to pass it to the next higher catch phrase. There are three ways that a catch phrase can reject a signal: explicitly (with a **REJECT** statement), implicitly (by not catching the signal), or by first catching the signal, and then "falling off the end" without executing a **RESUME**, **EXIT**, **CONTINUE**, **RETRY**, **LOOP**, or **GOTO**.

8.2.2 Resume

After handling an exception, it's possible to return to the code that raised the signal. This is desirable if the code executed in the catch phrase has eliminated the source of the exception.

For example,

```
Node: TYPE = RECORD[
   index: CARDINAL,
   sequence: SEQUENCE length: CARDINAL OF SeqType];
PtrToNode: TYPE = LONG POINTER TO Node;
seq: PtrToNode;
...

GrowSequence: PROCEDURE [seqNeedsLengthening: PtrToNode]
   RETURNS[lengthenedSeq: Ptr ToNode] = {...};
--If seqNeedsLengthening is NIL then GrowSequence allocates a new sequence and
--returns a pointer, lengthenedSeq, to it. Otherwise, GrowSequence allocates a
--new sequence longer than seqNeedsLengthening.length, copies the data from
-- seqNeedsLengthening↑ to lengthenedSeq↑, frees seqNeedsLengthening↑,
--and returns a pointer, lengthenedSeq, to the new sequence.

InsertNode: PROCEDURE [object: SeqType] =
BEGIN
   IF (seq = NIL) OR (seq.index = seq.length) THEN seq ← GrowSequence[seq];
   seq[seq.index] ← object;
   seq.index ← seq.index + 1;

ProcessNextObject PROCEDURE[object: SeqType];
BEGIN
   IF DuplicateObject[object] THEN TakeAppropriateAction
   ELSE InsertNode[object];
END;
```

If the sequence is full, **InsertNode** calls **GrowSequence[seq]** to lengthen the sequence. It would improve modularity if **InsertNode** knew only how to add data to the sequence, and did not attempt to handle the exception. Instead, when the sequence is full, **InsertNode** would raise a signal to inform a catch phrase on the call stack (presumably one that knows how to grow the sequence) to take care of the problem. Once the sequence has been lengthened, the signal can be RESUMEd, returning control to **InsertNode**, which can then continue to add data to the sequence.

Call Stack

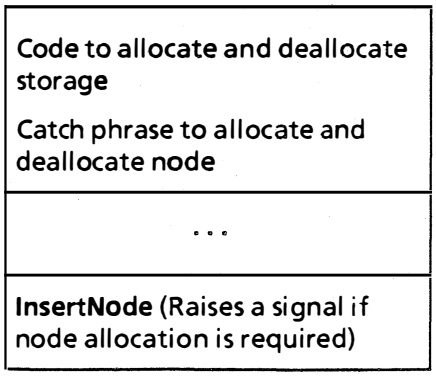


Figure 8.1

Figure 8.1 illustrates this scheme. It shows a box for a procedure that knows how to allocate and deallocate storage, and, lower on the stack, a box for the procedure **InsertNode**, which communicates with the previous procedure by raising a signal when it is necessary to allocate a new node.

Let's look at how to add the appropriate signal-raising and signal-handling code to the above fragment to accomplish this design.

First, we declare the following signal:

```
SequenceBoundsFault: SIGNAL[oldSeq: PtrToNode]
  RETURNS [newSeq: PtrToNode] = CODE;
```

We want to raise this signal when the sequence needs more space. This can occur either when the sequence needs to be initialized for the first time, or when the sequence needs to be extended beyond its present boundaries. We have modified **InsertNode** as follows:

```
InsertNode: PROCEDURE [object: SeqType] =
BEGIN
  IF seq = NIL THEN seq ← SIGNAL SequenceBoundsFault[seq]; --raise signal
  UNTIL seq.index < seq.length DO
    seq ← SIGNAL SequenceBoundsFault[seq];                --raise signal
  ENDLOOP;
  seq[seq.index] ← object;
  seq.index ← seq.index + 1;
END;
```

The first line of code checks to see if the sequence is **NIL**. If it is, it raises **SequenceBoundsFault**, passing **seq** as the sequence to be extended. When the signal is raised, normal program execution is suspended. The Signaller takes over and begins to examine catch phrases on the call stack. An appropriate one is found in the call to **InsertNode** in the revised **ProcessNextObject**:

```
ProcessNextObject PROCEDURE[object: SeqType];
BEGIN
  IF DuplicateObject[object] THEN TakeAppropriateAction
  ELSE InsertNode[object! SequenceBoundsFault = > --catch signal
                    RESUME[GrowSequence[oldSeq]]];
END;
```

The body of the catch phrase is dynamically bound to the signal call and is executed after passing in the parameter, **oldSeq**, of **SequenceBoundsFault**. This catch phrase only contains one line of code, the **RESUME** statement, which calls **GrowSequence[oldSeq]**. **GrowSequence** takes **oldSeq**, allocates a larger one (copying the data from **oldSeq↑**), and returns the new sequence. The signal is then resumed, which passes control back to **InsertNode**, in the statement that raised the signal. At this point, **seq** is assigned the newly allocated sequence returned by the **RESUME**. **InsertNode** now has a freshly allocated sequence into which it can insert data.

The **UNTIL** loop handles the case of no space for new data in the existing sequence **SequenceBoundsFault** works in the same way as just described. (The raising of the signal appears in a loop for robustness, in case the catch phrase does not allocate enough new space to cover **InsertNode**'s needs in a single call. The copying operation described above is

performed each time the signal **SequenceBoundsFault** is raised in the **UNTIL** loop of **InsertNode**.)

Figure 8.2 shows the state of the call stack when a full sequence is encountered. **ProcessNextObject** has called **InsertNode**, which has raised **SequenceBoundsFault[seq]** to signify the need for a larger sequence. This resulted in a run-time system call to the Signaller, which created a call to the catch phrase for **SequenceBoundsFault** (labelled **CatchFrame: ProcessNextObject** in the figure). The catch phrase has then called **GrowSequence**, which will allocate a new sequence and deallocate the old one. When **GrowSequence** returns, the catch phrase will execute a **RESUME**, and return the longer sequence to **InsertNode**.

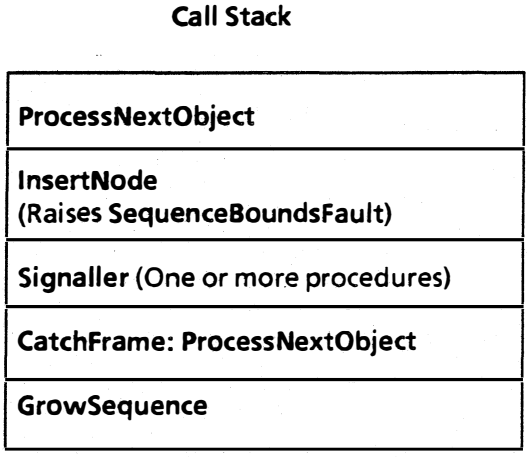


Figure 8.2

Signals do not automatically return after execution of a catch phrase; you must indicate where control is to continue if you do not want the Signaller to continue up the call stack looking for catch phrases. In this case we wanted to return to the point where the signal was raised, so we used **RESUME**. Allowing a signal to "fall off the end" of a catch phrase, is not a **RESUME**, but rather an implicit **REJECT**.

8.2.3 Retry and continue

There are times when an unsuccessful action raises a signal and it is appropriate to repeat the action until it is successful. For instance, if the File Tool is unable to open a connection to a specified service on the first try, you might want it to keep trying until it was successful or until you told it to stop. **RetryExample** provides an example of this. Run the program by typing **RetryExample** in the Executive, followed by the name of a server. (You should move the program to the Tajo volume via Command Central, etc.) The program simulates a failure to open a connection to the specified server. (Notice the message to that effect.) On the second attempt the simulated connection is made.

Take a look at the source listing to see how this retry was accomplished. **RetryExampleImpl** primarily consists of one procedure, **RetryProc**, which gets the server name from the user's input and then tries to open a connection. Inside **OpenConnection**

the signal **TimeOut** can be raised if the connection is not established within a certain time period. This signal is defined in the **SignalsDefs** interface as

```
TimeOut: ERROR;
```

OpenConnection has been rigged for this example to raise the signal **TimeOut** the first time it is called. We catch this signal in the call to **OpenConnection**, print a message to the user to explain the problem and **RETRY**. This causes the program to make the procedure call to **OpenConnection** again. The second call succeeds and we post a message indicating the open connection. Figure 8.3 shows the situation after the signal is caught.

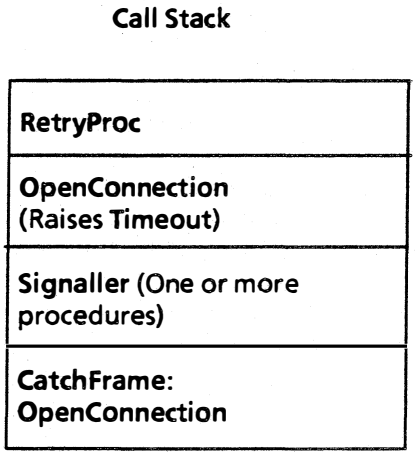


Figure 8.3

When the catch phrase executes the **RETRY**, *there is a jump to the beginning of the statement that contains the catch phrase*, in this case, the call to **OpenConnection**:

```
OpenConnection[server! Timeout => BEGIN ...RETRY END]
```

When an **ENABLE** clause is used to define the catch phrase, the **BEGIN-END** block surrounding the **ENABLE** clause is the "statement that contains the catch phrase." For example, if **RetryProc** had been coded this way:

```
...
BEGIN
ENABLE Timeout => BEGIN ... RETRY END;
...
OpenConnection[server];
END;
```

then the **RETRY** would jump to the beginning of the outermost **BEGIN-END** block.

CONTINUE is similar to **RETRY**, except that the jump is to the statement following the one that contains the catch phrase, or for an **ENABLE** clause, the statement following the **BEGIN-END** block surrounding the clause. **CONTINUE** is used when the catch phrase determines that it is desirable to skip the signal-raising statement rather than retry it.

8.2.4 Exit, loop and goto

The Mesa statements **EXIT**, **LOOP**, and **GOTO** can be used within a catch phrase just as they are used in **BEGIN-END** blocks and loops. These statements are legal within a catch phrase whenever the catch phrase is enclosed within a loop or **BEGIN-END** block in which they would normally be legal.

As an example, consider a program fragment that reads data from a file and inserts it into a linked list in sorted order. (We use the system interface **Stream**, discussed later in the course, to read the file. **Stream** raises the signal **Stream.EndOfStream** at end of file.)

```
DIRECTORY
   Heap USING [Create, Delete],
   MStream USING [Handle, . . .],
   Stream USING [EndOfStream, GetWord, . . .],
. . .;

ExitExample: PROGRAM
   IMPORTS Heap, MStream, Stream, . . . =

BEGIN
--TYPES
Node: TYPE = RECORD[
   data: CARDINAL ← 0,
   nextNode: PtrToNode ← NIL];
PtrToNode: TYPE = LONG POINTER TO Node;
PtrToPtrToNode: TYPE = LONG POINTER TO PtrToNode;

--Variables
z: UNCOUNTED ZONE ← NIL;
headOfList: PtrToNode ← NIL;

--Heap allocation / deallocation procedures
CreateStorageArea: PROCEDURE = BEGIN z ← Heap.Create[initial: 20]; END;

DestroyStorageArea: PROCECURE = {. . .};

MakeNode: PROCEDURE[nextNode: PtrToNode]
   RETURNS[nodePtr: PtrToNode] = {. . .};

FreeOneNode: PROCEDURE[freeThisNode: PtrToPtrToNode]
   RETURNS[nodePtr: PtrToNode] = {. . .};

FreeAllNodes: PROCEDURE =
BEGIN
   tempNodePtr: PtrToNode ← headOfList;
   UNTIL tempNodePtr = NIL DO
      tempNodePtr ← FreeOneNode[@tempNodePtr];
   ENDLOOP;
   headOfList ← NIL;
END;
```

```
--File Management Procedures
OpenDataFile: PROCEDURE [fileName: LONG STRING]
  RETURNS[sh: MStream.Handle] = {...};

CloseDataFile: PROCEDURE[sh: MStream.Handle]
  RETURNS[default: MStream.Handle ← NIL] = {...};

GetNextData: PROCEDURE[sh: MStream.Handle]
  RETURNS[n: CARDINAL] =
  BEGIN
    RETURN[Stream.GetWord[sh]]; --raises Stream.EndOfStream
  END;                          -- at "end of file"

--Linked List Management
ProcessData: PROCEDURE =
  BEGIN
    insertHere: PtrToPtrToNode ← NIL;
    sh: MStream.Handle ← OpenDataFile[MyFile];
    n: CARDINAL ← 0;
    DO
      n ← GetNextData[sh! Stream.EndOfStream = > EXIT];
      insertHere ← SearchLinkedList[n];
      InsertNode[insertHere, n];
    ENDLOOP;
    sh ← CloseDataFile[sh];
  END;

SearchLinkedList: PROCEDURE[n: CARDINAL]
  RETURNS [ insertionPoint: PtrToPtrToNode] = {...};

InsertNode: PROCEDURE[insertionPoint: PtrToPtrToNode, n: CARDINAL] = {...};

...
END.
```

The loop in **ProcessData** gets the next data item from the file, searches the list to see where it belongs and inserts it. Execution of the loop ends at the end of the file. The procedure **Stream.GetWord**, which is called in **GetNextData**, raises the signal **Stream.EndOfStream** when there is no more data to be transferred. The signal is caught in the call to **GetNextData** in **ProcessData**. The loop is then **EXIT**ed and control is transfered to

```
sh ← CloseDataFile[sh];
```

which closes the file before returning.

8.2.5 Unwind

A **GOTO, EXIT, RETRY, LOOP** or **CONTINUE** statement can cause a jump out of a catch phrase into the surrounding code. When a jump of this sort occurs, there may be several procedure calls on the stack below the target of the jump that will be prematurely exited when the jump is accomplished. (The signal was necessarily raised by the procedure on the bottom of the call stack, so neither that procedure nor any of the procedures between it and the procedure with the catch phrase will be completed when the jump is executed.) Since these

procedures may have been in the midst of doing something when the signal was raised, Mesa provides a facility for them to wrap up any unfinished operations.

Before executing the jump, the Signaller raises a special signal called **UNWIND** to tell all catch phrases that had previously rejected the signal that they are about to be removed. **UNWIND** propagates along the same path as the original signal: from the **BEGIN-END** block in which the original signal was raised to the **BEGIN-END** block containing the catch phrase executing the jump. It is the responsibility of each of these blocks to catch **UNWIND** and clean up its operations. The Signaller stops **UNWIND** when it reaches the catch phrase that is making the jump. The jump is then executed and control returns to the program.

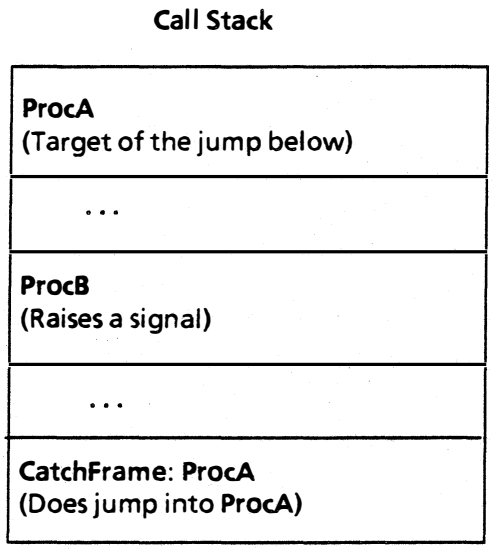


Figure 8.4

In Figure 8.4, **ProcB** has raised a signal which was caught by a catch phrase in **ProcA**. When that catch phrase does a jump, all the procedures below **ProcA** will be removed from the call stack and all **BEGIN-END** blocks within **ProcA** below the target of the jump will be exited. All of the catch phrases more deeply nested than the one executing have (necessarily) rejected the signal, so **UNWIND** propagates through this set of catch phrases. Because **UNWIND** stops after going through the catch phrases that rejected the original signal, it never results in an uncaught signal.

When doing a **GOTO**, **EXIT**, **RETRY**, **LOOP** or **CONTINUE** from a catch phrase, you must be aware that the **UNWIND** signal is going to be raised and that you need to clean up any work in progress in the procedures and **BEGIN-END** blocks lower on the call stack. If you forget, your programs may have space leaks from storage that should have been deallocated, or they may develop strange bugs from things such as files that should have been closed.

As an example, let's modify the previous fragment to allow the user to cancel the operation of inserting data from MyFile into the linked list. If the user hits the **ABORT** key (detected

by the call to the system interface **UserInput**) then the file transfer and insertion operation will be terminated.

```
DIRECTORY
  ...
UserInput USING [UserAbort],
FormSW USING[ProcType, ...],
Put USING[ Line, ...],
  ...;

UnwindExample: PROGRAM
  IMPORTS Heap, MStream, Stream, UserInput, ... =

BEGIN
  ...
--Signal declaration
UserAbort: ERROR = CODE;
  ...
CheckForAbort: FormSW.ProcType =
--Later chapters discuss sending text to a tool message subwindow
  BEGIN
  ENABLE
    UserAbort = > BEGIN GOTO abort; END;
  Put.Line[PtrToSomeToolsDataStructure.msgSW, "Processing File "L];
  ProcessData[];
  Put.Line[PtrToSomeToolsDataStructure.msgSW, " ... done" L];
  EXITS
    abort = >Put.Line[PtrToSomeToolsDataStructure.msgSW, " ...aborted" L];
  END;

ProcessData: PROCEDURE =
  BEGIN
    insertHere: PtrToPtrToNode ← NIL;
    sh: MStream.Handle ← OpenDataFile[MyFile];
    n: CARDINAL ← 0;
    BEGIN
    ENABLE
      UNWIND = >
        BEGIN
          IF sh # NIL THEN sh ← CloseDataFile[sh];
          IF headOfList # NIL THEN FreeAllNodes;
        END;
    DO
      IF UserInput.UserAbort[PtrToInputWindow] THEN ERROR UserAbort;
      --If the user has pressed the abort key raise the global signal UserAbort
      n ← GetNextData[sh! Stream.EndOfStream = > EXIT];
      insertHere ← SearchLinkedList[n];
      InsertNode[insertHere, n];
    ENDLOOP;
    sh ← CloseDataFile[sh];
    END;
  END;
```

```
...
--mainline code
...
CheckForAbort;
...
```

On each pass through the **DO** loop of **ProcessData**, we check to see if the user has hit the **ABORT** key. If so, the error **UserAbort** is raised. (See the Style section for a discussion of when to use **ERROR** and when to use **SIGNAL**.)

We catch the signal and print a message to the user that the action has been aborted. Since this signal has been declared as an **ERROR**, the catch phrase cannot **RESUME**. It must remove **ProcessData** from the stack, but at this point **ProcessData** has an open file and a linked list filled with nodes allocated from a heap. By providing a catch phrase for **UNWIND** in **ProcessData**, we get the chance to deallocate the nodes in the linked list and close the file before the procedure is removed. (See the Style section for a discussion on why the **ENABLE** clause is in an embedded **BEGIN-END** block.)

Note: It is common to recognize an exception condition (either by boolean checking or by catching a signal), and then raise a signal to pass this information on to a higher level procedure. This is often done to hide the lower level's implementation from the higher level's implementation. When debugging an uncaught signal, it is important to remember to check on the call stack for nested signals. For example, the apparent signal may have been raised in a catch phrase for some other signal. The root of the problem may be more apparent from the original signal than the one being debugged.

8.3 Summary

Signals and errors are an alternative to status polling. They are best at handling rare events, since raising a signal requires fewer checks than status polling within a loop, but processing a signal (with the Signaller) takes more time than processing a boolean statement. Using signals also helps the reader of a program to see which exceptions are being handled and to identify the code that handles them.

Though raising a signal is similar to calling a procedure, there are several differences:

- The code for a signal is dynamically bound to the signal at run-time, whereas the code for procedures is specified at compile-time.
- Normal execution halts during the processing of a signal, and the Signaller takes control.
- Execution can proceed at several places after a signal is raised, whereas after a procedure call execution must proceed after the statement that made the call.

The code for processing a signal is contained in a catch phrase. Catch phrases can occur either after an **ENABLE**, or after an ! in a procedure call. Catch phrases after an **ENABLE** can catch signals from any procedure calls nested within the **BEGIN-END** block, but catch phrases in procedure calls can only catch signals nested within that procedure call.

When the Signaller takes control, it does the following:

1. Looks up the call stack for a catch phrase that recognizes the signal, starting with the **BEGIN-END** blocks in the code that raised the signal.
2. Executes any catch phrases found for the signal, branching as indicated in the catch phrase. If no jump is indicated, it continues looking up the call stack.
3. If it can't find a catch phrase in any of the procedures on the call stack, the signal is uncaught, and the debugger is called via the special signal **UncaughtSignal**.

There are several ways to tell the Signaller how to continue execution after a catch phrase. You can use the Mesa statements **GOTO**, **EXIT**, or **LOOP**, with their normal effects. There are also several signal-specific jump statements. Doing a **RESUME** is similar to returning from a procedure call: control returns to the statement that raised the signal. However, you cannot **RESUME** an error. (This is the only difference between signals and errors.) **CONTINUE** causes execution to be transferred to the first statement after the one containing the the catch phrase. **RETRY** retries the statement that contains the catch phrase. (If the catch phrase is in an **ENABLE** clause, then the "containing statement" means the **BEGIN-END** block that contains the **ENABLE**.) **REJECT** tells the Signaller to continue looking up the call stack for another catch phrase that recognizes the signal. If you don't specify any jump statement the catch phrase performs an implicit reject.

GOTO, **EXIT**, **LOOP**, **CONTINUE**, and **RETRY** each cause a jump into the procedure containing the catch phrase. This means that the procedure and **BEGIN-END** blocks below it will be removed from the call stack. The Signaller generates the special signal **UNWIND** to allow catch phrases that have previously rejected the signal to do clean up, such as closing files and deallocating storage.

8.4 Style

8.4.1 Scope

The scope of an **ENABLE** clause places it outside the scope of variables declared in the same **BEGIN-END** block, since the **ENABLE** clause must precede any declarations. (See page 8.5 of the *Mesa Language Manual* for a diagram of clause scopes.) To permit the catch phrase in the **ENABLE** clause to have access to local variables, the **ENABLE** clause must be more deeply nested than the local variables. To accomplish this, declare the **ENABLE** clause and the executable statements within an extra **BEGIN-END** block. The **ENABLE** clause will then know about the variables since they are declared in a surrounding block:

```
BEGIN
  Declarations
  BEGIN
    ENABLE
    Statements
  END
END
```

8.4.2 Errors vs. signals

An **ERROR** is used instead of a signal when a **RESUME** cannot be handled, since it is illegal to **RESUME** an **ERROR**. You don't want a catch phrase to do a **RESUME** if you do not want to return to the procedure that generated the **ERROR**, either because it would be inappropriate, or

because something catastrophic has happened. In the program **UnwindExample**, we used the ERROR **UserAbort**. We made **UserAbort** an ERROR since the user wants the procedure to stop. This is a case where it would be inappropriate to resume execution.

8.4.3 A caution

In the RESUME example in §8.2.2, the catch phrase returned a pointer for use by the RESUMEd procedure. If some intermediate procedure held the value of the old pointer it would not have been informed of the new value, and presumably an error situation would arise when control returned to it. When you code a catch phrase to replace a node out from under a pointer, make sure that any code that used the old node will use the revised pointer.

8.5 Questions

1) In the following code fragment, to which statement will the CONTINUE branch?

```
commands ← 0;
BEGIN
ENABLE
   AlreadyDone => CONTINUE;
   GetToken[token];
   DoCommand[token]; -- where AlreadyDone would get raised
   commands ← commands + 1;
   ResetStatus[];
END
Write["Commands completed."L];
```

In the following code fragments, list the order that the statements labeled <statement n> will be executed.

2)

```
Sig1: SIGNAL = CODE;
x: CARDINAL ← 0;
...
FOR counter: INTEGER IN [1..3] DO
   ENABLE
      Sig1 => RETRY;
   <statement 1>
   IF counter = 2 THEN
      BEGIN
      ENABLE
         BEGIN
         Sig1 => <statement 2>;
         UNWIND => x ← 1;
         END;
      <statement 3>;
      IF x = 0 THEN
         SIGNAL Sig1;
      <statement 4>;
      END;
   <statement 5>
ENDLOOP; ...
```

3)

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO
      BEGIN
      ENABLE
         Sig1 = > LOOP;
      <statement 1>;
      IF counter = 1 THEN
         SIGNAL Sig1;
      <statement 2>;
      END;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

4)

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO
      BEGIN
      ENABLE
         Sig1 = > CONTINUE;
      <statement 1>;
      IF counter = 1 THEN
         SIGNAL Sig1;
      <statement 2>;
      END;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

5)

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO
      BEGIN
      ENABLE
         Sig1 = > EXIT;
      <statement 1>;
      IF counter = 1 THEN
         SIGNAL Sig1;
      <statement 2>;
      END;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

6)

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO
   ENABLE
      Sig1 = > LOOP;
   <statement 1>;
   IF counter = 1 THEN
      SIGNAL Sig1;
   <statement 2>;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

7)

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO
   ENABLE
      Sig1 = > CONTINUE;
   <statement 1>;
   IF counter = 1 THEN
      SIGNAL Sig1;
   <statement 2>;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

8)

```
Sig1: SIGNAL = CODE;
...
Proc1: PROCEDURE =
   BEGIN
      SIGNAL Sig1;
   END;
...
IF TRUE THEN
   BEGIN
   ENABLE
      Sig1 = > RESUME;
   <statement 1>;
   Proc1[!Sig1 = > CONTINUE];
   <statement 2>;
   Proc1;
   <statement 3>;
   END;
<statement 4>;
```

9)

```
Sig1: SIGNAL = CODE;
...
BEGIN
ENABLE
   Sig1 = > RESUME;
<statement 1>;
IF TRUE THEN
   BEGIN
   ENABLE
      Sig1 = > GOTO TheEnd:
   <statement 2>;
   SIGNAL Sig1;
   <statement 3>;
   EXITS
      TheEnd = > <statement 4>;
<statement 5>;
EXITS
   TheEnd = > <statement 6>;
END;
...
```

10) In the following pseudo-Mesa code, what happens when the call **Proc1[0]** is made? (Assume that catch-cases 4 and 7 reject **Sig1**.) Which catch-cases are executed, and in what order?

```
Proc1: PROC [x: CARDINAL] =
   BEGIN -- block A
   ENABLE { -- Catch phrase-1
      Sig1 = > GOTO punt; -- Catch-case-1
      Sig2 = > <Catch-case-2>;
      UNWIND = > <Catch-case-3>};
   Stmt1;
   Stmt2;
      BEGIN -- block B
      ENABLE -- Catch phrase-2
         Sig1 = > <Catch-case-4>;
      Stmt3;
      Stmt4;
      OtherProc[x ! -- Catch phrase-3
         Sig2 = > <Catch-case-5>;
         UNWIND = > <Catch-case-6>];
      END; -- block B, and scope of Catch phrase-2
   Stmt5;
   EXITS
      punt = > Stmt6;
   END; -- Proc1, and scope of Catch phrase-1
...
OtherProc: PROC [x: CARDINAL] = {stillOtherProc[x ! -- Catch phrase-4
      Sig1 = > <Catch-case-7>;
      Sig2 = > <Catch-case-8>;
      UNWIND = > <Catch-case-9>]};
```

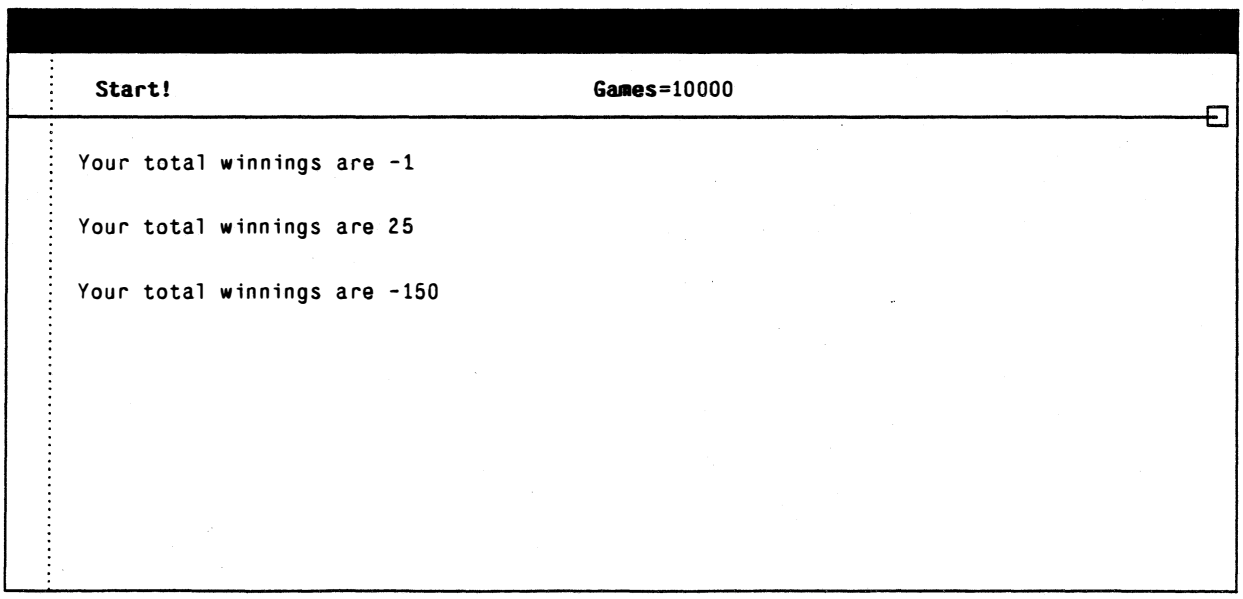
```
StillOtherProc: PROC [x: CARDINAL] = {
  IF x = 0 THEN ERROR Sig1 ELSE ERROR Sig2};
```

11) In the program below, what value does b get?

```
Question3: PROGRAM =
  BEGIN
  Sig: SIGNAL [c1: CARDINAL] RETURNS [c2: CARDINAL] = CODE;
  Proc: PROCEDURE [c1, c2: CARD] RETURNS [BOOLEAN] =
    BEGIN
      ENABLE Sig = > {c2 ← c1; RESUME};
      If c2 # c1 THEN c2 ← SIGNAL Sig[c2];
      RETURN [c1 = c2]
    END;
  c1, c2: CARDINAL;
  b: BOOLEAN;

  --Mainline code
  b ← Proc[1,2];

  END.
```

8.6 Exercise

In this programming assignment, you will alter a program that has been written to play the game of blackjack. The user initially specifies the number of games the program will play with itself. There will only be 2 players in the game: the dealer and the player. When the user clicks **Start!**, the program will play out all of the games; the player's winnings will be output to a file sub-window when all of the games are finished:



In this game of blackjack, the player bets 1 dollar on every hand. If he gets blackjack (a total of 21 in exactly two cards), then he wins 2 dollars. If the dealer gets blackjack, the player loses. If the game continues, the player receives hits (additional cards) according a conservative strategy based on his hand, and the dealer's face card. If he busts (exceeds 21), he loses. Otherwise, the dealer receives hits until his total is a hard 17 (a hand in which an ace is counted as 1 rather than 11) or above. If the dealer busts, the player wins 1 dollar. Finally, if the game has reached this stage, the 2 hands are compared. The players wins 1 dollar if his hand is greater; his winnings remain the same if the hands tie; and he loses if the dealer's hand is greater. There is no double-down, splitting, or insurance in this version of blackjack.

When the user invokes **Start!**, the following procedure in the implementation module is called:

```
PlayBlackJack: PUBLIC PROCEDURE[output: Window.Handle ← NIL, gamesToBePlayed:
  CARDINAL ← 0] =
 --This procedure will play Blackjack as many times as specified in gamesToBePlayed.
 --After the games have been played, results are written out to the window handle
 --output.
BEGIN
 playerTotal: CARDINAL;
 dealerTotal: CARDINAL;
 playerHasAce: BOOLEAN;
 dealerHasAce: BOOLEAN;
 dealerHole: CardType;
 dealerFace: CardType;
 winnings: INTEGER ← 0;

 THROUGH [1..gamesToBePlayed] DO
   IntializeDeckForNewGame;
   [playerTotal,dealerTotal,playerHasAce,dealerHasAce,dealerHole,dealerFace] ←
     Deal[];
   IF playerHasAce AND (playerTotal = 11) THEN
    BEGIN
     winnings ← winnings + 2; --Player has Blackjack
     LOOP;
     END;
   IF dealerHasAce AND (dealerTotal = 11) THEN
    BEGIN
     winnings ← winnings - 1; --Dealer has Blackjack
     LOOP;
     END;
  [playerTotal] ← HitPlayer[playerHasAce, playerTotal, dealerFace];
   IF playerTotal > 21 THEN
     BEGIN
     winnings ← winnings - 1; --Player busted
     LOOP;
     END;
   dealerTotal ← HitDealer[dealerHasAce, dealerTotal];
   IF dealerTotal > 21 THEN
     BEGIN
     winnings ← winnings + 1; --Dealer busted
     LOOP;
     END;
   SELECT playerTotal FROM
     < dealerTotal = > winnings ← winnings - 1;
     > dealerTotal = > winnings ← winnings + 1;
     ENDCASE = >    NULL; -- Push
   ENDLOOP;
 Put.CR[output];
 Put.Text[output,"Your total winnings are "L];
 Put.LongDecimal[output, winnings];
 Put.CR[output];
END;
```

The procedures **Deal**, **HitPlayer**, and **HitDealer** all call the following procedure when they need a card:

```
NewCard: PROCEDURE RETURNS [card: CardType] =
  --This procedure returns the next card in the deck. If at any point, the last card in
  -- the deck is used, the non-used cards in the deck are shuffled, and play continues
  --where it left off
  BEGIN
  IF freeCard = 53 THEN
    [deck, firstCard, freeCard] ← Shuffled[deck, firstCard];
  card ← deck[freeCard];
  freeCard ← freeCard + 1;
  RETURN;
  END;
```

In the procedure **NewCard**, **deck** is an array of 52 records with each record representing one card. Dealing is accomplished by stepping through the deck one card at a time. At any point during a game of blackjack, **firstCard** is an index indicating the first card that was dealt for that hand. **freeCard** is an index indicating the top card on the remaining deck (the next card to be dealt). Thus, when **freeCard** is 53, **deck**, **firstCard**, and **freeCard** are reinitialized by calling the procedure **Shuffled,** which makes sure that the cards on the table are not included in the shuffle. To complete this assignment, you don't have to know how **Shuffled** works, just that it does the right thing when passed the right arguments.

Currently, if the dealer runs out of cards at any point in the game, the cards are in use are shuffled, and the game continues where it left off. So if only 1 card remains in the deck, that card will be dealt, the rest of the deck will be shuffled, and the dealing will continue.

Modify this program (using a signal) so that if the dealer runs out of cards while dealing the initial hand (the first 4 cards), that game is started over with a shuffled full deck of 52 cards. If the dealer runs out of cards while hitting the player, the unused cards in the deck should be shuffled, and the game continued where it had paused (as before). If the dealer runs out of cards while hitting himself, then the dealer loses the game and the next game is started with a shuffled full deck of 52 cards. The file that you will be altering is **BlackjackImpl.mesa**. Other files you will need are **BlackjackDefs.mesa, BlackjackControl. mesa**, and **Blackjack.config**. Once you have the new version of **BlackjackImpl.mesa,** answer the following questions:

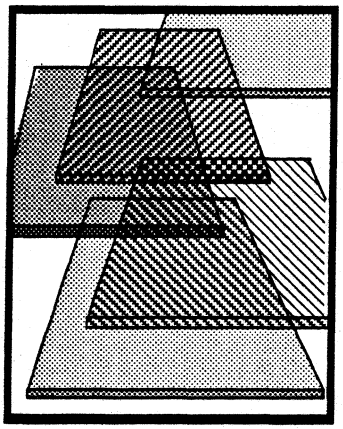
1. Briefly describe how you could have completed the the assignment without using a signal.
2. Signals could have been used to indicate **DealerBlackjack, DealerBusted,...** From an efficiency point of view, why isn't this a good idea?

8.7 References

Chapter 8 of the *Mesa Language Manual* describes the syntax of signals and some reasons for using them.

Section 4 of *Mesa: A Designer's User Perspective* gives some background information on signals.

9



Variant records

Programmers often find it convenient to aggregate information of different types. For example, suppose you want a data base of statistics for individual softball players. For each player, you want to know things like name (**LONG STRING**), position (enumerated **TYPE**), times at bat (**INTEGER**), hits (**INTEGER**), etc. When the information is the same for all players, you can use the Mesa **RECORD** type to group the data for each player. However, some players have additional pieces of information that are relevant only to the position they play. For example, if a player is a pitcher, you want to keep track of the number of walks given up, and the number of strikeouts pitched, in addition to the common information that you keep track of for all players. Or, if a player is an infielder, you might want to know the number of errors committed. In cases where members of a class have information that is relevant only to their subclass, you should use the variant **RECORD** construct.

In this chapter, we discuss how to declare variant **RECORD** types, how to declare, allocate and initialize variant **RECORD** variables, how to use constructors to assign values to variant **RECORD**s, and how to access the fields of variant **RECORD**s.

9.1 Definition of terms

adjective — An *adjective* is an identifier constant from an enumerated **TYPE** used to select one of the alternatives in a variant **RECORD** template.

tag — The *tag* is a field of a variant **RECORD**; tag is used to select one of the alternative "arms" of the variant part by matching one of the adjectives.

discrimination — A *discrimination* statement provides access to the fields in the variant part of a variant **RECORD** variable, based on the value of the tag.

9.2 Discussion

9.2.1 Declaring variant RECORDs

There are basically two parts to declaring a record variable. Step one is to declare a **TYPE** that provides a "template" – that is, the **TYPE** declaration shows all the fields that a variable of that **TYPE** will have Step two is to declare variables of the newly defined **RECORD**

type. Variant **RECORD**s are done the same way. The only difference is that the **TYPE** declaration must show the fields for all possible alternative variants of the **TYPE**.

It is worth taking some time to study the syntax of variant **RECORD**s to make your use of them less error-prone. We declare the **TYPE** as follows:

identifier: TYPE = RecordTC

The syntax for **RecordTC** is shown in Fig. 9.1. Refer to it as you read this discussion.

```
RecordTC            ::= MachineDependent RECORD [VariantFieldList]
MachineDependent    ::= empty | MACHINE DEPENDENT

VariantFieldList    ::= CommonPart identifier : Access VariantPart |
                        VariantPart |
                        NamedFieldList |
                        UnnamedFieldList

CommonPart          ::= empty |
                        NamedFieldList ,

VariantPart         ::= SELECT Tag FROM
                        VariantList
                        ENDCASE

Access              ::= empty |
                        PUBLIC |
                        PRIVATE

Tag                 ::= identifier : Access TagType |
                        COMPUTED TagType |
                        OVERLAID TagType

TagType             ::= TypeSpecification | *
VariantList         ::= Variant | VariantList Variant
Variant             ::= IdList ⇒ [VariantFieldList] , |
                        IdList ⇒ []

    . . .

NamedFieldList      ::= IdList : Access TypeSpecification DefaultOption |
                        NamedFieldList, idList: Access TypeSpecification
                        DefaultOption
```

Figure 9.1 RecordTC Syntax

Obviously, the syntax presents a lot of possibilities for declaring a variant **RECORD** type. The main things to notice are the syntax for the variant field list, for the variant part and

for the tag within the variant part. If a RECORD has a common part and a variant part, there will be an identifier for the variant part and a second identifier for the tag.

Let's look at a simple example. There is a variant RECORD type declared in the program **SoftballDataTool**. (You should retrieve the files `SoftballDataTool.mesa` and `SoftballDataTool.bcd` from the course directory, if you don't already have them on your local disk.) This program is designed to solve the problem of keeping track of information for people on a softball team. Let's look first at the TYPE declarations.

The declaration for **SoftballPlayerData** is a variant RECORD:

```
SoftballPlayerData: TYPE = RECORD[
  name: LONG STRING ← NIL,
  timesAtBat: INTEGER ← 0,
  hits: INTEGER ← 0,
  otherInfo: SELECT position: Position FROM
    outfielder = > [
      bestPosition: OutfieldPosition,
      errors: INTEGER ← 0],
    infielder = > [
      bestPosition: InfieldPosition,
      doublePlays: INTEGER ← 0,
      errors: INTEGER ← 0],
    pitcher = > [strikeouts, walks: INTEGER ← 0],
    catcher = > [],
  ENDCASE];
```

The fields in the common part include **name**, **timesAtBat** and **hits**. We want these three pieces of information about every player. Notice that the syntax requires that you declare all fields of the common part before you declare the variant part. The identifier for the variant part, **otherInfo**, comes just after the fields for the common part.

Each player has a **position**, which is the tag identifier. The TYPE of this field is enumerated: **Position: TYPE = {outfielder, infielder, pitcher, catcher};**. The constants of the enumerated TYPE are used as adjectives in the variant part of the variant RECORD. In our example, the value of **position** for any given player may be either **outfielder**, **infielder**, **pitcher**, or **catcher**. The remaining fields in the RECORD representing any individual player will depend on the value in the tag field. If a player's **position** is **outfielder**, for example, the RECORD representing that player will have two fields (**bestPosition** and **errors**) in addition to the fields in the common part of the RECORD. So, a RECORD representing an **outfielder** has a total of five fields, while the RECORD of an **infielder** has a total of six fields. Notice that a **catcher**'s RECORD only has three fields, because

```
catcher ⇒ []
```

is the way to express the fact that this variant has no additional fields.

This is a relatively simple example. The syntax for RECORD types provides many possibilities, such as bound variant types, implicit tags and computed tags.

9.2.2 Allocation of variant RECORDs

Now that we have declared a variant RECORD type, we can declare variables of that TYPE. You declare and initialize variant RECORD variables in the usual way. For example, notice

```
noPlayer: SoftballPlayerData ← [NIL, 0, 0, catcher[]];
```

in `SoftballDataTool.mesa`. This is the declaration and initialization of a variant RECORD variable. You may be wondering how the Compiler can allocate space for a variable whose size may change during the course of execution of the program; after all, we may assign some other variant to **noPlayer** at some point. The answer is that when a variable is declared to be of TYPE **SoftballPlayerData**, the Compiler allocates enough space for the largest variant.

This program also illustrates allocation from a heap. Instead, the space for the **dataSeq** is dynamically allocated from the system heap by the following statement:

```
IF dataPtr = NIL THEN
    dataPtr ← Heap.systemZone.NEW[Data[numberOfPlayers]];
```

in the procedure **ClientTransition**. Here the run-time system allocates enough space for each member of the sequence to hold the largest possible variant.

9.2.3 Initialization of and assignment to variant RECORD variables

Variant RECORDs are initialized and assigned values like regular RECORDs, except that you must supply appropriate information about the variant part. Here's a helpful way to look at variant record initialization: the variant part is another, embedded record, whose type is determined by the tag, and the syntax for constructing this embedded record is exactly the same as for a regular record.

The RECORD constructor that you use to initialize a variant RECORD variable must specify a value for the tag field, and values for the appropriate fields for that variant. In the above example, the value **catcher** is assigned to the tag field of **noPlayer**. Recall that the **catcher** variant had no additional fields, so no additional values are given in the above constructor. We see other examples of initialization of variant RECORD variables in the procedure **InitDataBase**. For example

```
dataPtr[0] ← [String.CopyToNewString[s: "Ralph"L, z: Heap.systemZone],
  140, 128, pitcher[133, 1]];
```

assigns **"Ralph"** to the **name** field, 140 to the **timesAtBat** field, and 128 to the **hits** field of the RECORD. The **position** field is assigned the value **pitcher**, 133 is assigned to the **strikeouts** field in the variant part, and 1 is assigned to the **walks** field of the variant part of the RECORD.

An alternate way of stating this assignment is:

```
dataPtr[0] ← SoftballPlayerData[
  name: String.copyToNewString[s: "Ralph"L, z: Heap.systemZone],
  timesAtBat: 140,
  hits: 128,
  otherInfo: pitcher[
    strikeOuts: 133,
    walks: 1]];
```

9.2.4 Accessing the fields of a variant RECORD variable

Finally, now that we have declared a variant RECORD type and variant RECORD variables, we are ready to use these variables. A typical situation is when a procedure accepts a

parameter that is of some variant RECORD type, and processes the information contained in the RECORD variable. For example, take a look at the procedure **DisplayData**. This procedure displays the information about each player in the data base in the tool's message subwindow. Notice that it expects a parameter of TYPE **SoftballPlayerData**.

The "discrimination statement" solves the problem of making sure the procedure knows which variant it is dealing with. The common fields of the actual parameter can be accessed normally, but the fields in the variant part can be accessed *only* inside the discrimination statement, which is

```
WITH player: playerData SELECT FROM
  outfielder ⇒ {...};
  infielder ⇒ {...};
  pitcher ⇒ {...};
ENDCASE;
```

Notice how the structure of the discrimination statement mirrors the structure of the TYPE declaration of **SoftballPlayerData**.

Inside the discrimination statement, an "alternate name" is given to the actual parameter by

```
WITH player: playerData SELECT FROM
```

The fields of the variant part of **player** (but not **playerData**) become accessible inside whichever arm is selected, based on the value in the tag of **playerData**. This construct allows the compiler to detect any attempt to access an "incorrect" field within a given arm. For example, if you write

```
Put.Decimal[toolData.msgSW, player.strikeouts];
```

inside the **outfielder** arm of this discrimination statement, the compiler will tell you that "strikeouts is not valid as a field selector. . . ." This prevents you from trying to access a field in an incorrect variant at run time.

Since the discrimination statement relies on the value in the tag field of the RECORD, suppose you just change that value in the tag field. That is, what if you add

```
playerData.position ← pitcher
```

as the first statement in **DisplayData**? Would the discrimination statement always select the **pitcher** arm of the discrimination statement, and try to use the value **strikeouts** for every kind of player? No, Mesa won't allow you to selectively access the tag field of a variant RECORD. In fact, if you try to write the above statement, the Compiler will tell you that "playerData.position cannot be updated. . . ." The only way you can change the variant tag is to assign a new value to the entire variant part using a constructor for that variant part. Variant RECORDs in Mesa are type-safe.

9.3 Summary

This chapter introduced the fundamentals of variant RECORDs. One important feature of Mesa's variant records is that they are type-safe. You can depend on the discrimination statement, in concert with the syntax, to prevent errors associated with accessing the fields in the variant parts of RECORDs.

Several topics related to variant RECORDs that we did not discuss include "bound" variant types, and "implicit" and "computed" tags. The built-in predicate ISTYPE, and the built-in operator NARROW are also available to assist you in your use of variant RECORDs. These features, along with a variation of the discrimination statement that is more efficient in certain cases than the one we looked at, are described in the *Mesa Language Manual.*

9.4 References

Section 6.4 of the *Mesa Language Manual* discusses variant RECORDs, including declaring variant RECORD types and variables, giving values to variant RECORD variables, and accessing the fields of variant RECORDs. This section also discusses several other points regarding particular uses of variant RECORDs that we did not discuss in this chapter.

9.5 Exercises

Modify the **SoftballDataTool** (used as an example in this chapter) to include the following information:

If a player is an infielder, has he been traded ?

If he has been traded:

-- how many times has he been traded ?

-- in what year was he last traded ?

If he has NOT been traded:

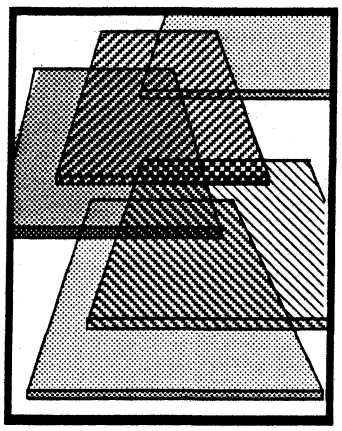
-- how many years has he played for the team ?

-- is he likely to be traded this season ?

You should include this information in a variant section, which is enclosed by the infielder section. Thus, you will create a variant within a variant record. You will have to add this new information for any infielders already existing in the database. Assume that existing infielders have never been traded.

Once you have added the new variant section, a new player will be joining the team. His name is Larry, he is an infielder who plays third base, and he has been traded 3 times, the last time in 1983. You will have to increase the **numberOfPlayers** in order to add him to the database, and print out his statistics along with those of the rest of the team. Obviously, you will also have to change the output routines to dispaly the new information.

10



Concurrency

Mesa provides language support for concurrent execution of multiple processes, as well as *monitors* and *condition variables* to help synchronize such processes.

In this chapter, we discuss how to use the **FORK** and **JOIN** operators to create new processes and later resynchronize them. We also illustrate how to monitor access to a module's global variables, and how to use condition variables to accomplish more complex forms of synchronization. We do not discuss how to monitor data implemented by a multi-module abstraction, or data that is encapsulated in an object rather than in a module; you will have to consult the *Mesa Language Manual* for information on these topics.

10.1 Definition of terms

Asynchronous call — An *asynchronous call* is a procedure call that initiates an operation and then returns control to its caller without waiting for the operation to complete.

Background process — A *background process* is a process that receives machine resources only if higher priority processes are idle or blocked.

Condition variable — A *condition variable* is a Mesa construct by which processes wait for or provide notification of an event. A condition variable is associated with a monitor.

Critical section — A *critical section* is a portion of a program in which only one process may be executing at a time. In Mesa, access to critical sections is arbitrated by monitors.

Hint — A *hint* is information that is usually accurate and is easy for a program to use. A program can detect when a hint is inaccurate and find the truth in some other (usually less efficient) way.

Monitor — A *monitor* module is a Mesa module that controls access to shared data.

Monitor invariant	A *monitor invariant* is a logical assertion about the state of monitored data whenever the monitor is unlocked (i.e., exited). Every monitor has a monitor invariant.
Monitor lock	A *monitor lock* is essentially a hidden data item associated with each monitored record or program that indicates when a process has entered and not yet exited a critical section.
Process	A *process* is effectively a procedure activation that runs concurrently with its caller, allowing asychronous activities.
Synchronous call	A *synchronous call* is a procedure call that returns control only after the operation completes.

10.2 Discussion

Mesa casts the creation of a new process as a special procedure call. You create a new process by FORKing a procedure rather than simply calling it; the new process then runs concurrently with its caller. The new process has a different call stack, with the forked procedure as the root of the activation. Mesa allows any procedure (except an internal procedure of a monitor; see section 10.2.3.1) to be invoked in this way.

10.2.1 JOINing processes

Once you have created concurrent processes, there are various levels of synchronization possible, depending on the role that your forked process is to perform. For example, you might fork a process when you have a long computation to perform, and you would like to allow other processing to take place concurrently. When you create such a process, you later need to synchronize that process with its parent so that it can return the result of the computation. You can accomplish this synchronization with the JOIN operation. JOIN establishes a rendezvous point: the first process to reach the rendezvous is blocked until the other arrives. When both processes have arrived, the forked process returns its results and is then terminated.

To illustrate this, here is an example that iteratively reads a large buffer of data and processes it. A sequential implementation might look like this:

```
Control: PROCEDURE =
  BEGIN
  buffer: LONG POINTER TO Buffer ← zone.NEW[Buffer];
  DO
    ENABLE
      NoMore => EXIT;
    ReadBuffer[buffer];
    ProcessBuffer[buffer];
    ENDLOOP;
  zone.FREE[@buffer];
  END;
```

ReadBuffer collects input data in **buffer**, and then **ProcessBuffer** manipulates the data. The signal **NoMore** is raised when there is no more data, causing the DO loop to terminate.

A problem with this code is that you can not read a buffer of data while processing one, nor process a buffer of data while reading one. Since these operations are distinct, it would be useful (and more efficient) to read the next buffer of data while processing the previous one. This double buffering scheme might look like this:

```
Control: PROCEDURE =
  BEGIN
  Status: TYPE = {normal, end};
  readBuffer: LONG POINTER TO Buffer ← zone.NEW[Buffer];
  processBuffer: LONG POINTER TO Buffer ← zone.NEW[Buffer];
  status: Status ← normal;
  p: PROCESS RETURNS[status: Status];          --declare the process

  status ← ReadBuffer[readBuffer];
  WHILE status = normal DO
    SwapBuffers[readBuffer, processBuffer];
<<points readBuffer to the buffer that has just been processed and points
processBuffer to the buffer that has just been read>>
    p ← FORK ReadBuffer[readBuffer];
    ProcessBuffer[processBuffer];
    status ← JOIN p;
    ENDLOOP;
  zone.FREE[@readBuffer];
  zone.FREE[@processBuffer];
  END;
```

Control now allocates two buffers, one of which can be processed while the other is being filled with the next block of data. **Control** reads in an initial buffer of data and then loops until the reading process returns a state other than normal. During the loop, we swap buffers and then we fork **ReadBuffer**. Thus, we can fill the new buffer while we process the old one. At the end of the loop, we synchronize the two processes with the **JOIN** operator.

Some things to notice from this example:

- **FORK** always returns a value (of type **PROCESS**) and thus a **FORK** cannot stand alone as a statement. Unlike a procedure call, which returns a **RECORD**, you cannot discard the value of the **FORK** by writing an empty extractor. Thus **FORK ReadBuffer[readBuffer]** is assigned to **p**.

- The **JOIN** appears as either a statement or an expression, depending upon whether or not the process being joined returns anything. When the forked procedure has executed a **RETURN** *and* the **JOIN** is executed (in either order),

  the returning process is deleted, and

  the joining process receives the results, and continues execution.

- There is no *intrinsic* rule against multiple activations (calls and/or forks) of the same procedure coexisting at once. Of course, it is possible to write procedures that will work incorrectly if used in this way, but the mechanism itself does not prohibit such use.

10.2.2 Detached processes

Not all processes follow the FORK/JOIN paradigm; there are others whose role is better cast as continuing provision of services, rather than one-time calculation of results. Such processes are called "detached", since they never need to be resynchronized with their caller. If the lifetime of a detached process is bounded at all, its deletion is a private matter, since it involves neither synchronization nor delivery of results.

Pilot provides the facilities for detaching processes. The **Process** interface, documented in section 2.4.1 of the *Pilot Programmer's Manual*, includes operations to check on the state of a process, to set process timeouts, to set process priorities, to abort processes, and to detach processes.

Process.Detach takes a process and detaches it from its creator. If you use this procedure to create a detached process, the **Process** interface will take care of deleting the process when it returns from its root procedure.

Consider a tool with one command, which takes a long time to process. Typically this command runs in the notifier and therefore prevents concurrent user interactions. To avoid this, you can FORK the command as a new detached process:

```
Command: FormSW.ProcType =
  BEGIN
  Process.Detach[FORK RealCommand];
  END;
```

10.2.3 Monitors

FORK/JOIN enables very simple synchronization: you can synchronize two process when a computation has been completed. However, you need a more general mechanism to allow processes to communicate while work is in progress. Specifically, the FORK/JOIN construct does not provide access control (mutual exclusion) to shared data. Thus, we coded the double buffering example to ensure that **ReadBuffer** and **ProcessBuffer** never shared a buffer by executing the pointer swap while only one process existed (and thus there could be no contention to the data).

To enable more sophisticated interaction, Mesa provides an interprocess synchronization mechanism that is a variant of monitors adapted from the work of Hoare, Brinch Hansen, and Dijkstra. The underlying view is that processes share little, but when they do, the interaction reduces to carefully synchronized access to shared data.

10.2.3.1 Mutual exclusion to shared data

A monitor is a module instance. It thus has its own global frame, and its own procedures for accessing this (global) data. Unlike normal PROGRAM module instances, however, a monitor module has an associated monitor lock, which guarantees that only one process at a time can access the data. (The lock can also be associated with the object being shared; see section 9.4.5 of the *Mesa Language Manual*).

Monitor modules are declared much like program or definitions modules; for example:

```
M: MONITOR [arguments] =
  BEGIN
  ...
  END.
```

A call into the monitor implicitly acquires the lock; returning from the monitor releases the lock. When a process attempts to enter a monitor and the lock is already held, it must wait until the current process finishes and releases the lock. The monitor lock thus ensures that only one process at a time can change the data, thereby guaranteeing the integrity of the monitor invariant. (A monitor invariant is an assertion defining what constitutes a "good state" of the data for that particular monitor.)

It is important to realize that the mutual exclusion takes place at the entry and exit points of a monitor. In Mesa, these entry/exit points are encapsulated in procedures called **ENTRY** procedures. The code within an **ENTRY** procedure is a critical section: a call to an **ENTRY** procedure acquires the monitor lock, a return from an **ENTRY** procedure releases the monitor lock. Entry procedures are declared as:

```
P: ENTRY PROCEDURE [arguments] RETURNS [results] = ...
```

The entry procedures will usually comprise the set of public procedures visible to clients of the monitor module. (There are some situations in which this is not the case; see external procedures, below). The usual Mesa default rules for **PUBLIC** and **PRIVATE** procedures apply.

Many monitors will also have *internal* procedures, which are common routines shared among the several entry procedures. These execute with the monitor lock held, and may thus freely access the monitor data as necessary. Internal procedures should be private, since direct calls to them from outside the monitor would bypass the acquisition of the lock. You can only call internal procedures from an entry procedure or another internal procedure. They are declared as follows:

```
Q: INTERNAL PROCEDURE [arguments] RETURNS [results] = ...
```

The attributes **ENTRY** or **INTERNAL** may be specified only on a procedure in a **MONITOR** module (or on an **INLINE** procedure in a definitions module).

Some monitor modules may also wish to have *external* procedures. These are declared as normal non-monitor procedures:

```
R: PROCEDURE [arguments] RETURNS [results] = ...
```

Such procedures are logically outside the monitor, but are declared within the same module for reasons of logical packaging. For example, a public external procedure might do some preliminary processing and then make repeated calls into the monitor proper (via a private entry procedure) before returning to its client. Since it is outside the monitor, an external procedure must *not* reference any monitor data nor call any internal procedures. The compiler checks for calls to internal procedures within external procedures, but does not check for accesses to monitor data.

Generally speaking, a chain of procedure calls involving a monitor module has the form:

```
Client procedure                    -- outside module
        ↓
External procedure(s)               -- inside module but outside monitor
        ↓
Entry procedure                     -- inside monitor
        ↓
Internal procedure(s)               -- inside monitor
```

Any deviation from this pattern is likely to be a mistake. A useful technique to avoid bugs and increase the readability of a monitor module is to structure the source text in the corresponding order:

```
M: MONITOR =
  BEGIN
  <External procedures>
  <Entry procedures>
  <Internal procedures>
  <Initialization (main-body) code>
  END.
```

To illustrate mutual exclusion using monitors, consider the case where many processes may be capable of inspecting, incrementing, and decrementing a counter of active and inactive windows of a multiple instance tool. The operation **Activate** decrements the inactive counter by one and increments the active counter. The **Deactivate** operation does the reverse. To ensure consistent data (i.e. the number of active windows plus the number of inactive windows equals the number of instantiated windows) the increment/decrement to the active and inactive counters must occur atomically. Otherwise, it would be possible for an **Inspect** operation to return a counter that has only been partially updated.

```
KeepCount: MONITOR =
BEGIN
CounterType: TYPE = RECORD[active: INTEGER, inactive: INTEGER];
counter: CounterType ← [0,0];
Activate: ENTRY PROCEDURE =
  BEGIN
  ENABLE UNWIND => NULL;  --see section 10.5.3 for a discussion of this statement
  counter.active ← counter.active + 1;
  counter.inactive ← counter.inactive - 1;
  END;
Deactivate: ENTRY PROCEDURE =
  BEGIN
  ENABLE UNWIND => NULL;  --see section 10.5.3 for a discussion of this statement
  counter.active ← counter.active - 1;
  counter.inactive ← counter.inactive + 1;
  END;
Inspect: ENTRY PROCEDURE RETURNS[counter: CounterType] =
  BEGIN
  ENABLE UNWIND => NULL;  --see section 10.5.3 for a discussion of this statement
  RETURN[counter];
  END;
END.
```

10.2.4 Synchronization with condition variables

In addition to providing mutual exclusion; monitors also allow a sophisticated form of synchronization. For example, a process may only want to execute monitored code if certain conditions hold. If the conditions hold, the process continues as usual. If a condition is not satisfied, however, the process blocks and releases its hold of the monitor lock. A new process can then enter the monitor, eventually make the condition true, and notify the blocked process that it may continue. This kind of synchronization is provided by *condition variables.*

Condition variables are declared as:

```
c: CONDITION;
```

All the fields of a condition variable are private to the process mechanism; you can only access a condition variable via the condition variable operations **WAIT**, **NOTIFY**, and **BROADCAST**.

WAIT condition blocks the current process and releases the monitor lock. Since a **WAIT** always releases the monitor lock while waiting, you must restore the monitor invariant (i.e., return the shared data to a "good state") before waiting.

NOTIFY condition wakes up one process waiting on the condition. (Each condition variable has an associated queue.) If no process is waiting on the condition, the notification is discarded. Unlike **WAIT**, **NOTIFY** does not release the monitor lock. Therefore you can leave the monitored data in an arbitrary state, so long as you restore the invariant before the next time you release the lock (by exiting the entry procedure).

BROADCAST condition wakes up all processes waiting on the condition variable. If no processes are waiting on the condition, the broadcast is discarded. Like **NOTIFY**, the monitor lock is held during this operation.

10.2.4.1 Producer/Consumer problem

Consider the buffering scheme described in the beginning of this chapter. Because of the synchronization limitations imposed by **FORK/JOIN**, we could only use two buffers. A more general solution, however, would allow the two operations to share a buffer pool. This buffer pool would be bounded, as shown in the example on the next page:

```
DIRECTORY
   Heap USING [systemZone],
   MStream USING [Handle, ReadOnly, ReadWrite],
   Process USING [Detach],
   Stream USING [Delete, EndOfStream, GetChar, Handle, PutChar];

CircularBuffer: MONITOR IMPORTS Heap, MStream, Process, Stream =
BEGIN
maxElements: CARDINAL = 10;   --max number of buffers
bufferSize: CARDINAL = 128;
zone: UNCOUNTED ZONE ← Heap.systemZone;

Elmt: TYPE = LONG POINTER TO Buffer;
Buffer: TYPE = RECORD[
   length: CARDINAL ← 0,
   chars: ARRAY [0..bufferSize) OF CHARACTER ← ALL[' ]];
BufferArrayType: TYPE = ARRAY [0..maxElements) OF Elmt ← ALL[NIL];

get, put: CARDINAL [0..maxElements] ← 0;  --which buffer being read/written
bufferArray: BufferArrayType;
notEmpty: CONDITION;
notFull: CONDITION;

-- The consumer gets a buffer from the monitored array of buffers and writes its
-- contents to another file. This process blocks if there are no buffers available.
Consumer: PROCEDURE[outStream: MStream.Handle] =
BEGIN
   DO
      myBuffer: Elmt ← ConsumeBuffer[];
      FOR i: CARDINAL IN [0..myBuffer.length) DO
         ch: CHARACTER ← myBuffer.chars[i];
         IF ch = '& THEN GOTO Exit;
         Stream.PutChar[outStream, ch];
      ENDLOOP;
      zone.FREE[@myBuffer];
   ENDLOOP;
   EXITS Exit = > Stream.Delete[outStream];
END;

-- Producer produces buffers of information obtained from reading a file.
-- It blocks when there is no more room in the monitored array of buffers
Producer: PROCEDURE[inStream: MStream.Handle] =
BEGIN
   DO
      myBuffer: Elmt ← zone.NEW[Buffer];
      FOR i: CARDINAL IN [0..bufferSize) DO
         myBuffer.chars[i] ← Stream.GetChar[inStream! Stream.EndOfStream = >
            {myBuffer.length ← i; GOTO Exit}];
      ENDLOOP;
      ProduceBuffer[myBuffer]; -- put buffer in monitored buffer array
   ENDLOOP;
   EXITS Exit = > Stream.Delete[inStream];
END;
```

```
-- Produce Buffer is called when the Producer needs a buffer.
ProduceBuffer: ENTRY PROCEDURE[element: Elmt] =
BEGIN
   ENABLE UNWIND => NULL;
   WHILE (put + 1) MOD maxElements = get DO WAIT notFull ENDLOOP;
   bufferArray[put] ← element;
   put ← (put + 1) MOD maxElements;
   NOTIFY notEmpty
END;

-- Consume Buffer returns a previously allocated buffer to the available buffer list
ConsumeBuffer: ENTRY PROCEDURE RETURNS[element: Elmt] =
BEGIN
ENABLE UNWIND => NULL;
   WHILE get = put DO WAIT notEmpty ENDLOOP;
   element ← bufferArray[get];
   get ← (get + 1) MOD maxElements;
   NOTIFY notFull;
END;

Init: PROCEDURE[] =
BEGIN
   inStream: MStream.Handle ← MStream.ReadOnly[
      name:"inFile"L,
      release: [NIL,NIL]];
   outStream: MStream.Handle ← MStream.ReadWrite[
      name:"outFile"L,
      type: text,
      release: [NIL,NIL]];
   Process.Detach[FORK Consumer[outStream]];
   Process.Detach[FORK Producer[inStream]];
END;

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

In this example, **bufferArray** is an array that can contain at most **maxElements** (10) elements (buffers). The **bufferArray** starts out empty. The **Producer** (the process reading input) allocates buffers, fills them with information, and adds them to the buffer pool via **ProduceBuffer**. If the buffer pool is full, **ProduceBuffer** waits until there is room. After adding the element to the buffer, **ProduceBuffer** notifies any waiting consumers that another element is available. Similarly, the **Consumer** (the process processing the input) receives its elements by calling **ConsumeBuffer**. If there are no elements in the buffer pool **ConsumeBuffer** waits. Once an element becomes available, **ConsumeBuffer** removes it and notifies any waiting producer processes that the buffer pool is not full.

Notice that a condition variable *c* is always associated with some boolean expression describing a desired state of the monitor data. Each **WAIT** must be embedded in a loop that checks the validity of the corresponding boolean. In Mesa, **NOTIFY** is regarded as a *hint* to a waiting process; it causes a process waiting on the condition variable to resume execution at some convenient time in the future. When the waiting process resumes, it will reacquire the monitor lock. But there is no guarantee that some other process will not enter the monitor before the waiting process. Therefore, the waiting process must

reevaluate the condition before continuing. The general pattern for condition variable code is therefore:

Process waiting for condition:

```
WHILE ~BooleanExpression DO
  WAIT c
  ENDLOOP;
```

Process making condition true:

```
make BooleanExpression TRUE;        -- i.e. as side effect of modifying global data
NOTIFY c;
```

When appropriate, the process mechanism always does a **NOTIFY**, even when there are no processes waiting to be notified. The reason for this is that the built in check (and discard mechanism) is more efficient than any explicit test you could use to avoid the **NOTIFY**. Thus, for example, **ProduceBuffer** always notifies **notEmpty** even if no process is waiting.

This arrangement results in an extra evaluation of the condition after a wait. In return, however, it avoids extra process switches and puts no constraints on when the waiting process must run after a notify. This method is preferable and efficient in Mesa because in general few processes are waiting on the same condition variable at the same time (not many processes will be notified), and context switching is fast (it does not take long for all processes to recheck the state).

10.2.4.2 Single resource manager

Controlling access to a limited shared resource is another common problem that requires interprocess synchronization. The following code segment illustrates a simple storage allocator for objects of uniform size.

```
StorageAllocator: MONITOR =
  BEGIN
  storageAvailable: CONDITION;

  Block: TYPE = RECORD [...];              -- or some other data type
  ListPtr: TYPE = LONG POINTER TO ListElmt;
  ListElmt: TYPE = RECORD[block: Block, next: ListPtr];
  freeList: ListPtr ← NIL;

  Allocate: ENTRY PROC RETURNS [elmt:ListPtr] =
    BEGIN
    ENABLE UNWIND => NULL;
    WHILE freeList = NIL DO WAIT storageAvailable ENDLOOP;
    elmt ← freeList;
    freeList ←elmt.next;
    END;
```

```
  Free: ENTRY PROC [elmt:ListPtr] =
    BEGIN
    ENABLE UNWIND => NULL;
    elmt.next ← freeList;
    freeList ← elmt;
    NOTIFY storageAvailable;
    END;
  END...
```

freeList is the global linked list of available storage. **Allocate** waits until **freeList** is not empty to remove an element. **Free** puts an element back on the **freeList** and notifies any process waiting in **Allocate** that more storage is available.

10.2.4.3 Variable size, single resource manager

If a resource manager manipulates variable sized objects, notification will not work as well. The difficulty is that **NOTIFY** only wakes up one process when more storage is available. Since the size of storage requests vary, available storage may not be enough to meet the needs of the process that is awakened, but it may be enough to satisfy another waiting process.

In this case, you should use **BROADCAST** instead of **NOTIFY**. A **BROADCAST** wakes up all waiting processes. Since the **WAIT** condition statement occurs in a **WHILE** loop, each process will check state before continuing and put itself to sleep if there is not enough storage. Thus, processes that need a smaller amount of storage will be able to continue.

Here is an example of this sort of storage allocator:

```
StorageAllocator: MONITOR =
  BEGIN
  storageAvailable: CONDITION;

  Block: TYPE = RECORD [...];              -- or some other data type
  ListPtr: TYPE = LONG POINTER TO ListElmt;
  ListElmt: TYPE = RECORD[block: Block, next: ListPtr];
  freeList: ListPtr ← NIL;

  Allocate: ENTRY PROC[size: CARDINAL] RETURNS [elmt:ListPtr] =
    BEGIN
    ENABLE UNWIND => NULL;
    UNTIL <storage chunk of size words available> DO WAIT storageAvailable ENDLOOP;
    elmt ← <remove chunk of size words>;
    END;

  Free: ENTRY PROC [elmt:ListPtr, size: CARDINAL] =
    BEGIN
    ENABLE UNWIND => NULL;
    <put back storage of size words>
    ....
    BROADCAST storageAvailable;
    END;
  END...
```

Again, the waiting processes treat notification only as a hint. A process that is awakened does not assume that the condition is true; rather, it assumes that state has changed, and that it should check to see if the condition is true.

10.3 Issues and concerns

This section discusses some issues associated with monitors and processes: how to abort a process, and the relationships between signals and processes, and signals and monitors.

10.3.1 Aborting a process

In addition to **NOTIFY** and **BROADCAST**, you can also resume a waiting process with a timeout or an abort. We discuss Abort in this section; for a discussion on using timeouts see section 9.3.2 of the MLM.

Abort does really not abort the process; it merely raises a signal that indicates to the process that it should clean itself up and return. (If the process is detached, Pilot will destroy it when it returns.) However, the aborted process is free to do arbitrary computations before returning, or indeed to ignore the abort entirely.

You can raise the signal **Abort** by calling **Process.Abort**, with the process to be removed as its argument. The signal is raised the next time the process **WAIT**s on any condition variable that has aborts enabled (the default is to not have aborts enabled; you can call **Process.EnableAborts** to reverse this). If the process is currently waiting it is aborted immediately.

If you want to abort a process that never waits on a condition variable, you must periodically force the process to pause. **Process.Pause** causes a process to wait with aborts enabled for a specified length of time.

10.3.2 Signals and process

Though the creation of a new process via **FORK** is similar to a procedure call, the new process has a different call stack with the forked procedure as the root of the activation. The implication of this is that signals will not cross process activations. Any signal not caught by a new process will not continue to propagate to its parent; instead the debugger will be invoked with an uncaught signal.

10.3.3 Signals and monitors

Signals interact with monitors (entry procedures) in two special ways; in raising a signal and in handling **UNWIND**. Both cases are motivated by the need to release the monitor lock.

When you raise a signal from an entry procedure, the lock is not released. Thus, catch phrases, which can invoke arbitrary operations, may deadlock if they try to reenter the monitor. For errors, you can avoid this with the **RETURN WITH ERROR** construct.

RETURN WITH ERROR *NoSuchObject*;

This statement has the effect of removing the currently executing process from the call chain before issuing the ERROR. Thus, if you execute this statement within an entry procedure, the monitor lock is released before the error is started.

For example, consider the following code segment:

```
Failure: ERROR [kind: CARDINAL] = CODE;

Proc: ENTRY PROCEDURE[...] RETURNS[c1, c2: CHARACTER] =
  BEGIN
  ENABLE UNWIND => ...
  ...
  IF cond1 THEN ERROR Failure[1];
  IF cond2 THEN RETURN WITH ERROR Failure[2];
  ...
  END;
```

Executing ERROR **Failure[1]** raises a signal that propagates until some catch phrase specifies an exit. At that time unwinding begins; the catch phrase for UNWIND in **Proc** is executed and then **Proc**'s frame is destroyed. The lock is held until the unwind occurs.

Executing RETURN WITH ERROR **Failure[2]** releases the monitor lock and destroys the frame of **Proc** before propagation of the signal begins. The catch phrase for UNWIND is not executed in this case. The signal **Failure** is actually raised by the system, after which **Failure** propagates as an ordinary error.

Another important issue regarding signals is the handling of UNWIND. The monitor lock is released as part of the UNWIND, so any entry procedure that may experience an UNWIND must catch it and restore the monitor invariant:

```
Proc: ENTRY PROCEDURE[...] =
  BEGIN
  ENABLE UNWIND => BEGIN <restore invariant> END;
  ...
  END;
```

At the end of the outermost UNWIND catch phrase, the compiler appends code to release the monitor lock before the frame is destroyed.

Even if you don't have to restore the monitor invariant, you should still catch UNWIND in every entry procedure in which it might propagate. The compiler will not generate the code to release the lock unless the UNWIND catch phrase is present. If the monitor is not released during an UNWIND, ensuing calls to the monitor will deadlock.

10.4 Summary

You can spawn new processes from existing ones via the FORK operation. FORK creates a new process, with the invoked procedure as the root of the activation, and returns a process id of type PROCESS to identify the object.

Once instantiated, a new process will either run forever, run for a finite time and return values to (or need to be synchronized with) another process, or run for a finite time without returning results to another process. In the first case, FORKing the new process is sufficient.

In the second case, when a process is expected to return results, you can synchronize its return with the JOIN construct. At this junction, the returning process is deleted and the joining process receives the results and continues its execution.

In the third case, when a process is not JOINed, you must ensure that the process activation is removed. If you use Process.Detach, Pilot will delete the process when it returns to its root procedure.

Concurrent processes create a need for cooperation and communication. Monitors and condition variables provide this cooperation by allowing controlled access and synchronization through shared variables and code.

Mesa monitors are module instances with an associated monitor lock. Mutual exclusion to shared variables (global variables in the monitor module) is ensured by allowing only one process to hold the lock at a time.

In addition to a collection of data and an associated lock, a monitor contains a set of procedures that perform operations on the data. There are three kinds of procedures: entry, internal, and external. External procedures are declared as normal procedures and logically live outside the monitor. Calls to these procedures do not acquire the monitor lock. Entry procedures provide controlled access into the monitor. Calls to an entry procedure either acquire the monitor lock or block until the lock can be acquired. Internal procedures contain the common routines shared among the several entry procedures. These procedures execute with the monitor lock held, and therefore may freely access the monitored data.

Synchronization is accomplished with condition variables and the operations WAIT, NOTIFY, and BROADCAST. A WAIT releases the monitor lock before it blocks. NOTIFY and BROADCAST do not release the lock. Therefore WAIT statements occur in loops, since the condition that was notified may no longer be true when the blocked processes wakes up.

This chapter discussed only the most common form of monitor lock, the global monitor lock. Mesa also supports more specialized forms of monitors, including monitored records and object monitors. Consult chapter 9 of the *Mesa Language Manual* for more details.

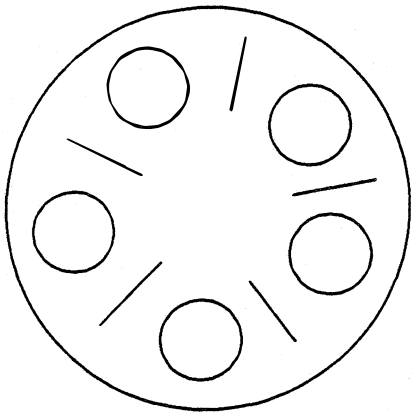
10.5 References

Read Chapter 9 of the *Mesa Language Manual* on Processes and Concurrency.

Read "Experience with Processes and Monitors in Mesa" by Lampson and Redell. (Page 191 of the *Office Systems Technology* book.)

10.6 Exercises

The basic assignment for this chapter is to implement the dining philosophers problem. In this problem, you have 5 philosophers at a dining table. However, there is only one chopstick between each plate, and a philosopher needs 2 chopsticks to eat. At any given time, a philosopher may be thinking, eating, or waiting for the philosopher next to him to put down a chopstick so he can use it.



You can tell a philosopher to try to start eating, or to stop eating and start thinking. When a philosopher is told to start eating, he will look around for some chopsticks and start eating if he can; otherwise he will wait. When a philosopher is told to start thinking, he stops eating (puts down his chopsticks); other waiting philosophers will then see if they can start eating.

```
Philosopher1: {thinking, waiting, eating}
Philosopher2: {thinking, waiting, eating}
Philosopher3: {thinking, waiting, eating}
Philosopher4: {thinking, waiting, eating}
Philosopher5: {thinking, waiting, eating}

Philosopher # 1 is eating.
Philosopher # 2 must wait to eat.
Philosopher # 1 has finished eating.
Philosopher # 2 is eating.
```

There are two levels to this problem, easy and hard. The hard assignment is to solve the dining philosophers problem by yourself. For the easy assignment, we have provided two interfaces and part of the implementation; you only need to write two procedures. If you are adventurous, go start solving the problem now. If you are less adventurous, read the next page to get some help in solving this problem.

For the easier version of this problem, you need to implement the procedures **BeginEating** and **EndEating** from the DP interface:

```
-- DP.mesa

DP: DEFINITIONS =
  BEGIN
  numOfPhils: CARDINAL = 5;

  BeginEating:  PROCEDURE[philosopher: CARDINAL];
  EndEating:    PROCEDURE[philosopher: CARDINAL];
  IsWaiting:    PROCEDURE[philosopher: CARDINAL];
  IsEating:     PROCEDURE[philosopher: CARDINAL];

  END..
```

BeginEating will be called every time a philosopher (a process) thinks it might be able to eat. The philosopher will look around him (look at an array) and see if he can start eating. If he can't, he informs the world that he must wait to eat, calls the procedure **DP.IsWaiting**, and then waits. If he can eat, he informs the world that he is eating, uses his chopsticks (sets some variables in an array) and calls the procedure **DP.IsEating**.

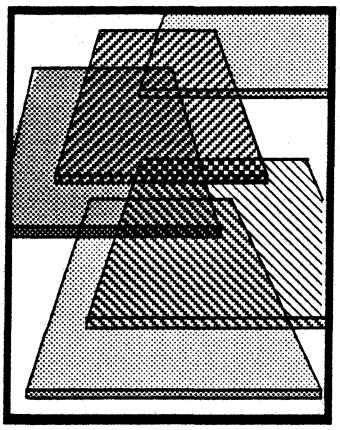
EndEating will be called every time a philosopher has been told to stop eating and start thinking. He should inform the world that he is no longer eating, set down his chopsticks, and tell all waiting philosophers (if any) that they might want to try to start eating. Note that although the tool refers to philosophers 1 through 5, **philosopher** in the above procedures will range from 0 through 4.

To communicate with the world, use the procedures provided in the **ToolDefs** interface:

```
-- ToolDefs.mesa

ToolDefs: DEFINITIONS =
  BEGIN

  PostText: PROCEDURE[string: LONG STRING];     --writes a string of text
  PostLine: PROCEDURE[string: LONG STRING];     --writes a string of text with CR
  PostNumber: PROCEDURE[num: CARDINAL];         --writes a number

  END..
```

You need to write the implementation module DPImpl.mesa, which implements the procedures **BeginEating**, and **EndEating** in the **DP** interface. Use a monitor and a condition variable to synchronize access to the chopsticks by the 5 philosophers (processes). You will need the files DP.mesa, ToolDefs.mesa, DPTool.mesa, and DiningPhilosophers.config, which are on the course directory for this chapter.

11



Introduction to Tajo

This chapter provides a brief introduction to some of the basic ideas behind the design of the Tajo tools environment. The next ten chapters will expand on the ideas contained in this chapter, and illustrate how those ideas are implemented in the design of a new tool.

11.1 Definition of terms

Call back procedure — *A call back procedure* is a procedure that is passed as a parameter to another procedure, and is eventually called from that procedure.

Client — *A client* is a program (as opposed to a person) that uses the services of another program or system.

11.2 Discussion

11.2.1 Windows and subwindows

Most XDE tools use a window for their primary user interface. At the most basic level, a Tajo *window* is just a virtual terminal shell. Tajo provides basic operations on these window shells (such as moving them on the screen), but clients are responsible for adding some functionality to the window. One way to do this is to design your own user interface and implement it using Tajo's low-level routines.

In most cases, however, you do not need to implement your own user interface. Most new tools are built from standard subwindow types, such as file subwindows, message subwindows, form subwindows, and tty subwindows. Each of these subwindow types defines and implements a certain type of user interface. Thus, you can add functionality to a window by specifying that it should consist of some combination of standard subwindows.

This approach has two chief advantages over the approach of writing your own user interface. First, it is much easier for you. Second, it makes life easier for the people who will be using your new tool. Tajo tries to maintain a user interface that is consistent across

all tools, so that the user interface is both easy to learn and easy to use. Thus, you are encouraged to use the standard subwindow types whenever appropriate.

11.2.2 Plug-in modules

The XDE is based on *plug-in* applications. The basic idea is that the XDE is one *self-contained* (but expandable) unit; it does not import any specific procedures that it expects a client to supply, so new applications do not have to be bound in. Rather, any client can call in and announce that it implements some facility. The new application is then "plugged in" to Tajo.

The application is not necessarily run right away, however; instead, once it is loaded, it waits for the user to call it. Instead of a main procedure that calls subroutines, therefore, each tool contains an initialization procedure and individual command execution routines. Loading a tool calls its initialization procedure, which registers the available commands with the system. When the tool is fully initialized, control returns to the system. Thus, a tool simply provides a set of functions and arranges for Tajo to notify it when the user wants it to perform some action.

This style is characterized by the phrase "*Don't call us; we'll call you.*" The motivation for this approach is that the user should be in control, and that he should be able to interact with any tool at any time. Thus, tools are expected to respond to user commands, but should never seize exclusive control of the processor or act independently.

Once a program has been loaded, it remains loaded until the user specifically unloads it or until Tajo is rebooted. This means that software is also reusable: since a program remains loaded, the user can call its command routines at any time.

11.2.3 Notification

This approach means that Tajo is responsible for notifying a tool of user actions (mouse movements, keystrokes, and mouse clicks) that are directed toward its window. There are two processes that cooperate in this notification: one (high priority) that just queues the actions, and another (normal priority) that dequeues each user action and sends it to the appropriate window. (The "appropriate window" is usually the window with the input focus. However, some actions, such as mouse clicks, are sent to the window containing the cursor, which may or may not be the window with the input focus.) Once the action has been directed to a window, it is looked up in a TIP (Terminal Interface Package) to determine which procedure in the associated tool is to be called.

The action lookup table, or TIP table, specifies translations between a sequence of user actions and a sequence of program actions. Each tool window has an associated chain of user-editable TIP tables. A user action is looked up in the first table associated with the designated window. If the event matches the left hand side of a statement in that TIP table, the right hand side (result list) of that statement is executed. If no match is found in that table, the next table in the chain is checked, and so on. If no match is found in any table, the event is discarded.

11.2.4 Virtual memory

Pilot implements a single page-oriented virtual memory shared by all Mesa software, including Pilot itself. All processes run in the same address space, which means that both code and data are shared. (Such sharing is not just permitted, but is encouraged.) To complement the virtual memory, Pilot provides a file system, which serves as the backing store for swapping.

Any page of virtual memory that contains information must have associated with it a page from a file to and from which it can be swapped. Files are associated with virtual memory by mapping a file or portion of a file to virtual memory. The interval of virtual memory used is normally allocated as part of the mapping operation. Each map unit, or mapped interval, is typically subdivided into *swap units*, which consist of one more more pages. Swapping can be done either on demand or under program control. Demand swapping is done by swap units rather than pages; when a page needs to be swapped in, Pilot will bring in that page and any adjoining pages of its swap unit.

Swapping under program control is done via swapping commands, which you can use to specify that you are through with an interval of virtual memory, or that you will be needing one soon.

11.2.5 The File system

The XDE file system is built on top of the Pilot file system. The Pilot file system provides a single, flat (non-hierarchical) directory, and primitives such as file creation and deletion. Pilot expects the XDE file system to super-impose further structure on its files; the emphasis at the Pilot level is on simple, powerful operations for accessing information.

The XDE local file system, called MFile, provides a hierarchical directory structure. Directories are just files containing name translation tables that provide the virtual memory addresses of either files containing data or additional directories. Thus, you can store a file on an arbitrarily deep directory structure.

The XDE file systems provide two ways to access a file. The first way is via streams, which provide sequential access to a file. Thus, a program can use streams to read or write data in a series of bytes, words, or blocks of bytes. The second way is by mapping the file. A client that wants to read from a file will map that file into a virtual memory interval and then use explicit or demand swapping to swap it to real memory. If the file is being updated in place, the client will simply store into the relevant locations of virtual memory. Subsequently, when the interval is unmapped or otherwise swapped out of real memory, the file will reflect the new contents. Doing file access via mapping is a great deal less automated than doing it by streams; if you use mapping to access a file, you need to know what you are doing.

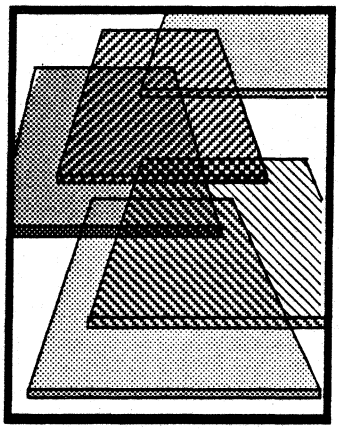
MFile also provides a sophisticated paradigm for sharing files among cooperating processes. Most file systems consider processes to be antagonistic, so they prevent one process from acting on a file if there is a chance that those actions will harm another process using that file. If several processes need to cooperate in the use of a file, they must communicate explicitly among themselves.

MFile, however, provides sharing among processes that do not have to communicate with one another, nor know one another's identities. When a client registers itself with the file system, it can provide a call back procedure. When there is an access conflict, MFile will call that procedure to find out whether the client is willing to release the file. Thus, a client that has provided such a call back procedure will always be notified when another process wants to use that file.

11.3 Summary

All of the ideas presented here are discussed fully later in the course; this chapter serves as an introduction to the rest of the Mesa Course.

12



The Exec interface

In the XDE, a program can interact with users either through its own tool window or through the Executive window. The Executive paradigm is the simpler of the two, and is thus often used for programs that do not require much user interaction or for programs that have a simple syntax. For example, to delete a file it is just as easy to type **delete** *filename* in the Executive as it is to type *filename* into a form field of a tool and invoke **delete!**.

On the other hand, tool windows are an advantage when you need a lot of interaction, when there are a lot of options or parameters to remember and change, or when you use the same commands repeatedly (as with the File Tool). To increase generality, there are many programs and commands, such as file deletion, that can be used either from the Executive or from an individual tool window, depending on the circumstances.

The **Exec** interface provides many routines that make it easier to write a program that runs from the Executive. Using these routines frees you from writing your own user interface and lets you concentrate on writing the code that actually performs the desired command. This chapter discusses how to write code that uses the Executive, and introduces many of the routines in the **Exec** interface. Chapters 17 and 18 discuss how to write programs that have their own window interface.

12.1 Discussion

The first time you type the name of a program into the Executive, you *load* that program and run it once. All programs remain loaded until you specifically unload them or until you reboot the system. Loading a program also *registers* an associated command and name a command procedure to be called when that command is invoked. When you later invoke that registered command, the command procedure is called; the command procedure is then responsible for interpreting the rest of the command line and calling other procedures to get the work done. When the command has been executed, the program returns to its quiescent loaded state until you next invoke the command. Thus, programs are not run in the traditional sense, but are loaded and then wait to be called by the user. We call this style of program execution "*don't call us, we'll call you*".

The "*don't call us, we'll call you*" approach means that Tajo is responsible for *notifying* a program when the user wants it to do something. Tajo checks the Executive window for input and interprets that input by searching its list of known commands. When the

command is found, the Executive invokes the procedure that corresponds to that command. (If the command is not found in the list, the Executive will print an error message to that effect.)

To simplify the code that has to gather input and process it, the **Exec** interface provides many I/O procedures, such as routines for reading and writing to the Executive window. The procedures for writing are first passed to your program from the **Exec** interface and then called by your procedures. Procedures that have been passed as arguments to your program and are then called within your code are referred to as "*call back procedures*".

12.2 Writing programs that use the Executive

To use the Executive, you type *Name Argument* to the Executive. This loads and runs the program *Name.bcd*, registers *Name.~* as a command, supplies the procedure within *Name.bcd* that processes the command, and then calls that procedure. The procedure is then responsible for reading the rest of the input line to obtain its parameter, *Argument*, calculating a result, and displaying that result in the Executive window. Thus, to write programs that use the Executive interface, you need to know how to register a command, how to get information from the input line, and how to output information to the Executive.

12.2.1 Registering a command

Exec.AddCommand is the procedure used to register a command with the Executive. It takes four parameters: **name**, **proc**, **help**, and **unload**:

name is the name of the command to be registered. By convention, a.~ suffix is used to differentiate commands from programs.

proc is the command procedure that the Executive will call when a user invokes the command.

help is a procedure supplied by the client program that prints a message in the Executive window describing how to use the command. The Executive will call this procedure when the user invokes a **Help.~** ***name*** command.

unload is a procedure that the program uses (before it is unloaded from memory) to put itself into a clean state, free allocated memory, and "un-register" its command from the Executive's list. The Executive will call this procedure when the user invokes an **Unload.~** ***name*** command.

An example of **Exec.AddCommand** occurs in the following program (along with several other routines from the **Exec** interface):

```
DIRECTORY
Exec,
Format,
String;

ExecFactorial: PROGRAM
IMPORTS Exec, Format, String =
BEGIN
   Factorial: PROCEDURE[n: CARDINAL] RETURNS[factorial: LONG CARDINAL] =
   BEGIN
      inputTooBig: CARDINAL = 0;
      SELECT n FROM
         = 0    = > RETURN[1];
        IN [1..12]  = > RETURN[n*Factorial[n-1]];
      ENDCASE       = > RETURN[inputTooBig];
   END;

--Print out help information to the Executive
--Called when user types Help Fact to the Executive
HelpProc: Exec.ExecProc =
BEGIN
   OutputProc: Format.StringProc ← Exec.OutputProc[h];
   OutputProc["Fact calculates the factorial of a CARDINAL"L];
   Format.CR[OutputProc];
   OutputProc["number less than or equal to 12"L];
   Format.CR[OutputProc];
END;

-- Read the argument from the command line with Exec.GetToken
--call Factorial to calculate the factorial
--print the results with a call back procedure named OutputProc.
Fact: Exec.ExecProc =
BEGIN
   answerString: LONG STRING ← [16];
   number: CARDINAL;
   answer:LONG CARDINAL;
   token,switches: LONG STRING ← NIL;
   OutputProc: Format.StringProc ← Exec.OutputProc[h];

   [token,switches] ← Exec.GetToken[h]; -- get arguments
   IF token = NIL THEN RETURN;
   number ← String.StringToNumber[s:token, radix:10];
   answer ← Factorial[number];
   IF answer = inputTooBig THEN OutputProc["Input Too Big"L]
   ELSE
   BEGIN --put numeric answer in string format for output
      String.AppendLongNumber[s:answerString, n: answer, radix:10];
      OutputProc["The Factorial of "L];
      OutputProc[token];  -- print answer to Executive
      OutputProc[" is "L];
      OutputProc[answerString];
   END;
```

```
    token ← Exec.FreeTokenString[s:token]; --Free strings
    switches ← Exec.FreeTokenString[s:switches];
END;
--register Fact with the Executive
Exec.AddCommand[name:"Fact.~"L,.proc: Fact, help:HelpProc];
END.
```

This call to **AddCommand** does not specify an **unload** argument, since **ExecFactorial** is coded to leave itself in a clean state each time a **Fact** command is processed. A "clean state" is one in which the program has released all storage allocated during run time: in this case **Fact** does not have any global data and it calls **FreeTokenString** to release its strings. Thus the default **Exec.DefaultUnloadProc** is sufficient, so the **unload** parameter may be omitted:

```
Exec.AddCommand[name: "Fact.~"L, proc:Fact, help: HelpProc];
```

If your procedure does not leave itself clean you would add a fourth argument to the call to **Exec.AddCommand** that specified a procedure to call when unloading. The procedure for unloading has two tasks: first, it must free any global data and second, it should make a call to **Exec.RemoveCommand** to remove the instantiation of the command. **Exec.RemoveCommand** takes a handle to the Executive (**h**) and a **LONG STRING** (**"Fact.~"**) as arguments. For example:

```
UnloadProc: Exec.ExecProc =
BEGIN
--Free global variables (if any)
Exec.RemoveCommand[h,"Fact.~"L]; --remove Fact.~ from the Executive
END;
```

The procedure **HelpProc** is invoked when a user types **Help Fact** in the Executive; in this example, two lines of text are printed as an aid to the user. **Fact** is called whenever the user types **Fact** to the Executive. It is up to the writer of **Fact** to read and process the arguments that follow the command.

The procedures **proc**, **help** and **unload** are usually provided by the writer of the command being registered, although default values such as **Exec.DefaultUnloadProc** are available. These procedures are of type

```
Exec.ExecProc: TYPE = PROCEDURE [
  h: Exec.Handle] RETURNS [outcome: Exec.Outcome ← normal];
```

Each procedure is called with argument **h**, which is a handle to the Executive that called it. **h** identifies the particular instantiation of the Executive window in which the command was invoked. **proc**, **help** and **unload** use this variable when passing information to and from an Executive window.

The variable **outcome** that is returned by each of these procedures, indicates the status of the returning procedures, and can be **normal**, **warning**, **error**, or **abort**. If everything went smoothly, the outcome **normal** should be returned. If problems were encountered that the Executive should know about, the other outcomes can be used.

12.2.2 Getting information from the command line

The Executive provides support for interpreting the user's input as a series of pairs in the form `token/switches token2/switches2` ..., where each pair is separated from each other pair by white space (one or more spaces or tabs), `token` and `switches` are separated by /, and the input line is terminated by a return.

Each token or switches is a set of characters terminated by a delimiter, which can be white space, a slash or a return. If a token or switches has quotes around it, like "this is one token and/or switch" then you can include white space in it. The first token on a command line is interpreted by the Executive as the command name, as in this example:

```
Command Argument1/Switches1 Argument2/Switches2
```

`Tokens` and `Switches` are simply arguments for the command typed into the Executive. Tokens can appear without any switches and switches can appear without a token. The argument-tokens are usually used as arguments (or parameters) by the command-token, and the switches-tokens are used to tell the command how to interpret the argument. For example, many commands use /f to mean that the argument is a file name, as in User.cm/f.

The procedure that parses the input line into tokens and switches is defined as:

```
Exec.GetToken: PROCEDURE [h: Exec.Handle] RETURNS [token, switches: LONG STRING];
```

where **h** is the input parameter available within each **Exec.ExecProc**. If either **token** or **switches** is empty, **GetToken** will return **NIL** for that variable. When the entire input line has been parsed, **GetToken** will continually return **[NIL, NIL]**. When a non-**NIL** token or switches is returned, it is stored in memory allocated by **Exec** and it is the client's responsibility to free this space using **Exec.FreeTokenString.**

The procedure **Fact** in **ExecFactorial** is responsible for getting the user's input from the command line. Fact uses the **Exec.GetToken** procedure to read in both a token and a switch. In this example only the token is analyzed. The token is first converted into a **CARDINAL** by using the **String.StringToNumber** procedure in the string interface. The **CARDINAL** is then passed to **Fact** and the factorial is calculated. In the example program, the line

```
[token,switches] ← Exec.GetToken[h]; -- get arguments
```

reads the number into **token** and nothing (**NIL**) into **switches**. When the program is finished with the strings **token** and **switches**, it makes calls to

```
token ← Exec.FreeTokenString[s:token]; --Free strings
switches ← Exec.FreeTokenString[s:switches];
```

which deallocates the strings and sets the pointers to **NIL**.

The first token on a command line is not available through **GetToken**, since it has already been digested by the Executive as the command (or object file) name to be run. Thus, if the user types:

```
Command/g Arg1/a Arg2
```

successive calls to **GetToken** would return:

1) [NIL, "g"]

2) ["Arg1", "a"]

3) ["Arg2", NIL]

4) [NIL, NIL]

and then continue to return [NIL, NIL] if called again.

12.2.3 Displaying in the Executive window

After **Fact** returns, the results need to be displayed in the Executive window. The **Exec** interface supplies the procedure **Exec.OutputProc**, which outputs strings to an Executive window. To get this procedure, call:

```
OutputProc: Format.StringProc ← Exec.OutputProc[h];
```

This assigns a procedure body to **OutputProc**. (Note that **h** is defined within **Fact** because **Fact** has been defined to be an **Exec.ExecProc**.) To display text, call this procedure, passing the desired string, as in:

```
OutputProc["this will print in the Executive "L];
```

In the factorial example, **OutputProc** is used to print out help information in the **HelpProc** procedure and to print the answer to the factorial in the Executive window. Because the procedure returned by **Exec.OutputProc** is of type **Format.StringProc** it can be used as an argument to other procedures in the **Format** interface. For example, **HelpProc** calls **Format.CR** to output a carriage return to the Executive:

```
Format.CR[OutputProc];
```

12.2.4 Other useful procedures

This section discusses some of the other procedures in the **Exec** interface that you might find useful.

You may want to allow your programs to *Abort* when the user hits the **STOP** key. This is useful, for example, when a program is stuck in an infinite loop or when you type in a command that you realize should not be executed. An example of this procedure is

```
DO
  IF Exec.CheckForAbort[h] THEN EXIT;
END
```

The procedure **CheckForAbort** is of type **CheckAbortProc**. It takes a handle to the Executive and returns a **BOOLEAN** indicating whether or not you wish to abort the command. In this example you will **EXIT** the loop if the **STOP** key is struck.

The **Exec.GetChar** and **Exec.PutChar** offer single character I/O. **Exec.GetChar** is a procedure of type **Exec.GetCharProc**; it takes a handle to the Executive and returns the next character

on the command line. The first character that **Getchar** reads is the one immediately after the command name; after the last character has been read **Getchar** returns an **Ascii.Nul** character. The **Exec** interface also provides **Exec.EndOfCommandLine**, which tests for the end of the command line. **Exec.EndOfCommandLine** takes a handle to the Executive and returns a **BOOLEAN** indicating whether the end of the line has been reached. A short program fragment illustrates the use of these commands:

```
ReadAndPrint: Exec.ExecProc =
BEGIN
  letter: CHARACTER;
  letter ← Exec.GetChar[h];
  WHILE ~Exec.EndOfCommandLine[h] DO
    Exec.PutChar[h,letter];
    letter ← Exec.GetChar[h];
  ENDLOOP;
END;
```

12.3 Summary

Using the Executive either as your primary user interface or as a supplement to a tool window is a good idea when your commands are fairly simple. The **Exec** interface makes it easy for you to present the user with a uniform interface. This interface uses a call-back scheme: when a program is run, it registers a command-name and command-procedure with the Executive, which is called-back by the Executive when a user invokes the command.

To use the user interface supplied by the **Exec** interface you need to know how to register a command with the Executive, how to get input from the command line, and how to print information in the Executive window:

- To register a command with the Executive, use **Exec.AddCommand**. You need to supply the name of the command, the procedure to be called when the command is invoked, a help procedure which describes how to use the command, and, if necessary, an unload procedure.

- To get input from the command line, use **Exec.GetToken**. This call returns a record containing a pair of strings. The first element in the pair is the token, the second element is the switches. **Exec.FreeTokenString** is used to free the space allocated to the pair of strings and should be called before exiting the command procedure.

- To output to the Executive, use the procedure returned by a call to **Exec.OutputProc**. The procedure returned by this call is a **Format.StringProc** and can be used to display strings in the Executive window.

12.4 Style

To avoid confusion, use a command name that is the same as the name of the program that implements that command. This makes it easier to execute the command the first time (when the command hasn't yet been registered), since the Executive will load and start a

program with the given name if it cannot find an appropriate command. Once the program is started, and the command is registered, the Executive will execute the command.

12.5 References

Chapter 4 of the *XDE User's Guide* discusses the Executive from the user's point of view.

Chapter 5 of the *Mesa Programmer's Manual* discusses the **Exec** interface.

Section 7.2 of the *Pilot Programmer's Manual* discusses the **Format** interface. Procedures from this interface are used to format data of various types into strings for output.

12.6 Exercise

In this exercise you will write a simple line editor that has four commands: Insert, Delete, Find, and Replace. Each of these commands will operate on a character string that you enter into the Executive at the start of the program. The syntax for each command is explained below

Insert [/b] {key_1/$info_1$}..[key_N/$info_N$}

Insert inserts the character string *info* either before or after *key*. The default should be after if no switch is specified but the user may specify before with the "/b" switch immediately after the **Insert** command. You should write Insert so it takes any number of *info*/*key* pairs and so it prints the modified string after each insertion. If no key is specified the info should be appended to the end of the string.

Delete {key_1}..[key_N]

Delete removes the item name *key* and replaces it with the value in info. If *key* is not found a message should be printed to that effect and if it is found the resulting string should be printed.

Find {key_1}..[key_N]

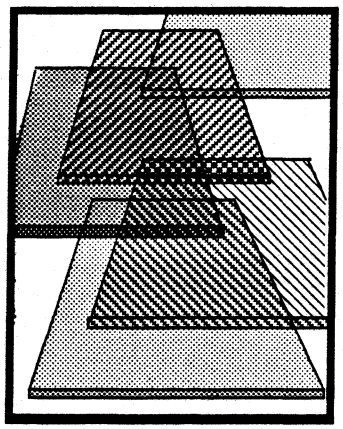
Find calculates the index of the item named *key* within the string and prints this value to the Executive. If the *key* is not found a message should be printed.

Replace {old_1/new_1}.. [old_N/new_N]

Replace locates the item named *key* and replaces it with the value in *info*. (The string may need to be lengthened or shortened depending on the lenth of *key* and *info*). If the *key* is not found an error message should be printed.

Your program should add each of these commands to the Executive and provide help procedures for each command. An Unload procedure should also be provided in order to clean up when the editor is no longer needed. A template is provided to help you write code for this program; this template is stored on >Chapter 12>ExecEditorTemplate.mesa.

13



MFile

The Mesa file system supports concurrent, cooperating client processes, and coordinates accesses to files. The file system facilitates inter-process cooperation by asking clients to provide procedures that the file system can call to ask the clients to give up a file or to tell other clients that a file is available. The view that its clients are cooperative allows it to support a more sophisticated sharing of files among independent processes than is possible in traditional environments.

The Mesa file system is novel in that it supports cooperation between clients that do not know about one another. This approach allow design strategies that would be impractical under other circumstances. You can be secure in the knowledge that if an optional use of a file is interfering with other work, the offending program will be informed that another process wishes to access that file

This chapter discusses how to acquire a file from the file system, how to return it when you are done, and how to provide call-back procedures to enable this cooperative sharing.

13.1 Definition of terms

Mesa file system — The Mesa file system is a virtual tree-structured file system that allows notification of events and provides a protocol for releasing files to facilitate interprocess cooperation.

13.2 Discussion

If you want to create or write into a file, you must first acquire the file. In the process of acquiring the file you may find that it is unavailable (another process is using it), in which case you may wish to be told when the file does become available for use. In addition, when you have acquired a file, other processes may wish to use it also, thus creating a need for cooperation. This chapter discusses how the XDE file system provides file access to cooperating processes with as few conflicts as possible.

13.2.1 Gaining access to files

To perform operations on a file, you must first obtain a **LONG POINTER** (handle) to the file You use the file handle to access both the contents of a file and the properties of a file.

these properties are defined when the file is first created and are stored in the **MFile.Object**. The **MFile** interface provides several procedures for obtaining this handle, the most general of which is **MFile.Acquire**.

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

MFile.Object TYPE; --opaque type prevents direct access by programmer

MFile.Acquire: PROCEDURE[
  name: LONG STRING,
  access: MFile.Access,
  release: MFile.ReleaseData,
  mightWrite: BOOLEAN ←FALSE,
  initialLength: MFile.InitialLength ←MFile.dontCare,
  type: MFile.Type ←unknown] RETURNS[MFile.Handle];
```

name is the name of the file that you want to acquire. The **access** parameter specifies the desired access; this can be any of the following values:

```
MFile.Access: TYPE = MACHINE DEPENDENT {anchor(0), readOnly, readWrite, writeOnly,
log, delete, rename, null};
```

anchor lets you determine whether a file exists and, if so, to read its properties, but does not allow you to read or delete the file.

readOnly allows you to read the file but not to write it.

readWrite permits you to read and write the file and to change the length of the file.

writeOnly access lets you write the file and change the length but does not allow reads.

log truncates the file to length zero each time a client accesses the file, thus allowing new data to be appended to the file. For example, the compiler uses a log file that it rewrites each time you run the compiler.

delete permits you to delete the file.

rename lets you change the name/file binding of a file either by renaming a file or swapping two files

null access is provided only for initialization and must not be used for accessing the file.

The **mightWrite** parameter is only significant if **access** is **anchor** or **readOnly**. If **mightWrite** is **TRUE**, **MFile.Acquire** will not return a handle on a file in a write-protected directory. We will discuss the **release** parameter in detail in section 13.2.3.

In addition to specifying the different access methods, you must also specify the type of file you will be reading (or writing). The file type is a value of **MFile.Type**:

```
MFile.Type: TYPE = MACHINE DEPENDENT {unknown(0), text, binary, directory, null(255)};
```

unknown indicates that the file does not have one of the other file system types.

text indicates that the file contains characters.

binary files may contain arbitrary data.

directory files are special files containing part of the directory structure of a file system.

null is only used when copying file handles with the same access.

If **access** is **anchor**, **readOnly**, **delete**, or **rename**, the file must already exist or the error **MFile.Error[noSuchFile]** will be reised. If access is **readWrite**, **writeOnly**, or **log**, the file system first checks to see if the file already exists. If it does, **Acquire** ensures that the number of bytes in the file is at least as large as **initialLength**, although it does not set the logical length of the file. If the file does not exist, the file system will create a new file of size **initialLength** and type **type**.

```
fileName: LONG STRING ← "MyFile.txt"L;
releaseData: MFile.ReleaseData ← [NIL, NIL];
fileLength: MFile.InitialLength ← 1000;
fileHandle: MFile.Handle ← MFile.Acquire[name: fileName, access: readOnly, release:
  releaseData, initialLength: fileLength, type: text];
-- Perform operations on the file
```

13.2.1.1 Other methods of acquiring files

In addition to **Acquire**, you can access files with **MFile.ReadWrite**, **MFile.ReadOnly**, and **MFile.WriteOnly**. These procedures are really shorthand methods of calling **MFile.Acquire**, so you can use them to reduce the number of parameters **MFile.Acquire** requires and to increase the readability of your code. The definitions for these procedures are shown below:

```
MFile.ReadWrite: PROCEDURE [name: LONG STRING,
  release: MFile.ReleaseData,
  type: MFile.type,
  initialLength: MFile.InitialLength ← MFile.dontCare] RETURNS[MFile.Handle];

MFile.ReadOnly: PROCEDURE [name: LONG STRING,
  release: MFile.ReleaseData,
  mightWrite: BOOLEAN ← FALSE] RETURNS[MFile.Handle];

MFile.WriteOnly: PROCEDURE [name: LONG STRING,
  release: MFile.ReleaseData,
  type: MFile.Type,
  initialLength: MFile.InitialLength ← MFile.dontCare] RETURNS[MFile.Handle];
```

13.2.2 Copying file handles

Occasionally you will acquire a file with one type of access and later want to change to a different access. One way to do this is to release your file handle and then perform another **Acquire**, but this is relatively slow. A better method is to simply copy the file handle with **MFile.CopyFileHandle**, which acts as an accelerator for **Acquire** and avoids looking up the file in the directory again.

```
MFile.CopyFileHandle: PROCEDURE [file: MFile.Handle,
  release: MFile.ReleaseData,
  access: MFile.Access ← null] RETURNS [MFile.Handle];
```

CopyFileHandle provides a way around some of the access controls provided by the file system; if the access requested for the copy is no stronger than that of the original access

(e.g. **readWrite** is stronger than **writeOnly**, which is stronger than **readOnly**, which is stronger than **anchor**), the file system will make the copy, even though it would not permit another client to gain that access to the file. (There is a list of access conflicts in the MFile chapter of the *Mesa Programmers Manual*.) Thus, if a client has a handle with **readWrite** access, it can get a copy with **readOnly** access or **readWrite** access, although another client requesting a handle with either of these accesses would be refused. You must be careful, however, since the file system assumes that a client requesting such conflicting handles is responsible for the potential chaos that might result if they are misused.

13.2.3 Releasing files

Once you have acquired a file, you will want to perform operations on it; the operations you can perform are discussed in the next two chapters (MSegment and Streams). Regardless of the operations you perform, however, you must release the file when you are finished or other processes will not be able to access the file. You release a file by calling **MFile.Release**, which returns control of the file to the file system.

```
fileHandle: MFile.Handle ← MFile.Acquire[...]; ...

MFile.Release: PROCEDURE[file: MFile.Handle]; ...

fileHandle ← NIL; -- always set the Handle to NIL
```

You should always set the file handle to **NIL** after you return from your call to **MFile.Release** since the file system does not do this automatically and if your program later attempts to access the file an error will result.

13.2.3.1 PleaseReleaseProcs

In section 13.2,1 we mentioned the **release** parameter used when acquiring a file handle. The **release** parameter is used to determine whether your process will allow another process to access the file. When you do not want any other process to have access you can simply set the release parameter to [**NIL**, **NIL**]: if you wish to allow multiple access to the file you must provide an **MFile.PleaseReleaseProc**. This release parameter is defined as:

```
MFile.ReleaseData: TYPE = RECORD[
  proc: MFile.PleaseReleaseProc ← NIL,
  clientInstanceData: LONG POINTER ← NIL];

MFile.PleaseReleaseProc: TYPE = PROCEDURE[
  file: MFile.Handle,
  instanceData: LONG POINTER] RETURNS[MFile.ReleaseChoice];

MFile.ReleaseChoice: TYPE = {later, no, goAhead, allowRename};
```

later means that the file will be released soon, so the file system should delay the **Acquire** until this occurs.

no tells the file system to reject any request to access the file.

goAhead specifies that you are willing to release the file, and can guarantee that you will not access the file afterward. We discuss methods of releasing files in later examples.

allowRename specifies that you do not object to having the file renamed.

Thus, when you set your release parameter to [NIL, NIL]; this says to the file system, "I want exclusive access to this file". When you want to allow other processes to be able to access the file, you must write an MFile.PleaseReleaseProc.

The Mesa file system facilitates inter-process cooperation by asking clients to provide procedures (PleaseReleaseProc) that the file system can call to ask the client to give up a file. We call such procedures *call-back* procedures because the file system will call back to the client (at the file system's discretion) via these procedures. For example, if a process wants to write a file that another process is currently reading, the file system will call the reading process's pleaseReleaseProc and ask it to relinquish the file. When the PleaseReleaseProc is called, it returns a ReleaseChoice to the Mesa file system. This ReleaseChoice determines whether the file system will grant access to the requesting process. If the PleaseReleaseProc relinquishes the file, the process that gives up the file must not access it again since there is no implicit release of the file; in other words, the client process should behave as if the last statement of its PleaseReleaseProc were MFile.Release.

In addition to the PleaseReleaseProc, the ReleaseData contains a clientInstanceData pointer, which points to client-specific information that the pleaseReleaseProc can access when it is called. For example, clientInstanceData might be a pointer to a tool message subwindow; thus, when another process attempts to access the file, the pleaseReleaseProc can display a message in the subwindow.

You should generally allow others processes to have a file if you are reading it or are not performing any critical tasks with the file. Some cases where you would not want to release the file include: writing a file, deleting a file, and changing the files properties. Below is a sample program which both reads and writes to a file; when it is writing to the file, the process will refuse to release the file, but it will release the file when it is reading.

```
DIRECTORY --Definitions module
MFile;
CopyFileDefs: DEFINITIONS =
BEGIN
   MyMonitor: PROGRAM;
   AcquireRead: PROCEDURE[fileName: LONG STRING] RETURNS[handle: MFile.Handle];
   AcquireWrite: PROCEDURE[fileName: LONG STRING] RETURNS[handle: MFile.Handle];
   SomeOneWantsFile: PROCEDURE[handle: MFile.Handle] RETURNS[file: MFile.Handle];
   MyReleaseProc: MFile.PleaseReleaseProc;
END.
```

```
DIRECTORY -- Monitor implementation
CopyFileDefs,
MFile;

MyMonitor: MONITOR
IMPORTS MFile
EXPORTS CopyFileDefs =
BEGIN
   pleaseFree, reading: BOOLEAN ← FALSE;
   -- When either of the acquire procedures is called, acquire the file handle
   -- from the file system and return it to the client
   AcquireRead: PUBLIC ENTRY PROCEDURE[fileName: LONG STRING]
   RETURNS[handle: MFile.Handle] =
   BEGIN
      reading ← TRUE;
      pleaseFree ← FALSE;
      RETURN[MFile.ReadOnly[name: fileName, release: [proc: MyReleaseProc,]]];
   END;

   AcquireWrite: PUBLIC ENTRY PROCEDURE[fileName: LONG STRING]
   RETURNS[handle: MFile.Handle] =
   BEGIN
      reading ← FALSE;
      pleaseFree ← FALSE;
      RETURN[MFile.WriteOnly[name: fileName, release: [proc: MyReleaseProc],
            type: text]];
   END;

   -- If another process wants to use the file and the file
   -- is in read mode, then release the file
   SomeOneWantsFile: PUBLIC ENTRY PROCEDURE[handle: MFile.Handle]
      RETURNS[file: MFile.Handle] =
   BEGIN
      IF pleaseFree THEN {pleaseFree ← FALSE; MFile.Release[handle]; RETURN[NIL]};
      RETURN[handle]; -- no one wants or is granted access
   END;

   --PleaseReleaseProc for file only release file if in read mode
   MyReleaseProc: PUBLIC ENTRY MFile.PleaseReleaseProc =
   BEGIN
      IF reading THEN {pleaseFree ← TRUE; RETURN[later]}
      ELSE RETURN[no];
   END;
END.
```

```
 --Main program CopyFileExample
DIRECTORY
CopyFileDefs,
MFile;
CopyFileExample: PROGRAM
IMPORTS CopyFileDefs, MFile =
BEGIN
   ENABLE
   MFile.Error = > GOTO exit;

   finishedUsingFile: BOOLEAN ← FALSE;
   fileName: LONG STRING ← "MyFile.txt"L;
                                                --get file from file system
   myFileHandle: MFile.Handle    ← CopyFileDefs.AcquireRead[fileName];
   UNTIL finishedUsingFile DO
      -- perform some tasks using the file, if another process requests file
      -- then exit
      IF CopyFileDefs.SomeOneWantsFile[myFileHandle] = NIL THEN GOTO exit;
   ENDLOOP;
   MFile.Release[myFileHandle]; --Release file so it may be reacquired
   myFileHandle ← CopyFileDefs.AcquireWrite[fileName]; --with write access
   finishedUsingFile ← FALSE;
   UNTIL finishedUsingFile DO
      -- perform some more tasks using the file but don't release
   ENDLOOP;
MFile.Release[myFileHandle]; -- release the file and set the handle to NIL
myFileHandle ← NIL;
   EXITS
   exit = > NULL;
END.
```

Note that the sample program is divided into two modules. The first part (**MyMonitor**) is a **MONITOR** that communicates with the file system; the second part (**CopyFileExample**) makes calls to **MyMonitor ENTRY** procedures to acquire a file and to determine whether any other process wants access to the file.

CopyFileExample first acquires the file with **readOnly** access and performs operations on the file, periodically checking to see if others have requested the file. It performs these checks by calling the **MONITOR ENTRY SomeOneWantsFile**. **SomeOneWantsFile** checks to see if another process has made a request to access the file (**pleaseFree**); if so, it will release the file and return a value indicating that the file should not be used any longer. Thus, **CopyFileExample** is willing to give up the file when it is in this first loop. However, when **CopyFileExample** reacquires the file with **writeOnly** access, it is not willing to release the file and therefore does not make any calls to **SomeOneWantsFile**.

13.2.4 Notification

Sometimes a client process may wish to be notified when a file becomes available for a particular access; for example, a file window may wish to know whenever there is a new version of the file it contains. In other words, whenever the file is changed the window would like to redisplay the new version. Another common use of notification is when a process relinquishes a file via its **PleaseReleaseProc** and would like to regain access as soon as the file becomes available again. Client processes ask to be notified when a file is

available by calling MFile.AddNotifyProc with the file name and access of interest, and the NotifyProc to be called when the file becomes available.

```
MFile.AddNotifyProc: PROCEDURE [
  proc: MFile.NotifyProc,
  filter: MFile.Filter,
  clientInstanceData: LONG POINTER];

MFile.Filter: TYPE = RECORD [
  name: LONG STRING ← NIL,
  type: MFile.Type ← unknown,
  access: MFile.Access];

MFile.NotifyProc: TYPE = PROCEDURE [
  name: LONG STRING,
  file: Handle,
  clientInstanceData: LONG POINTER] RETURNS [removeNotifyProc: BOOLEAN ← FALSE];
```

The notification is performed by a special process in the file system, which maintains a list of files that are eligible for notification. This process checks the name and the Filter information to determine if the desired file is available; if so, the NotifyProc is called. Because of the nature of a multiprocess environment there are several important items to note when using NotifyProcs:

When the NotifyProc is called by the file system, the file argument to the NotifyProc contains a Handle on the file if the file exists; otherwise, file is NIL. The NotifyProc should always check this handle before using it.

file belongs to the system, so if you want a handle on the file you must call MFile.CopyFileHandle on the handle passed in, and must explicitly specify the access required. In addition, when using MFile.CopyFileHandle the file system does not guarantee that you will be able to obtain the desired access, thus notification can only be viewed as a strong hint.

There is no guarantee about the order of notification or about how quickly notification will take place. This is due to the fact that notification takes place in another process.

To avoid deadlock with the file system, the NotifyProc should not call AddNotifyProc or RemoveNotifyProc; you should use the BOOLEAN result of the NotifyProc to remove itself from the notify list.

13.2.4.1 Removing notification

Under some circumstances you may want to remove the NotifyProc from consideration yourself; this is done with a call to MFile.RemoveNotifyProc.

```
MFile.RemoveNotifyProc: PROCEDURE [
  proc: MFile.NotifyProc,
  filter: MFile.Filter,
  clientInstanceData: LONG POINTER];
```

As mentioned above you should not make a call to MFile.RemoveNotifyProc from within your NotifyProc. A sample program illustrating NotifyProcs is shown below.

```
DIRECTORY
MFile;
FileDefs: DEFINITIONS =
BEGIN
   NotifyProcExample: PROGRAM;
   Acquire: PROCEDURE[fileName: LONG STRING] RETURNS[handle: MFile.Handle];
   NotifyProc: MFile.NotifyProc;
   CanIKeepTheFile: PROCEDURE[handle: MFile.Handle] RETURNS[yes: BOOLEAN←FALSE];
   ReleaseProc: MFile.PleaseReleaseProc;
END.

DIRECTORY
FileDefs,
MFile;
NotifyProcExample: MONITOR
IMPORTS MFile
EXPORTS FileDefs =
BEGIN
   fileName: LONG STRING ← NIL;
   pleaseRelease: BOOLEAN ← FALSE;
   ready: CONDITION; --wait on ready when the file is in use by others

   -- Acquire acquires the file handle. If the file handle is not available,we
   -- add a notify proc and wait for it to become available

   Acquire: PUBLIC ENTRY PROCEDURE[fileName: LONG STRING]
      RETURNS[handle: MFile.Handle ← NIL] =
   BEGIN
      handle← MFile.Acquire[
         name: fileName,
         access: readOnly,
         release: [ReleaseProc] !MFile.Error = > {
            MFile.AddNotifyProc[
               proc: NotifyProc,
               filter: [name: fileName, access: readOnly],
               clientInstanceData: NIL];    -- if the file is in use add the notify proc
            WAIT ready;                     -- and wait here until the file is again available
            RETRY;  -- make another attempt to acquire the file
         }];

      pleaseRelease ← FALSE;     -- no one has requested file
      RETURN[handle]; -- file  was acquired so return handle
   END;

   NotifyProc: PUBLIC ENTRY MFile.NotifyProc =
   BEGIN
      removeNotifyProc ← TRUE;
      NOTIFY ready; -- allow acquire entry to wake up
   END;
```

```
    -- if another process requested the file then release it and wait
    CanIKeepFile: PUBLIC ENTRY PROCEDURE [handle: MFile.Handle]
      RETURNS [yes: BOOLEAN←FALSE] =
    BEGIN
      IF pleaseRelease THEN {
        MFile.Release[handle]; -- some other process wants file so release it
        MFile.AddNotifyProc[   -- ask to be notified when it becomes available
          proc: MyNotifyProc,
          filter: [name: fileName, access: readOnly],
          clientInstanceData: NIL];
        WAIT ready;            -- wait here until file is available again
        pleaseRelease ← FALSE;
        RETURN[FALSE]};        -- inform caller that the file must be reacquired
      RETURN[TRUE];            -- file was available so keep using it
    END;

    --PleaseReleaseProc for file always releases the file
    ReleaseProc: PUBLIC ENTRY MFile.PleaseReleaseProc =
    BEGIN
      pleaseRelease ← TRUE; --file will be released when CanIKeepFile is called
      RETURN[later];
    END;
  END.

  DIRECTORY
  FileDefs,
  MFile;
  CopyFileExample: PROGRAM
  IMPORTS FileDefs =
  BEGIN
    fileName: LONG STRING ← "SomeFile.txt"L;
    fileHandle: MFile.Handle ← fileHandle ← FileDefs.Acquire[fileName];
    DO
      -- perform operations on the file
      -- if anyone else wants the file then give it up and sleep
      -- until it becomes available again
      IF ~ FileDefs.CanIKeepFile[fileHandle] THEN fileHandle ← FileDefs.Acquire[fileName];
    ENDLOOP;
  END.
```

CopyFileExample first attempts to acquire a file by calling the MONITORed **Acquire** procedure. If the file is not available, **Acquire** requests that the file system notify it when the file does become available; the **Acquire** ENTRY then WAITs on the condition variable **ready.** When the process that is currently using the file returns it, **ready** is NOTIFYed by the notify procedure **NotifyProc. Acquire** then RETRYs to access the file; this may or may not be successful due to the fact that other processes may attempt to acquire the file also; thus, **Acquire** may again be forced to WAIT.

When the file is finally acquired, **CopyFileExample** performs operations on the file while checking to see if another process wants to use the file. It checks by calling **CanIKeepFile**, which returns a BOOLEAN indicating whether it has given up the file at the request of another process. **CanIKeepFile** tests a global MONITORed variable **pleaseRelease** (that is set to TRUE by the **pleaseReleaseProc** if another process attempts to gain access to the file.) If **pleaseRelease** is TRUE, **CanIKeepFile** releases the file (doing the release notifies the other process that the file is available), and then waits until the file is available again before it

returns. If no other process has requested the file, **CanIKeepFile** immediately returns **FALSE** and **CopyFileExample** can continue to process the file.

Note that this code is not re-entrant; that is, there can only be one copy of some of the global variables. This is not ideal programming style; you will learn how to avoid the global variables later in the course.

13.2.5 Manipulating Files

You may want to perform operations on files without accessing their contents. For example, you may want to copy one file to another, delete a file, rename a file, or create a new subdirectory. In this section we will briefly discuss the procedures that the Mesa file system provides for doing these tasks.

The **MFile.Copy** procedure copies a file into another file. The client must have **readOnly** or **readWrite** access to **file**, and it must be able to open the file **newName** for **writeOnly**.

```
MFile.Copy: PROCEDURE [file: MFile.Handle, newName: LONG STRING];
```

You can easily delete a file by calling **MFile.Delete** with the file handle, or rename a file with **MFile.Rename**.

```
MFile.Delete: PROCEDURE[file: MFile.Handle];
MFile.Rename: PROCEDURE[file: MFile.Handle, newName: LONG STRING];
```

MFile.CreateDirectory creates a directory if one does not already exist. All the intermediate subdirectories on the path will be created as necessary, (*e.g.*, if **dir** is *<Tajo >Defs >Source* and subdirectory *Defs* does not exist, it will be created as well as subdirectory *Source*.

```
MFile.CreateDirectory: PROCEDURE[dir: LONG STRING];
```

13.2.5.1 Obtaining information about files

Mesa files all have properties (dates, length, protection, type, etc.) and you can retrieve many of these by calling the appropriate **MFile** procedure. Some of the more useful of these procedures are **GetCreateDate**, **GetLength**, and **GetProtection**, each of which takes an MFile.handle as a parameter and returns information about the file.

```
MFile.GetCreateDate: PROCEDURE [file: MFile.Handle] RETURNS [create: Time.Packed];
MFile.GetLength: PROCEDURE [file: MFile.Handle] RETURNS [MFile.ByteCount];
MFile.GetProtection: PROCEDURE [file: MFile.Handle]
  RETURNS [deleteProtected, writeProtected, readProtected: BOOLEAN];
```

13.3 Summary

In a multiprocess environment processes frequently need access to common files. Traditionally, file systems grant access to files on a competitive basis. The Mesa file system, however, assumes that processes are cooperative, thus permitting the maximum amount of sharing. Cooperation is facilitated via call-back procedures that allow a process to release a file and then to reacquire it without any direct interprocess communication.

When a process acquires a file from the file system, it can include a **PleaseReleaseProc**. The file system will call this **PleaseReleaseProc** when another process requests the file with a conflicting access. The **PleaseReleaseProc** may refuse to release the file (**no**), delay the request to use the file until it has been released (**later**), allow the file to be renamed (**allowRename**), or relinquish the file (**goAhead**).

A process may also ask to be notified when a specified file becomes available by calling **MFile.AddNotifyProc**, which registers a call-back procedure, (a **NotifyProc**), with the file system. When the file becomes available, the **NotifyProc** is called and can perform operations in attempt to access the file.

The **MFile** interface provides procedures to manipulate files, to change their properties, and to create or also destroy directories. In contrast to many environments the XDE allows programs to manipulate files and their properties

13.4 References

Chapter 47 of the *Mesa Programmer's Manual* defines the MFile interface and provides additional information on **PleaseReleaseProcs**.

Chapter 4 of the *Pilot Programmer's Manual* provides information about the file system.

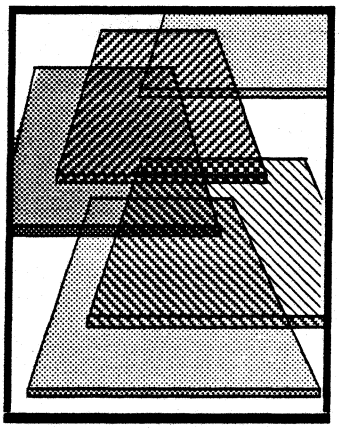
13.5 Exercise

The exercise for this chapter is to write a monitor to implement a multi-window tool. The tool has two commands, **AcquireSW1** and **AcquireSW2**, that attempt to acquire a file and display it in the corresponding subwindow. The problem is that both commands attempt to acquire the same file with conflicting access; thus only one will be successful. (Note: you need to try to acquire the files with **readWrite** access; if you request **read** access, there will be no access conflict.) The command that is unsuccessful must wait for the file to become available and again attempt to access and to display the file. The file will become available when you invoke the **Release** command, which releases the file.

You are to implement a monitor that has **ENTRY**s for **Acquire** and **Release**, which are called via detached processes when you invoke the corresponding commands. The arguments to the processes contain pointers to the tool subwindows so you can perform output to the tool windows. In addition, we provide a procedure that prints a file in a file subwindow, since printing the file requires attaching a stream to the file.

You need to write a **NotifyProc** (**NotifyProc**) and possibly a **PleaseReleaseProc** (**ReleaseProc**). The **NotifyProc** should ensure that the process waiting for the file gets notified that the file is available. The **PleaseReleaseProc** should always return **no** since you want requesting processes to wait until the file is explicitly released via the tool.

The tool window module is stored as WindowTool.mesa and the definitions file, which describes the procedures that you will write, is on FileDefs.mesa. A working implementation module for the monitor is stored in MyMonitor.mesa and the configuration file is ForkConfig.config.



MSegment

The **MSegment** interface supports mapping files to spaces in virtual memory called *segments*. You can use such segments as input/output buffers to improve the performance of programs that need to do a lot of file operations (reading and writing.) You can also use segments to impose a structure on a data file and manipulate that structure much like a simple Mesa variable. This chapter discusses how to create segments, how to perform I/O with the segments and how to delete the segments when you are finished.

14.1 Definition of terms

Dirty Page

A *dirty page* is a page in a segment that has information different from the information in the backing file. To bring the backing file up to date, dirty pages must be written to disk.

Page

A *page* is a piece of storage One page is **Environment.wordsPerPage** (256) words or **Environment.bytesPerPage** (512) bytes.

Real Memory

Real memory is that amount of fast-operating, random-access storage directly addressable by a processor. Currently, most Dandelion processors have between 1000 and 3000 pages (512K bytes and 1.5 Mbytes) of real memory.

Segment

A *segment* is a sequence of virtual pages. Once created, a segment occupies a fixed position in virtual memory; its starting and ending addresses never change. (However, you can change the properties of a segment with a procedure called **MSegment.Reset**; this procedure may have to change the location of the segment in order to change the properties..)

Segment-File Mapping

Segment-File Mapping is the process of associating a segment with a sequence of contiguous pages of a backing file. Since virtual memory is implemented by combining the resources of real memory with those of the file system, any portion of virtual memory that contains information must be associated with a file that acts as a backing store. For proper operation of segment-file mapping, the size of a segment in pages should be less than or equal to the size of the backing file in pages.

Swapping — *Swapping* is the act of bringing pages of data from virtual memory to real memory, or vice versa.

Virtual Memory — *Virtual memory* is a scheme by which the amount of storage available to the processor appears, from a programmer's point of view, to be larger than the size of the real memory. Data stored in virtual memory actually exists in files. In order for the processor to operate on data, the data must be mapped into real memory via an address translation scheme.

14.2 Discussion

A segment is a sequence of contiguous pages in virtual memory. Since the operating system cannot guarantee that an entire segment will be in real memory at any given time, it needs a backing file as a place to store the data from those parts of the segment that are not currently in real memory. Thus, every segment must be backed by a file (or some portion of a file.) If you try to write data into a portion of a segment that is not correctly mapped to a file, an address fault will occur.

Below are two examples of segment-file mappings. The first example illustrates the default case: the segment and backing file have the same size. The second example shows a backing file that is "larger" than its associated segment. It is possible to create a segment that is larger than its backing file but writing data into regions of the segment that have no backing will result in an address fault.

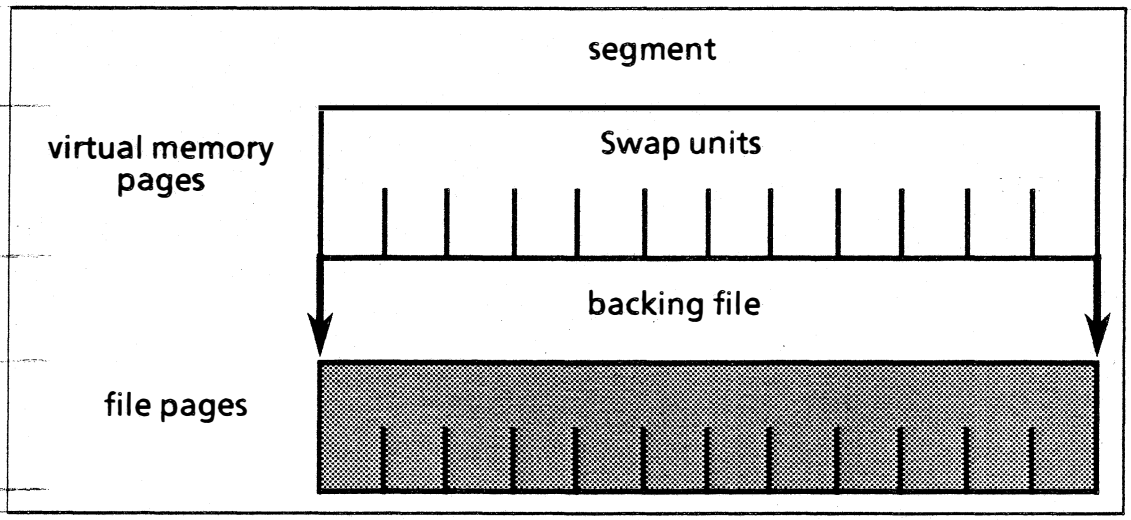


Figure 14.1a: The default segment-file mapping

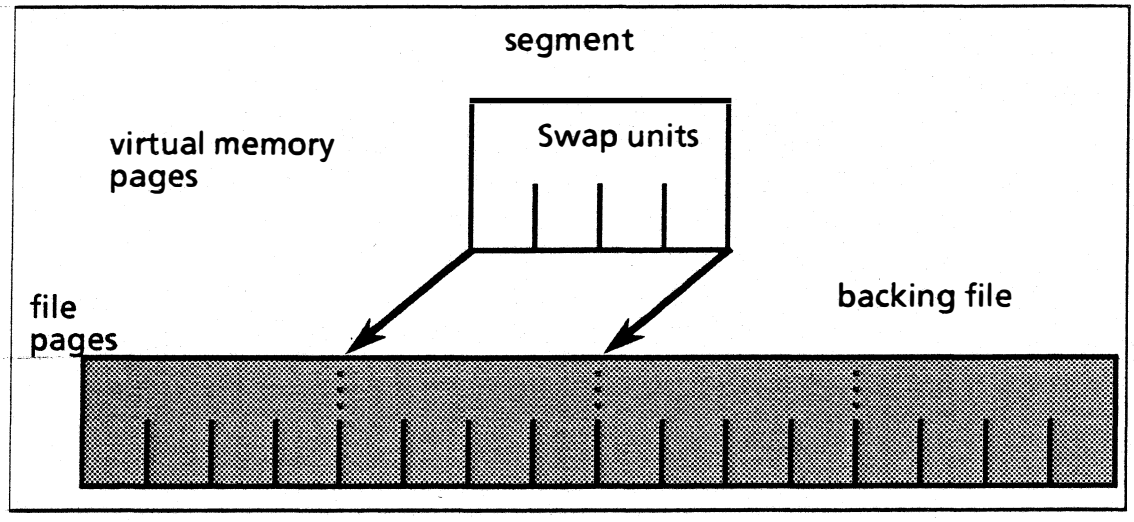


Figure 14.1b: Backing file length is a multiple of segment length

The easiest way to think about how the backing file works is to assume that the segment and its image in the backing file contain the same data.The real situation, however, is a little more complicated since there is a time lag between when something is changed in a segment and when that change is reflected in the backing file. If you constantly update the backing file you would not realize any performance improvement over normal file input/output. The whole advantage that a segment offers is that it is a buffer, and as a buffer, its contents are written out only occasionally.

There are three occasions when a segment is copied out into its backing file. First, when a segment is deleted, it copies out its data before it disappears. Second, you can explicitly back up a segment by calling the procedure **MSegment.ForceOut**. (We discuss this more completely later in this chapter.) Finally, the segment copies out data whenever it is swapped out of real memory (this is totally invisible to you.)

In most other ways, segments are like blocks of real memory storage. For example, segments start at an address that is fixed when they are created, so, like nodes in a heap, segments do not move once they are created. Further, you can reference a segment starting address, just like the address of the first word in a record node, with a **LONG POINTER** variable. Thus, you can impose any data structure upon a segment just by creating a variable that is a pointer to a data object and then assigning to that pointer the address of the segment. An entire data file can be manipulated merely by mapping sections of the file onto segments and then imposing some record structure onto the segments.

14.2.1 Creating a segment

You create a segment with **MSegment.Create**:

```
MSegment.Create: PROCEDURE[
  file: MFile.Handle ← NIL,      --handle to the backing file
  release: MSegment.ReleaseData,
  fileBase: File.PageNumber ← 0,
  pages: Environment.PageCount ← defaultPages,
  swapInfo: MSegment.SwapUnitOption ← defaultSwapUnits]
  RETURNS [segment: MSegment.Handle];

MSegment.Handle: TYPE = LONG POINTER TO MSegment.Object;

File.PageNumber: TYPE = LONG CARDINAL;

Environment.PageCount: TYPE = LONG CARDINAL;
```

Create creates a segment; the operations that you can perform on the segment are restricted by the access associated with the file that is passed in. You should note that ownership of the file is passed to the **MSegment.Handle** via the **file** parameter; if you want to maintain control of the file you will need to copy the file handle with **MFile.CopyFileHandle** before creating the segment. If file is **NIL**, **MSegment** will automatically create a nameless, temporary file to act as the backing store for the segment; when the segment is deleted the backing file will also vanish.

The **release** parameter is used to provide a **PleaseReleaseProc** to the file system when you want to allow multiple processes to access the file (see the MFile chapter for a discussion of **PleaseReleaseProcs**). If you do not want to share the file you can simply set the release value to [**NIL, NIL**]. **fileBase** is the starting point of the segment on the file and is defaulted to the beginning of the file. The **pages** parameter is the number of pages you want for the segment length; the default is the length of the file. **swapInfo** is the number of pages that each swap unit contains. This number should generally not be greater than one-tenth of the size of real memory.

14.2.2 Copying segments to and from files

Once you have established a segment on a backing file, you may want to copy the contents of that segment into another file. For example, you can selectively extract information from a large data file and create a new file that contains only a small subset of selected information. You can perform this selective copying with **MSegment.CopyOut**.

```
MSegment.CopyOut: PROCEDURE [
  segment: MSegment.Handle,        -- segment containing the desired information
  file:MFile.Handle,               -- file to be copied into
  fileBase: File.PageNumber,       -- copy starting position within the file
  count: Environment.PageCount];   -- copy count pages into the file
```

Beginning with the first page of **segment**, **CopyOut** copies **count** pages of **segment** into **file**, starting in position **filebase** of file. **CopyOut** is illustrated in Figure 14.2.

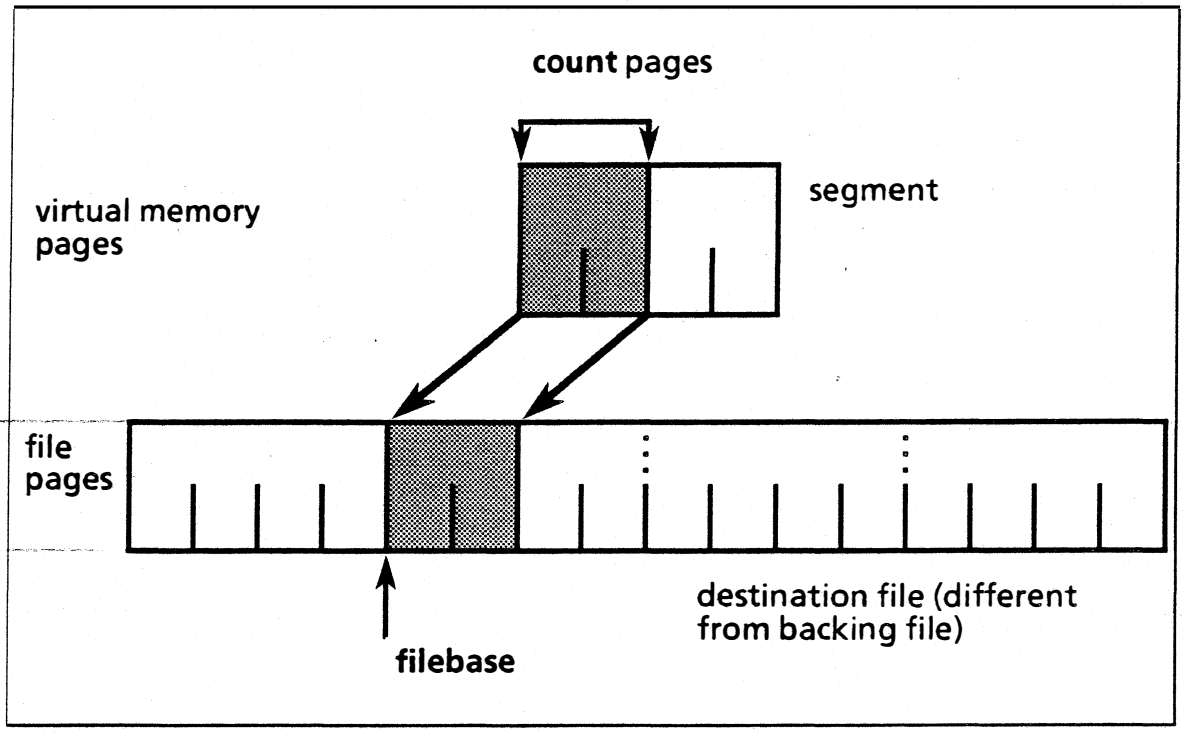


Figure 14.2 Segment.CopyOut

You may also wish to copy information from a file into a segment that is mapped to another file; you can do this with **MSegment.CopyIn**.

```
MSegment.CopyIn: PROCEDURE [
  segment: MSegment.Handle,          -- segment to be copied into
  file:MFile.Handle,                 -- file containing the desired information
  fileBase: File.PageNumber,         -- copy starting position within the file
  count: Environment.PageCount];     -- copy count pages into the segment
```

Notice that the parameters to **CopyIn** are identical to those of **CopyOut**; the only difference between the procedures is the direction of the copy. The two copy procedures are similar to the read and write operations of a traditional file system. Here is an example that copies one file to another using **CopyIn** and **CopyOut**:

```
DIRECTORY
Environment USING [bytesPerPage, PageCount, PageNumber],
Exec USING [AddCommand, ExecProc, GetToken, Handle, OutputProc],
Format USING [StringProc],
MFile USING [ByteCount, Error, GetLength, Handle, ReadOnly, Release, SetLength, WriteOnly],
MSegment USING [Create, CopyIn, CopyOut, Delete];

CopySegment: PROGRAM IMPORTS Exec, MFile, MSegment =
BEGIN
  GetFileNames: PROCEDURE[h: Exec.Handle] RETURNS[inFile, outFile: MFile.Handle ← NIL] =
  BEGIN
    inName, outName: LONG STRING ← NIL;
    [inName, ] ← Exec.GetToken[h]; -- Read in the filenames and ignore the switch
    [outName, ] ← Exec.GetToken[h];

    inFile ← MFile.ReadOnly[name: inName, release: [NIL, NIL]];
    outFile ← MFile.WriteOnly[name: outName, release: [NIL, NIL], type: unknown!MFile.Error
      => {MFile.Release[inFile]; REJECT}]; -- pass signal to Copy after releasing inFile
  END;

--create a segment on a temporary file and transfer the file contents
  TransferSegments: PROCEDURE[inFile, outFile: MFile.Handle] =
  BEGIN
    LengthOfFile: MFile.ByteCount = MFile.GetLength[file: inFile];
    ExtraBytes: MFile.ByteCount = LengthOfFile MOD Environment.bytesPerPage;
    FullPages: Environment.PageCount = LengthOfFile / Environment.bytesPerPage;
    PagesToTransfer: Environment.PageCount =
      FullPages + (IF ExtraBytes > 0 THEN 1 ELSE 0);     -- calculate the number of pages
                                                         -- to transfer
    segmentSize: Environment.PageNumber = 4;             -- segments are 4 pages long

    bufferSegment: MSegment.Handle ← MSegment.Create[release: [NIL, NIL], pages:
      segmentSize];
    ExtraPages: Environment.PageCount = PagesToTransfer MOD segmentSize;
    numberOfTransfers:LONG CARDINAL = PagesToTransfer / segmentSize +
      (IF ExtraPages = 0 THEN 0 ELSE 1);     -- calculate the number of segment copies to make

    MFile.SetLength[outFile, LengthOfFile]; -- set the length of the output file

  -- perform the copies from the input file to the segment and then from the
  -- segment into the output file.

    FOR pageCount:LONG CARDINAL IN [0..numberOfTransfers) DO
      MSegment.CopyIn[bufferSegment, inFile, pageCount*segmentSize, segmentSize];
      MSegment.CopyOut[bufferSegment, outFile, pageCount*segmentSize, segmentSize];
    ENDLOOP;
    IF bufferSegment # NIL THEN {MSegment.Delete[bufferSegment]; bufferSegment ← NIL};
  END;
```

```
-- Copy is called when the user types - Copy infilename outfilename
Copy: Exec.ExecProc =
BEGIN
   Write: Format.StringProc = Exec.OutputProc[h];
   inFile, outFile: MFile.Handle ← NIL;
   [inFile, outFile] ← GetFileNames[h! MFile.Error = >
      {Write["invalid or missing filename"L]; GOTO exit}]; --get filenames from Executive

   TransferSegments[inFile, outFile]; --create the segment and perform the file transfer
   Write["File transfer complete"L];
   MFile.Release[inFile]; --release the files since MSegment never owned them
   MFile.Release[outFile];
   EXITS
      exit = > RETURN; -- return to Executive if an error occurred when acquiring files
END;

--Mainline code
   Exec.AddCommand[name: "Copy.~"L, proc: Copy]; -- register the Copy command

END. --end of program
```

This program is invoked when you type **Copy** *inputfile outputfile* in the Executive. The **Copy** procedure first calls **GetFileNames**, which reads the names of the input and output files from the Executive, and then acquires the file handles, creating an output file if one does not already exist. Next, **Copy** calls **TransferSegments**, which creates a segment on a temporary file. (When no backing file is explicitly specified, **MSegment** automatically creates a temporary backing file.)

TransferSegments then transfers segments from the *inputfile* to the *outputfile* by alternating **CopyIn**s and **CopyOut**s until the entire *inputfile* has been copied to the *outputfile*. After the entire file is copied, we call **MSegment.Delete** to delete the segment on the temporary file, and then we release the input and output files. Calling **MSegment.Delete** on a segment with a temporary backing file automatically releases the backing file.

An important thing to notice from this example is that you need to set the length of the *outputfile* before you start the transfer. Otherwise, even if you create the output file with the correct length (number of pages) the file system will think that the file has a zero length (byte length). To set the length, call **MFile.SetLength**, as shown in the above example.

14.2.3 Forcing pages to the disk

When data integrity is very important (e.g. real time database applications), you will want to frequently force *dirty pages* of the segment to the disk. When you are entering information into a database, you may add information to a segment for some time before the segment is deleted, and implicitly written to the disk. If the system crashes during this time period, all your new information will be lost. To insure against this type of loss you should periodically force all dirty pages to the backing file with **MSegment.ForceOut**.

```
MSegment.ForceOut: PROCEDURE [segment: MSegment.Handle];
```

The example in the next section contains an example of **MSegment.ForceOut**.

14.2.4 Direct access within segments

Since segments let you create an arbitrary data structure on a file, you need a method to modify that data structure. To access a specific part of a segment you get a pointer to the start of the segment and then add an offset to the pointer to reach the address you want to modify. You can get the starting address of a segment by calling **MSegment.Address**:

```
MSegment.Address: PROCEDURE [segment: MSegment.Handle] RETURNS [LONG POINTER];
```

You can access the segment's contents by defining a Mesa structure for the segment and then **LOOPHOLE**ing the pointer returned by **MSegment.Address** into that structure. (Note: **LOOPHOLE** is a Mesa language operator that allows you to convert any data type into any other data type, provided that the two data types occupy the same number of words. See the MLM for details.) This technique gives you a view into the segment without having to **LOOPHOLE** the segment's data into a separate structure. Of course, you must know the structure of the segment before you can perform the preceding operations.

When you first insert information into your segment you must choose a structure for that information. The structure you choose is arbitrary, but ideally there should be an integral number of records in each segment (e.g. having one-half of a record or 1.7 records in a segment is poor practice.) You must also declare a pointer to this structure in order to access the record fields within the segment. *It is important to remember that you cannot assign default values to the structure, since it is only a template for the segment and not a variable.* The example below uses a template to write data into two areas of a segment.

```
DIRECTORY
Exec USING [AddCommand, ExecProc, GetToken, Handle],
MFile USING [Handle, ReadWrite, SetLength],
MSegment USING [Address, Create, Delete, ForceOut, Handle];

DirectAccessSegment: PROGRAM IMPORTS Exec, MFile, MSegment =
BEGIN
  -- Declare record structure for accessing the segment
  Data: TYPE = MACHINE DEPENDENT RECORD[
    name(0): PACKED ARRAY[0..14) OF CHARACTER,
    address(7): PACKED ARRAY[ 0..14) OF CHARACTER,
    id(14): LONG CARDINAL];

  SegmentOfData: TYPE = PACKED ARRAY [0..32) OF Data;
  tenPages: CARDINAL = 5120;  --ten pages of 512 bytes each

  GetFile: PROCEDURE[h: Exec.Handle] RETURNS[inFile: MFile.Handle ← NIL] =
  BEGIN
    inName: LONG STRING ← NIL;
    [inName,] ← Exec.GetToken[h];
    inFile ← MFile.ReadWrite[name: inName, release: [NIL, NIL], type: binary,
        initialLength: tenPages];
  END;
```

```
-- create segment and write to the 10th and 18th records
ModifySegment: Exec.ExecProc =
BEGIN
   segment: MSegment.Handle ← NIL;
   data: LONG POINTER TO SegmentOfData; -- pointer for manipulating records
   file: MFile.Handle ← GetFile[h];          -- read file name from Executive
   MFile.SetLength[file,tenPages];                  -- ensure file is the proper length
   segment ← MSegment.Create[file: file, release: [NIL, NIL], pages: 2];
   data ← MSegment.Address[segment];

   -- write to the 10th record of the segment
   data[10].name ← ['M,'a,'r,'k,' ,' ,' ,' ,' ,' ,' ,' ,' ,' ];
   data[10].address ←['X,'e,'r,'o,'x,' ,' ,' ,' ,' ,' ,' ,' ,' ];
   data[10].id ← 199;

   -- perform other operations, but ensure that information
   -- is safe by forcing out the segment's dirty pages
   MSegment.ForceOut[segment];

   -- perform another write to the data segment in record 18
   data[18].name ← ['F,'r,'e,'d,' ,' ,' ,' ,' ,' ,' ,' ,' ,' ];
   data[18].address ←['H,'i,'l,'l,'v,'i,'e,'w ,' ,' ,' ,' ,' ,' ];
   data[18].id ← 276;

   MSegment.Delete[segment]; -- delete will write dirty pages
END;

--Mainline code
Exec.AddCommand[name: "ModifySegment.~"L, proc: ModifySegment];

END.
```

The above program registers the command ModifySegment with the Executive. Invoking this command calls the procedure **ModifySegment**, which reads the name of a file from the Executive and sets the length of the file to ten pages (since the file may be new and not have any length). Next, we create a segment of length two pages on the file and set a pointer (**data**) to the beginning of the segment (with the implied structure of **SegmentOfData**.) Thus, you can write to the segment as if it were a variable of type **SegmentOfData**.

14.2.5 Copying segment handles

Under certain circumstances you may want to have more than one process use the same segment. For example, suppose you want to have a database that gives a handle to the segment of interest to each process that wants read access. To share a segment in this way, you must copy the segment **Handle** with **MSegment.CopySegment**.

```
MSegment.CopySegment: PROCEDURE [
   segment: MSegment.Handle] RETURNS [newSegment: MSegment.Handle];
```

The new handle will have the same access as the old; thus, you can have more than one **readWrite** handle to a segment. As with files, it is your responsibility to insure against conflicts when overwriting data in this segment.

14.3 Summary

The **MSegment** interface allows you to access the contents of a file by mapping the file to a segment of virtual memory. Mapping to a segment allows you to create a view of a file that corresponds to a data structure of your choice, thus allowing you to treat files nearly as if they were variables. When you create a segment, you need to supply a backing file. If you don't explicitly name your backing file, **MSegment** will create a temporary backing file for you.

MSegment.CopyIn and **MSegment.CopyOut** provide a method of copying information to and from files; these procedures copy data between a segment and a file much like the read and write, operations of traditional file systems. Neither procedure affects ownership of the files involved.

MSegment.Address returns the virtual memory address of a given segment. You use this procedure to directly access information contained in the segment via **LONG POINTER**s to the data structure stored in the segment.

Since a segment is just a buffer, changes must be written to the backing file before they become permanent. Thus, you should occasionally call **MSegment.ForceOut** to force dirty pages to the disk. Use this procedure when the integrity of the information you put in the segment is of great importance.

When you create a segment on a file, the ownership of the file is passed to the **MSegment.Handle**; you must copy the file handle to a separate handle if you want to keep a pointer to the file. When you delete a segment, the backing file for that segment is released. If your segment is backed by a temporary file, the file is deleted when you delete the segment.

14.4 Style

When dealing with segments, there are several important style issues. First, you should think carefully about the size of your swap units. If they are too large (more than one-tenth of the size of real memory), system performance will degrade severely. On the other hand, you can also have swap units that are too small. For example, if your program accesses a segment that contains data items that are 5 pages long, you should not have a swap unit size smaller than 5 pages. A smaller swap unit would require at least two disk accesses, whereas a 5 page swap unit might retrieve the entire structure in one access.

Segment size is another important consideration. An 8010 has only 2^{22} words, (16,000 pages) of virtual memory. All clients in the XDE must share this same virtual address space; thus, mapping a very large segment may not leave enough virtual memory for the remaining processes. Again it is important to understand the memory requirements of your process and the requirements of other running processes. On the other hand, if you use small segments, the time required to map each segment will become quite large; thus you should perform as little mapping as possible.

Given these two conflicting problems, a good rule of thumb is to map entire files if you will only use them for a short time (e.g. copying a file); or if you know that you will access each page in the file. You should map smaller segments when other processes must run in parallel or if the file is too big to map with available virtual memory.

14.5 References

Information on **MSegment** is contained in the *Mesa Programmer's Manual.*

The **Space** chapter in the *Pilot Programmer's Manual* describes how Pilot implements the virtual memory system.

14.6 Exercise

The exercise for this chapter asks you to construct a data structure for a simple airline reservation system on a file, and then manipulate that structure via segments. The data structure consists of two parts; the first part is a directory and the second part is the data.

The *directory* should contain an array of data records, each of which contain **FlightNumber**, **FromCity**, **ToCity**, and an **InUse** field. When you want to look for a particular flight number, you simply test each **FlightNumber** that is **inUse** until you find a match, and then calculate an address to access the data.

The *data* is composed of **Flights**, where each flight has an **ARRAY** of seats. Each seat has a **name**, **seatNumber**, and **InUse** field. You should make one flight fit on each data segment; thus, when you find a desired flight in the directory, you must map its corresponding data into a segment.

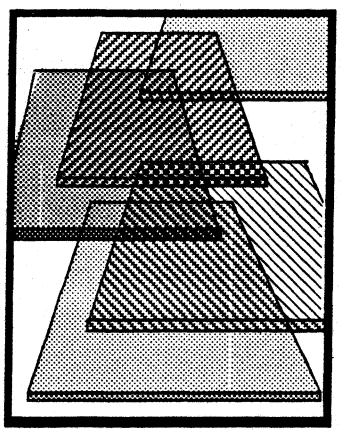
When a **FlightNumber** is deleted, its **inUse** **BOOLEAN** is set to **FALSE** and the **inUse** fields in the data object are marked as **FALSE**. If you want to insert information into the file you must first find an empty slot in the directory (by checking **inUse**), then you can insert into the first field in the data segment that is not **inUse** .

OpenSession accesses the data file and maps a segment to that file. **CloseSession** deletes the segment, thus releasing the data file.

The sizes and definitions for the above data objects are defined in the interface module Reservation.mesa. Your task is to implement five procedures, **OpenSession**, **CloseSession**, **Insert**, **Delete**, and **ShowFlights**, that manipulate and display the data structure. We have provided a tool interface that will call your procedures and accept input data. The tool is called ReservationTool.mesa and a template for your procedures is on ReservationTemplate.mesa

Notes:

15



Streams

Mesa *streams* allow you to transport serial data (bytes, words, or blocks of bytes) to and from various devices. Streams are most commonly used for reading and writing local files, but they can be used with other devices as well (such as floppy disks.)

In this chapter, we discuss how to attach a stream to a storage device, how to access the data once the stream is attached, and how to dispose of the stream when you are finished with it. We also discuss how to associate PleaseReleaseProcs with streams.

15.1 Definition of terms

Stream — A *stream* is an abstraction for device- and format-independent sequential access to a collection of data. Some streams also provide random access to the data.

Stream Object — A *stream object* contains the data and procedures for operations on the stream.

Stream Handle — A *stream handle* is a pointer to a stream object that identifies the particular stream being accessed.

Transducer — A *transducer* is a software entity (*e.g.*, module or configuration) that implements a stream connected to a specific device or medium. The **MStream** interface implements a Pilot transducer for accessing a file as a positionable byte stream.

Stream component manager — A *stream component manager* is the software entity that implements a stream component–a transducer, filter or pipeline. Although Pilot supports filters and pipelines, XDE does not currently use them; thus, stream component manager is synonymous with transducer.

15.2 Discussion

Mesa provides streams to remove the detail involved in transmitting and receiving data from devices such as local disk files. Mesa streams enable you to think about input and

output without knowing the details of the device with which you will communicate, and without writing low-level get- and put-data routines. Thus, your program need not depend on the nature of the device, and you can focus on the main logic of a program without being distracted by the details of the device. Additionally, if the details of the device itself should change in the future, you will not have to rewrite your program.

The stream abstraction is device and data-*independent*, but creating a stream is device- and data-*dependent*. Thus, you will use device-specific routines to create a stream, and then use the more general **Stream** interfaces to perform operations on the stream. In this chapter, we discuss only one form of stream creation: creating a stream to a local disk file.

To use streams you must:

1. Declare a stream handle.
2. Create the stream.
3. Perform operations (read or write) on the stream.
4. Delete the stream.

15.2.1 The stream handle

To a client of the **Stream** interface, a stream is a variable of type **Stream.Handle**, which is defined as :

```
Stream.Handle: TYPE = LONG POINTER TO Stream.Object;
Stream.Object: TYPE = RECORD [...];
```

A **Handle** references an **Object** that defines the mechanisms for data transfer to and from the particular device for which its stream was created. Transducers allocate and initialize an **Object** with the necessary information and return the **Handle** to the client. This **Handle** is then passed as a parameter to various operations in the **Stream** interface to identify the particular stream on which the operations are to be performed.

15.2.2 Creating a stream

The **MStream** interface supplies a convenient transducer for creating a stream to a local disk file. The following calls in the **MStream** interface are used to create streams to files:

```
MStream.Create: PROCEDURE [file: MFile.Handle, release: MStream.ReleaseData]
  RETURNS [MStream.Handle];

MStream.ReadOnly: PROCEDURE [name: LONG STRING, release: MStream.ReleaseData]
  RETURNS [MStream.Handle];

MStream.ReadWrite: PROCEDURE [
  name: LONG STRING, release: MStream.ReleaseData, type: MFile.Type ← unknown]
  RETURNS [MStream.Handle];

MStream.WriteOnly: PROCEDURE [
name: LONG STRING, release: MStream.ReleaseData, type: MFile.Type]
  RETURNS [MStream.Handle];
```

The **Handle** returned by these procedures is really a **Stream.Handle**, since **MStream** defines its **Handle** like this:

```
MStream.Handle: TYPE = Stream.Handle
```

Creating a stream on a file is a two step process: first you acquire the file, and then you attach a stream to that file. To use **MStream.Create**, which is the most general of the four calls, you must have a handle to the file. You pass the file handle to **Create**, which returns a stream handle. The call to **Create** makes the stream handle the new owner of the file.

MStream.ReadOnly, **MStream.ReadWrite**, and **MStream.WriteOnly**, on the other hand, are "accelerators"; you pass in a file name, and they acquire the file for you and attach the stream to it. We discuss when to use **Create** and when to use one of the accelerators in the next section.

The **release** parameter in the above procedure is of type **MStream.ReleaseData**, declared as

```
MStream.ReleaseData: TYPE = RECORD
  [proc:MStream.PleaseReleaseProc ← NIL,
  clientInstanceData: LONG POINTER ← NIL];
```

This parameter is used when a process wants access to a file that is currently attached to a stream. For example, if you have a read-only stream attached to a file and another process wants to write that file, then the **release** variable comes into play: the procedure specified by the **proc** field is called with the **clientInstanceData** pointer as a parameter. If **proc** is **NIL**, access will be denied. This method of access is discussed further in section15.2.7.

15.2.2.1 Examples of creating streams on files

To open a read-only stream attached to a file named Input.text, you could code:

```
-- variables
inputFile: MFile.Handle ← NIL;
fileReleaseData: MFile.ReleaseData ← [NIL, NIL];
inputStream: MStream.Handle ← NIL;
streamReleaseData: MStream.ReleaseData ← [NIL, NIL];

-- mainline code
-- use an MFile procedure to initialize the file handle and prepare for reading
inputFile ← MFile.ReadOnly[name: "Input.text"L, release: fileReleaseData];
inputStream ← MStream.Create[file: inputFile, release: streamReleaseData];
inputFile ← NIL; --clear the MFile.Handle.
```

Note the last line of the above example, where the **MFile.Handle** is set to **NIL**. Strictly speaking, this is not necessary, but it is advisable: once you have created the stream, ownership of the file's handle is transferred to the stream. Thus, you should set the file handle to **NIL** to avoid inadvertently using it.

In general, you will have to use **MStream.Create** when you need to do some processing with the **MFile.Handle** between calls to **MFile.ReadOnly** and **MStream.Create**, or if you need to open the file with a **MFile.Access** other than **readOnly**, **writeOnly** or **readWrite**. (If the latter is the case, call **MFile.Acquire** with the desired access.) For the most part, however, you can just use one of the accelerators, as illustrated below:

```
-- variables
inputStream: MStream.Handle ← NIL;
streamReleaseData: MStream.ReleaseData ← [NIL, NIL];

-- mainline code
-- use an MStream accelerator to acquire the file and create the stream
inputStream ← MStream.ReadOnly[name: "Input.text"L, release:
  streamReleaseData];
```

15.2.3 The basic data transmission operations

Once you have a stream, you can perform I/O. The basic input procedures are **Stream.GetByte**, **Stream.GetChar**, **Stream.GetWord**, and **Stream.GetBlock**. These procedures return a byte, a character, a machine word, or a block from the stream whose handle is **sH**. Since **GetBlock** is slightly different from the other three, we discuss it separately in the next section. The relevant declarations are:

```
Stream.GetByte: PROCEDURE [sH: Stream.Handle] RETURNS [byte: Stream.Byte];

Stream.GetChar: PROCEDURE [sH: Stream.Handle] RETURNS [char: CHARACTER];

Stream.GetWord: PROCEDURE [sH: Stream.Handle] RETURNS [word: Stream.Word];
```

These procedures return the next byte, character, or word (respectively). The amount of space needed to store the data being returned is predefined by the data's type.

The basic output operations for streams are **Stream.PutByte**, **Stream.PutChar**, **Stream.PutWord**, **Stream.PutString**, and **Stream.PutBlock**. (**PutBlock** is discussed in the next section.) The procedures are declared as follows:

```
Stream.PutByte: PROCEDURE [sH: Stream.Handle, byte: Stream.Byte];

Stream.PutChar: PROCEDURE[sH: Stream.Handle, char: CHARACTER];

Stream.PutWord: PROCEDURE[sH: Stream.Handle, word: Stream.Word];

Stream.PutString: PROCEDURE [sH: Stream.Handle, string: LONG STRING,
  endRecord: BOOLEAN ← FALSE];
```

The **endRecord: BOOLEAN** parameter of **PutString** controls how the stream deals with physical record boundaries. This should be defaulted to **FALSE**, unless you are doing I/O that relies on physical record boundaries.

Here is an example that creates two streams and does a byte-by-byte copy of one stream to the other :

```
stream1, stream2: MStream.Handle ← NIL;
releaseData: MStream.ReleaseData ← [NIL,NIL]; -- do not allow access by others
stream1 ← MStream.ReadWrite["File1"L,releaseData];
stream2 ← MStream.ReadWrite["File2"L,releaseData]; --create both streams

-- copy information from stream1 to stream2 until the end-of-stream is reached

DO
    Stream.PutChar[stream2, Stream.GetChar[stream1 !Stream.EndOfStream = > EXIT]];
ENDLOOP;
```

The standard way to recognize end-of-stream is by catching the signal **Stream.EndOfStream**, which is declared as:

```
Stream.EndOfStream SIGNAL [nextIndex: CARDINAL];
```

GetChar, **GetByte**, and **GetWord** all raise this signal when they attempt a **Get** beyond the end of the stream.

15.2.4 Data transmission by blocks

The block procedures are a little different. Here are the declarations:

```
Stream.Block: TYPE = Environment.Block;
Environment.Block: TYPE = RECORD [
  blockPointer: LONG POINTER TO PACKED ARRAY [0..0) OF Environment.Byte
  startIndex, stopIndexPlusOne: CARDINAL];

Stream.GetBlock: PROCEDURE [sH: Stream.Handle, block: Stream.Block]
  RETURNS [bytesTransferred: CARDINAL, why: Stream.CompletionCode,
  sst: Stream.SubSequenceType];

Stream.PutBlock: PROCEDURE [sH: Stream.Handle, block: Stream.Block
  endRecord: BOOLEAN ← FALSE];
```

GetBlock allows the client to buffer the stream's data. You must provide the storage for the buffer, which takes the form of a record pointed to by a variable of type **Stream.Block**. **PutBlock** is like **GetBlock** in that it allows buffering of data. However, since most transducers set up streams with internal buffering of data, it is not necessary to use **Get/PutBlock** just to achieve the efficiency of buffering. (On the contrary, it can cause a second layer of buffering without enhancing I/O speed.) However, blocked I/O is convenient for data already formatted into blocks.

Here is an example of using blocked transfers:

```
bufferSize: CARDINAL = 256;
buffer: PACKED ARRAY[0..bufferSize) OF Environment.Byte;
block: Stream.Block ← [@buffer, 0, bufferSize];
completionCode: Stream.CompletionCode ← normal;
UNTIL completionCode = endOfStream DO
  [block.stopIndexPlusOne, completionCode, ] ←
    Stream.GetBlock[stream1, block];
  Stream.PutBlock[stream2, block];
ENDLOOP;
```

This example illustrates another difference between the block operations and the other data transmission operations. **GetBlock** normally uses the completion code **endOfStream** instead of signalling **endOfStream**. To cause **GetBlock** to raise the signal, you can call **Stream.SetInputOptions** to set **signalEndOfStream** in **inputOptions** to **TRUE**:

```
myStream: MStream.Handle ← NIL;
fileOptions: Stream.InputOptions ← [signalEndOfStream: TRUE]; --allow signal
releaseData: Mstream.ReleaseData ← [NIL,NIL];
myStream ← MStream.ReadWrite["File1"L, releaseData];
Stream.SetInputOptions[sH: myStream, options: fileOptions];
```

15.2.5 Positioning and random accessing streams

There are two procedures that allow random access to data, provided that the physical device and stream component manager that the stream is attached to support random access. The relevant declarations in the **Stream** interface are:

```
Stream.Position: TYPE = LONG CARDINAL;
Stream.GetPosition: PROCEDURE [sH: Stream.Handle] RETURNS [position: Stream.Position];
Stream.SetPosition: PROCEDURE [sH: Stream.Handle, position: Stream.Position];
```

In both of the procedure declarations, the **position** parameter is the byte-index of the next data in the stream to be read or written, where the first byte in the file has the index 0. Here is some Mesa code to illustrate the use of these procedures:

```
-- Read the fiftieth byte in the stream
Stream.SetPosition[inputStream, 49];
byteIn ← Stream.GetByte[inputStream];

-- Read every other byte in the stream
Stream.SetPosition[inputStream, 0]; -- set position to start of file
DO
    byte ← Stream.GetByte[inputStream!Stream.EndOfStream => EXIT];
    nextPosition ← Stream.GetPosition[inputStream] + 1;
    Stream.SetPosition[inputStream, nextPosition];
ENDLOOP;
```

15.2.6 Deleting streams

Since a stream is a connection between a program and a device, the program should never terminate without telling the device that the connection is no longer open. For every stream you create, you must call **Stream.Delete** to close the stream when you are finished with it. Regardless of how you obtained the stream handle, you close it down by calling **Stream.Delete** with the stream handle as the parameter. **Stream.Delete** is declared as:

```
Stream.Delete: PROCEDURE [sH:Stream.Handle];
```

After closing the stream, you should always set the stream handle variable to **NIL**, to ensure that you don't accidentally try to use it later on. Here is an example of using **Stream.Delete**:

```
ioStream: MStream.Handle ← NIL;
releaseData:MStream.ReleaseData ← [NIL, NIL];

ioStream ← MStream.ReadWrite["MyFile"L, releaseData];

-- perform various I/O operations until done with the stream
Stream.Delete[ioStream];
ioStream ← NIL; --insure against accidental access
```

15.2.7 Handling multiple access to streams

As discussed in the MFile chapter, individual processes can cooperatively share files by registering **PleaseReleaseProc**s and **NotifyProc**s. The MStream provides a similar facility that allows a process to share a file to which it has a stream attached. Up to this point you

have seen only NIL PleaseReleaseProcs; the example below illustrates how to use a PleaseReleaseProc when you are willing to share the file.

```
-- CopyDefs.mesa
DIRECTORY
MStream,

CopyDefs: DEFINITIONS =
BEGIN
   FileState: TYPE = {busy, beingReleased, released};
   ExamplePleaseReleaseProc: PROGRAM;
   ChangeState: PROCEDURE[newState: FileState]; -- Monitor entries
   MyReleaseProc: MStream.PleaseReleaseProc;
END.

--Monitor for granting access to the file
DIRECTORY
Stream,
CopyDefs,
MStream;

ExamplePleaseReleaseProc: MONITOR
EXPORTS CopyDefs =
BEGIN
   state: CopyDefs.FileState;

   ChangeState: PUBLIC ENTRY PROCEDURE [newState: CopyDefs.FileState] =
      {state ← newState}; --set the global state to a new state

--PleaseReleaseProc for file
   MyReleaseProc: PUBLIC ENTRY MStream.PleaseReleaseProc =
   BEGIN
      SELECT state FROM
         busy          => RETURN[no];
         beingReleased => RETURN[later];
         released      => RETURN[goAhead];
         ENDCASE       => RETURN[no];
   END;
END.

--CopyStream programs runs in the Executive and copies one stream to another
DIRECTORY
Exec,
Format,
Stream,
CopyDefs,
MStream;
CopyStreamExample: PROGRAM
IMPORTS Exec, MStream,CopyDefs, Stream =
BEGIN
```

```
 -- Takes the names of input and output filenames and returns stream handles
 CreateStreams: PROCEDURE[inName, outName: LONG STRING] RETURNS[inStream, outStream:
 MStream.Handle] =
 BEGIN
    inReleaseData: MStream.ReleaseData ←
 [proc:CopyDefs.MyReleaseProc,clientInstanceData:NIL];
    outReleaseData: MStream.ReleaseData ← [proc: NIL, clientInstanceData:NIL];
    inStream ← MStream.ReadOnly[inName,inReleaseData];
    outStream ← MStream.WriteOnly[outName,outReleaseData,text];
    RETURN[inStream,outStream];
 END;

 -- Deletes both input and output streams and sets their handles to NIL
 DeleteStreams: PROCEDURE[inStream, outStream: MStream.Handle]
    RETURNS[in, out: MStream.Handle] =
 BEGIN
    Stream.Delete[inStream];
    Stream.Delete[outStream];
    RETURN[NIL,NIL];
 END;

 -- perform the actual stream copy
 Copy: PROCEDURE[inStream,outStream: MStream.Handle ] =
 BEGIN
    DO
       Stream.PutChar[outStream, Stream.GetChar[inStream!Stream.EndOfStream = > EXIT]];
    ENDLOOP;
 END;

 -- gets input file and output file and calls procedures to do real work
 CopyStream: Exec.ExecProc =
 BEGIN
    Write: Format.StringProc = Exec.OutputProc[h]; -- Write prints strings to the Exec
    inFileName, outFileName: LONG STRING ← NIL;
    inStream, outStream: MStream.Handle ← NIL;
    [inFileName, ] ← Exec.GetToken[h]; -- discard switches
    [outFileName, ] ← Exec.GetToken[h];
    CopyDefs.ChangeState[busy];          -- file is in use do not allow others to access
    [inStream, outStream] ← CreateStreams[inFileName, outFileName];
    Copy[inStream, outStream];  --do the stream copy
    CopyDefs.ChangeState[beingReleased]; --file will be available soon
    [inStream, outStream] ← DeleteStreams[inStream, outStream]; -- Delete streams
    CopyDefs.ChangeState[released]; -- okay for others to use file
    Write["The file '"L];
    Write[inFileName];
    Write["' has been copied to the file '"L];
    Write[outFileName];
    inFileName ← Exec.FreeTokenString[inFileName];
    outFileName ← Exec.FreeTokenString[outFileName];
 END;
```

```
-- Main Code
Exec.AddCommand[name: "CopyStream.~"L, proc: CopyStream];
CopyDefs.ChangeState[released];
END.
```

The mainline code for this example calls **Exec.AddCommand** to register the **CopyStream** command with the Executive. Thus, the **CopyStream** procedure is called whenever the user runs the program. **CopyStream** first reads in two file names (**inFileName** and **outFileName**) as arguments by calling **Exec.GetToken. GetToken** reads a token and a switch (separated by a "/") from the command line; in this case there are no switches, so the second argument returned by **Exec.GetToken** is elided. After the file names are acquired, but before the streams are created, **CopyStream** sets the state of the input file to busy so no other process will be able to access the file. **CopyStream** then creates the streams, performs the file transfer, and finally deletes the streams.

CreateStreams takes the names of the input and output files and creates a **ReadOnly** stream for the input file and a **WriteOnly** stream for the output file. The output file has a null **PleaseReleaseProc**; thus no other processes can gain access to the file until the stream is deleted. Since the output file is going to be rewritten by the **CreateStreams** command, other processes should not be allowed access. However, we are assuming that more than one process may want to access the input file while the **CreateStreams** command is executing. You want to allow others to have access whenever you are not using the file in a critical way (*i.e.* reading information from the file). Thus, to maximize the time other processes may access a file, **inStream** has an associated **PleaseReleaseProc**.

When a stream has an associated **PleaseReleaseProc** Tajo calls that **PleaseReleaseProc** whenever another process wants to access a file currently in use. The **PleaseReleaseProc** can return any of four enumerated values defined below:

```
MFile.ReleaseChoice: TYPE = {later, no, goAhead, allowRename}
```

In the above example, **MyReleaseProc** is called when another process attempts to access the input file. **MyReleaseProc** checks the state variable and returns the corresponding **ReleaseChoice** to Tajo. Tajo in turn either grants access to the requesting process or raises the appropriate **SIGNAL**. In this fashion, processes can share files cooperatively without direct communication (or even knowing of each others' existence.) Note that the **PleaseReleaseProc** and the procedure **ChangeState** must both be **MONITOR ENTRY**s to ensure that the state returned is the correct state of the stream.

After the stream is created, **CopyStream** calls **Copy**, which performs a simple character character transfer from the input file to the output file. Reaching the end of the input file raises a signal, which causes the program to exit the loop. Immediately after **Copy** is exited, the **state** of the file is set to **beingReleased**. Thus, processes that attempt to access the file are informed that it will be released soon and that they should try again later.

DeleteStreams relinquishes control over the streams and sets the **MStream.Handle**s to **NIL**. After **DeleteStreams** is finished the **CopyStream** command has no ability to access the input or output streams and other processes can now use the files. Finally, **CopyStream** makes a call to **CopyDefs.ChangeState** to set the state to **released**.

15.3 Summary

The **MStream** interface implements a transducer for creating streams connected to files on the local disk. You use **MStream** to create a stream for a file, and then use the more general **Stream** interfaces to perform operations on the stream.

Once you have created a stream, you can send output through it (**Stream.PutByte**, **Stream.PutChar**, etc.) and receive input from it (**Stream.GetByte**, **Stream.GetChar**, etc.). There are also procedures that allow you to position a stream (**Stream.GetPosition** and **Stream.SetPosition**), provided that the device to which the stream is connected allows random access.

When you are finished using a stream you should use **Stream.Delete** to close it. If the stream was for a file, you should be careful to set the file handle to **NIL** to prevent referencing it after its stream has been closed.

There are several aspects of using streams that we did not cover in this chapter and that you might want to investigate on your own, such as the **SIGNAL Stream.TimeOut** , the **ERROR Stream.InvalidOperation**, the attention flag procedures such as **SendAttention** and **WaitForAttention**, the procedure **SendNow**, and the procedure **SetSST**. These advanced concepts are documented in Chapter 3 of the *Pilot Programmer's Manual.*

15.4 Style

"Object-oriented programming" is a style of programming in which objects existing in the programming environment contain the definitions of operations. So, if you want an entity to perform some task, you just request *what* you want–the entity itself will determine *how* the task is to be accomplished. One advantage of this style is that the code for a given task is isolated in the object, and is (presumably) correct, so clients who want the task performed do not have to "re-invent the wheel" (with the attendant risk of inventing one with a flat tire). Secondly, the implementation of the task can be modified internally to the object without affecting the clients–they just continue to request the services, which the object provides in the usual way. The notions of abstraction and information hiding make the object-oriented approach a highly desirable programming methodology. It is well illustrated by the stream concept in Mesa, which, in conjunction with the stream component manager, determines "how" a variable such as an **Environment.Byte,** will be transferred when a client program requests such a transfer, regardless of the kind of device to which the stream is attached.

15.5 References

Chapter 3 of the *Pilot Programmer's Manual* provides background information about streams and gives their definition in Mesa.

The **MStream** chapter in the *Mesa Programmer's Manual* defines the interface for a transducer, for MStream.

Read the **MFile** chapter in the *Mesa Programmer's Manual*, paying particular attention to the material on file access.

15.6 Exercise

In this exercise you will perform a telephone directory update by applying changes contained in a *change log* onto a *master file* that contains the current directory. The directory consists of a series of fixed-size records, sorted alphabetically by name. The records contain the following information:

name
address
phone number

Basically, you use the tool to create a change log, then you integrate that change log with the master directory. You have to write the code that integrates the change log with the master directory.

There can be three kinds of changes in the change log: additions, deletions, and changes. To create a change log, you add entries one at a time and then invoke the **CreateChangeLog!** command. This command writes the *change log* in free form with fields separated by "/"s. The particular command that should be applied is denoted by a single letter **D** (Delete), **A** (Add) or **C** (Change). For example, a command that adds a new entry into the directory would look like this:

A/John Smith/2323 University Ave, Palo Alto/415-323-3399/

Thus, you follow these steps to create a change log:

1. Select a command type from the enumerated command field.
2. Fill in the name, address and phone fields.
3. Invoke **AddCommand** to add this command to the list of commands to perform.
4. Repeat steps 1-3 until all desired commands have been entered.
5. Type in the name of the *change log* file into the ChangeLog field.
6. Invoke **CreateChangeLog!** to create the change log.

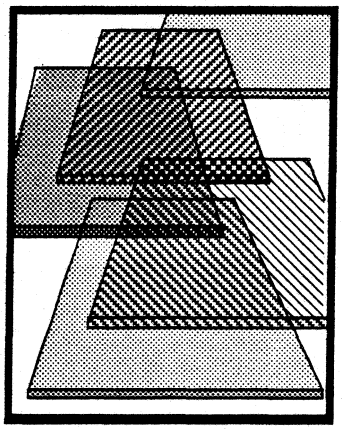
Your assignment is to write the implementation procedure for the **UpdateDirectory** command. This procedure is defined in the **DirectoryDefs** interface, and is called from the tool code. Thus, all you have to do is write the implementation and export it to the interface. To simplify things, we suggest that you put your implementation is a separate module, rather than adding it to the existing implementation module.

Both the old master file and the change log are sorted alphabetically by name; thus, your assignment is essentially to merge the two alphabetical listings and write the merged list to a new file. You should read from the old directory by blocks that contain no more than 8 records and read the *change log* by characters.

The old *master file* is stored as OldDir and the other required files are DirectoryTool.mesa, DirectoryImpl.mesa and DirectoryDefs.mesa. The basic assignment is to implement only the Add command, but for a more challenging exercise try to implement Change and Delete too.

Notes:

16



The FormSWLayout Tool

In chapter 12, we discussed programs that use the Executive for a user interface. In this chapter, you will take the next step towards understanding and using Tajo: you will learn how to generate a tool window interface using a tool called the FormSWLayoutTool.

The extensive layering of the XDE means that there are many different levels of routines that you can use to create a window interface, depending on the degree of flexibility that you want. For example, at the lowest level, you would have to write code to "paint" the window, to display text within that window, and to perform scrolling, selection, and cursor management. Usually, however, you will use system interfaces to perform these kinds of tasks for you; you don't have to think about low-level details unless you want unusual features or functionality.

This chapter introduces a tool called the FormSWLayoutTool, which is an *applications generator*: a tool that helps you write tools that have a window interface. When you use the FormSWLayoutTool, you are freed from writing the code to create the window interface; you need only write the code to actually implement the commands that you want your tool to perform.

This chapter focusses exclusively on using the FormSWLayoutTool; the next chapter, *Tool Window Interfaces*, explains the code that this tool produces, and discusses how to modify it or how to write your own window interface. You should run the FormSWLayoutTool in your CoPilot or Tajo volume and experiment with it as you read this chapter. (If you are familiar with the use of this tool, you should skim the chapter and go straight to the exercises.)

16.1 Preliminary reading

Read the sections of the User Interface chapter of the *Xerox Development Environment User's Guide* that discuss form subwindows.

16.2 Definition of terms

File subwindow	A *file subwindow* is a text subwindow that uses a disk file as its backing store. (*Backing store* refers to the data object used to hold the information that is displayed in the window.)
Form item	*A form item* is an item that appears in a form subwindow. Form items have a keyword (tag) and an associated field. The keyword serves as a reminder of a command or parameter; the field is where the user enters his chosen value for that parameter.
Form subwindow	A *form subwindow* provides the user with a means to indicate parameters, options, and commands for the tool.
Message subwindow	A *message subwindow* provides a simple way to post feedback to the user.
String subwindow	A *string subwindow* is a text subwindow whose backing store is a **LONG STRING**.
Text subwindow	A *text subwindow* provides a way to view text from a wide variety of sources. File and string subwindows are specific types of text subwindows.
TTY subwindows	A *TTY subwindow* provides teletype interaction with the user.

16.3 Discussion

In the XDE, there are several different standard subwindow types, each of which provides a different function. Basically, windows are composed of different combinations of subwindows, depending on the functionality that is desired. During the evolution of Tajo, many tool builders have chosen the same combination of subwindows: a message subwindow, a form subwindow, and a file subwindow. Because this combination is very common among existing tools, it has evolved into the "canonical" Tajo window. Furthermore, because windows are at the heart of Tajo, the system interfaces provide very strong support for creating and using the existing subwindow types. This means that much of the code to create a standard window is identical from tool to tool; you can create a new tool interface from an existing one just by changing the layout of the form subwindow.

The FormSWLayoutTool takes advantage of this situation; it allows you to graphically specify the form subwindow that you want your tool to have, and it then generates code to produce a "canonical" window with your new form subwindow. Thus, you just specify the commands and fields that you want your form subwindow to have, and let the FormSWLayoutTool generate the code.

The FormSWLayoutTool window has three subwindows: a message subwindow, a form subwindow, and a file subwindow. Figure 16.1 is an illustration of this window.

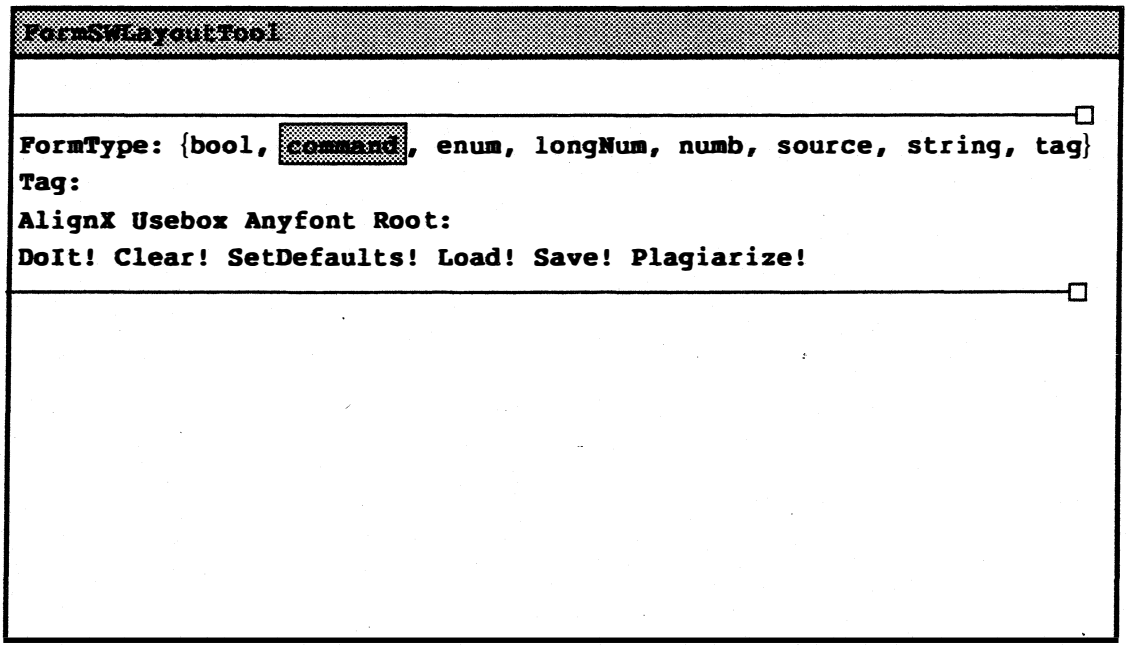


Figure 16.1 The FormSWLayoutTool

Basically, you use the FormSWLayoutTool commands to "draw" the layout of a form subwindow in the file subwindow. When you have laid out a subwindow that you are satisfied with, you can ask the FormSWLayoutTool to generate code. It will generate a "standard" window with three subwindows: a message subwindow, a form subwindow, and a file subwindow. The window generated by the FormSWLayoutTool always has these three standard subwindows; the only thing you specify is the format of the form subwindow. However, once the code has been generated, you can edit the code to add or remove subwindows, or reorder the existing ones. (We will discuss how to do this in the next chapter.)

16.3.1 Plagiarize

There are two ways to put a form item on your new form subwindow: you can add them individually to the file subwindow or you can "plagiarize" from another form subwindow on your screen. To plagiarize, you invoke the **Plagiarize!** command, and then click Point over the form subwindow that you wish to copy. (The cursor will change into an "eyeball" while you are in plagiarize mode.) When you click Point over the subwindow that you want to plagiarize, a copy of that subwindow will appear in the bottom subwindow of the FormSWLayoutTool.

Once you have plagiarized a subwindow, you can edit the plagiarized copy using the **DELETE**, **MOVE**, **STOP**, and **UNDO** keys. **MOVE** lets you move a selected form item around the file subwindow; **DELETE** deletes a selected item. **UNDO** brings back the last form item that you deleted; **STOP** lets you abort in the middle of a **MOVE** command. Thus, you can use **Plagiarize!** to copy an existing form subwindow, and then use the function keys to modify the plagiarized form.

16.3.2 Layout mode

You can also create form items "from scratch". The **FormType:** item in the FormSWLayoutTool command subwindow is an enumerated form item that has as its choices all possible types of form items; that is, every type of item that you can have in a form subwindow. To add a form item to your new form subwindow, you first select the type of item that you want from this enumeration, and then you enter a tag for it in the **Tag:** field. Whenever you have a value in the **Tag:** field, you are in *layout mode*. While you are in layout mode, moving the cursor into the bottom subwindow will cause the cursor to change into a copy of the tag to be added. To add your item, just click Point at the desired location; a new item will be added, of the type specified in the **FormType:** field, and with the tag specified in the **Tag:** field.

For example, suppose that you want to write the Story tool, a tool that generates short stories (or novelettes or novels). You want the user to be able to control the names and personalities of the characters, so you decide that you need a **CharacterName:** string field to contain the name, a **CreateCharacter!** command that creates a character with that name, and a **SetCharacterProps!** command to specify character attributes. Figure 16.2 is an illustration of what the FormSWLayoutTool would look like while you are creating the **CharacterName:** field. (Note that you don't have to include the punctuation that should follow the item, such as colon or exclamation point, in the Tag: field; the tool deduces the necessary punctuation from the type of the item and adds it automatically.)

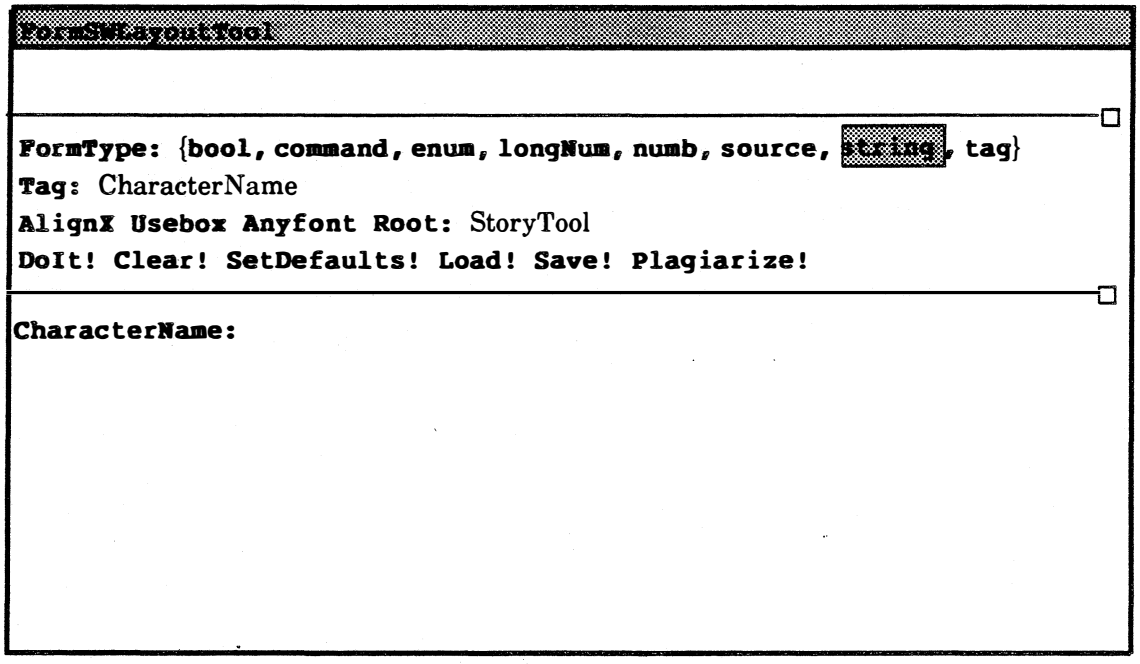


Figure 16.2 Adding the **CharacterName:** field

To get out of layout mode so that you can edit your new form using the DELETE, MOVE, STOP, and UNDO keys, you will have to delete the value in the **Tag:** field.

16.3.2.1 Enumerated items

All form items have an associated properties sheet, but in most cases you don't have to change the properties of an item. When you create an enumerated item, however, you have to provide the values that you would like to have as choices in your enumeration. To do this, select your enumerated item in the bottom subwindow, and press the CONTROL key. This will bring up a properties window. **Choices:** is used to list the values that you want as possible choices in your enumerated type. Individual entries should be separated by spaces; you can include spaces in your entry by quoting them. You can also use the properties sheet to specify whether you want all the choices to be displayed, or just one. Figure 16.3 is an illustration of the nearly-complete layout of the Story tool. The properties sheet is open, and the **Choices:** item reflects the possible choices of plot type. (Some of the entries in this properties window, such as EnumName, represent variable names and other parameters in the actual code. You will learn about the other items in the properties sheet in the next chapter.)

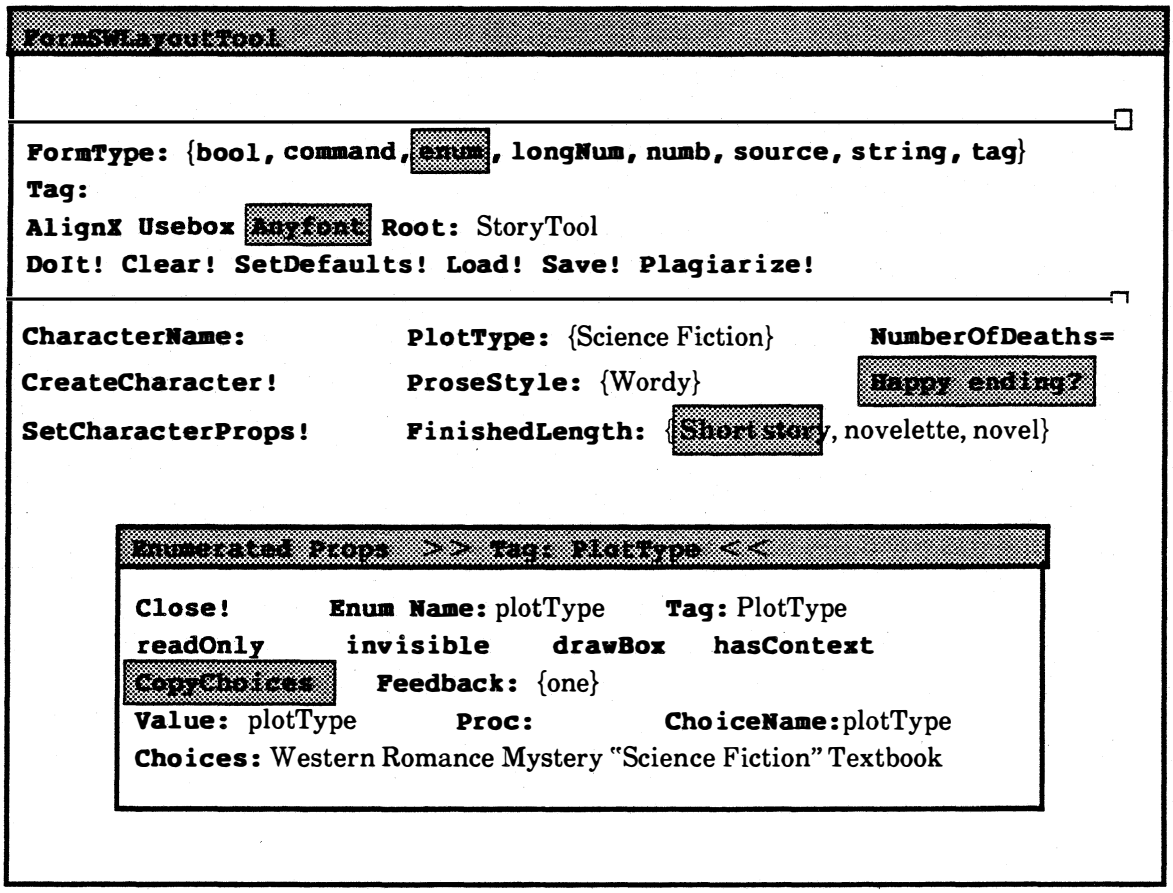


Figure 16.3 Setting the choices for an enumeration

16.3.3 The SetDefaults Command

The **SetDefaults** command brings up a property sheet that allows you to specify defaults for various things associated with the FormSWLayoutTool. For example, you can specify that the default setting for an enumerated should be **all** instead of **one**, or you can specify various characteristics of your string types. You should take a look at this property sheet to get an idea of the kinds of things that you can change. Some of the defaults refer to things that we won't discuss until the next chapter, so don't worry abut it if you don't understand what everything on the sheet refers to. For now, you only need to know what the **SetDefaults** command is good for.

16.3.4 FormSWLayoutTool booleans

The FormSWLayoutTool form subwindow also provides the booleans **AlignX, Usebox**, and **Anyfont. AlignX** controls the vertical spacing between form items. When **AlignX** is on, each column will start on multiples of a specific distance from the previous column. (This distance is defined to be the width of the character O.) **Usebox** specifies that the generated tool will have the same window box as the current size of the layout tool. **Anyfont** will cause the tool to generate code that will have proportioned space on the form subwindow regardless of the system font being used.

16.3.5 Generating the tool

When you have finished laying out your form subwindow, you can use the **Doit!** command to generate code. You should enter the name that you want your tool to have in the **Root:** field. **Doit!** will then generate a file called *Root*.mesa; this file will contain the code necessary to create a tool with a message subwindow, your form subwindow, and a file subwindow. (The value in the **Root:** field will be the name of the program, file, and tool that is generated; in the example, the root is Story, so the code will be in the file Story.mesa.) The code that the FormSWLayoutTool generates will compile successfully. If you then run **Story.bcd**, the tool window interface will appear on your screen (but the commands obviously won't do anything.)

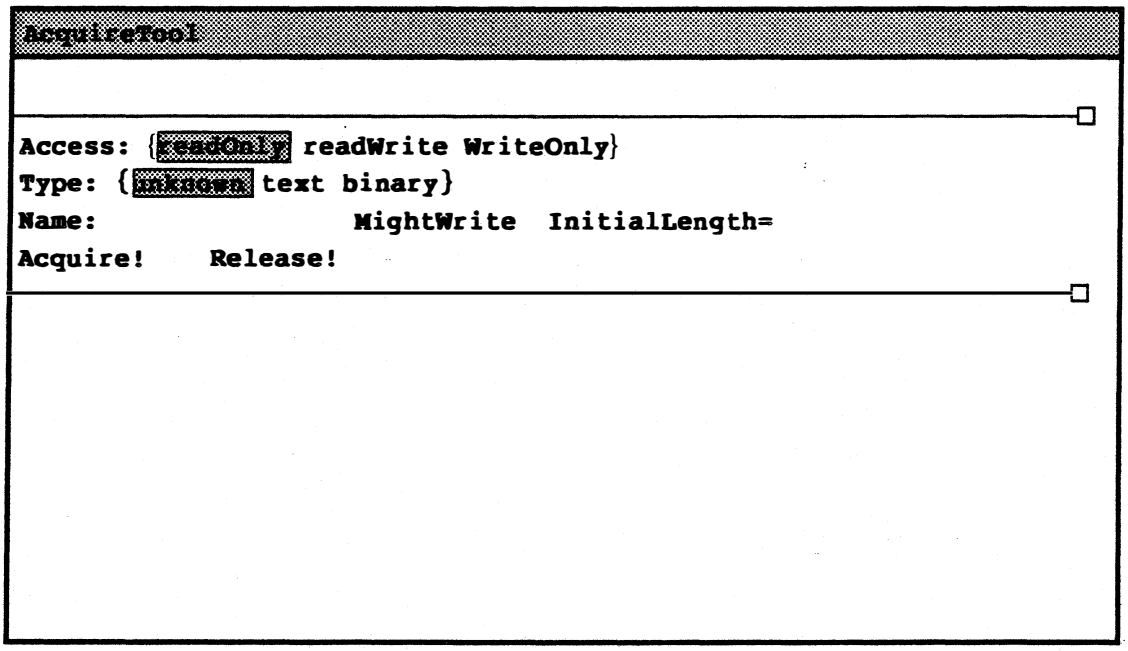
You can also save an unfinished form subwindow in an intermediate format with the **Save!** command. This generates a file with the extension **.by**; you can later use the FormSWLayoutTool's **Load!** command to load this intermediate state into the tool and continue editing. In general, it is a good idea to use the **Save!** command occasionally while you are working on a complex form subwindow. You will then have the **.by** file as a backup.

16.5 Summary

The FormSWLayoutTool makes it easy for you to create a tool window interface; you can use this tool without any knowledge of how windows are created. You simply "draw" a form subwindow and let the FormSWLayoutTool generate code for your window. You need only write the code to implement your commands, and you will have a functional tool.

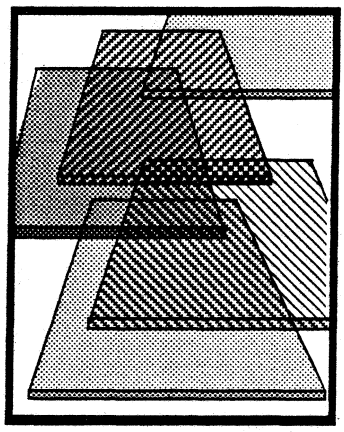
16.6 Exercise

The exercise for this chapter is to use the FormSWLayoutTool to create a tool that looks like the one shown below. Save the generated code because you will need it for the next chapter.



The Acquire Tool

Notes:



Tool window interfaces

The last chapter introduced the FormSWLayoutTool as a aid for generating your own applications, but did not discuss any of the details of how windows are created. This chapter picks up where the last left off: it explicates the code produced by the FormSWLayoutTool. When you are through with this chapter you should understand windows well enough to be able to modify the code produced by the FormSWLayoutTool or to write your own window code if you want more flexibility than the FormSWLayoutTool offers.

The next chapter will expand further on tool building by providing some details of how tools are integrated into the Tajo environment.

17.1 Discussion

In this chapter, we use as an example the code produced by the FormSWLayoutTool for a tool that has the same form subwindow as Command Central. This file is stored on the course directory as CommandCentral2.mesa; you can also generate your own with the FormSWLayoutTool if you like.

The .mesa files generated by the FormSWLayoutTool typically consist of some declarations, followed by procedure templates, followed by four procedures that do the window creation. The procedure templates correspond to the command items in your form subwindow. In the last chapter, you used a template generated by the FormSWLayoutTool and provided the code for these procedure templates. In this chapter, we will look at the rest of the code generated by this tool.

17.1.1 The data

The FormSWLayoutTool allocates its data in a **MACHINE DEPENDENT RECORD**. The example below shows the data that would be declared for creating the Command Central tool window:

```
DataHandle: TYPE = LONG POINTER TO Data;
Data: TYPE = MACHINE DEPENDENT RECORD [
  msgSW(0): Window.Handle ← NIL,
  formSW(2): Window.Handle ← NIL,
  fileSW(4): Window.Handle ← NIL,
  compile(6): LONG STRING ← NIL,
  bind(8): LONG STRING ← NIL,
  run(10): LONG STRING ← NIL,
  log(12): UNSPECIFIED ← 0];

data: DataHandle;
```

The data is stored in a machine dependent record so that the fields will be aligned at word boundaries. Word alignment is necessary because the code will later need to generate addresses for the locations of enumerated and boolean items. There is a **Window.Handle** for each subwindow of the tool window; if you want to add, remove, or rearrange the subwindows of your tool, you should edit this record accordingly.

(If you look at the declaration of **Window.Handle** in the *Mesa Programmer's Manual*, you will discover that this is a pointer to a **Window.Object**, which is an *opaque type*. The declaration of **Window.Object** gives the size of the type, but does not give any information about its structure. This makes the internal structure of the type invisible.)

The last four items in this record contain the storage for the strings that the user enters in the **Compile**:, **Bind**:, **Run**:, and **Log**: fields.

17.1.2 The call to Tool.Create

The actual window creation is done with a call to the **Create** procedure in the **Tool** interface. This procedure is declared as:

```
Tool.Create: PROCEDURE [
  name: LONG STRING,
  makeSWsProc: Tool. MakeSWsProc,
  initialState: Tool.State ← default,
  clientTransition: ToolWindow.TransitionProcType ← NIL,
  movableBoundaries: BOOLEAN ← TRUE,
  initialBox: Window.box ← ToolWindow.nullBox,
  cmSection, tinyName1, tinyName2: LONG STRING ←NIL,
  named: BOOLEAN ← TRUE,
  RETURNS [window: Window.Handle];
```

In the FormSWLayoutTool code, the call to **Tool.Create** is found in the procedure **Init**, as in:

```
Init: PROCEDURE = {
  wh ← Tool.Create[
    makeSWsProc: MakeSWs,
    initialState: default,
    clientTransition: ClientTransition,
    name: "CommandCentral"L,
    cmSection: "CommandCentral"L]};
```

name is the name that you specified in the **Root:** field of the FormSWLayoutTool form subwindow; this parameter is displayed in the herald of the tool if the **named** parameter is **TRUE** (which it is by default). **cmSection** specifies the name of the user.cm section that the tool will look at to set default parameters. **initialState** can be any of the three window states, or it can be default, as in this case. The value **default** specifies that the tool assumes its state depending on how it is created. For example, if the tool was created because the user ran it from the Executive, Tajo assumes that the user would like the tool to be active. If it is run from an initial command line in a user.cm, however, it will be loaded inactive.

The **movableBoundaries** and **initialBox** parameters are defaulted in this call to **Tool.Create**. **movableBoundaries** determines whether or not the user can move the boundary lines separating subwindows; this parameter is defaulted to **TRUE** and is almost always left that way. The **initialBox** parameter can be used to specify the tool box that the tool will initially occupy. (For example, you can set this parameter with the **UseBox:** field in the FormSWLayoutTool.) The value of **ToolWindow.NullBox** specifies that the window box will be allocated by the normal Tajo window box allocator.

The remaining two parameters of **Tool.Create**, both of which are procedures, are described below, in sections 17.1.3 and 17.1.4.

17.1.3 Subwindows

The functionality of a window is determined by the subwindows of which it is composed; each of the various subwindow types has a specific function. The Tajo facilities provide very strong support for using these existing subwindow types; thus, when you create a window interface you don't have to worry about basic window facilities such as the scrollbar and window herald. Instead, you only need to specify the number and type of subwindows that you would like your tool to have. Thus, the heart of your window code is a procedure of type **Tool.MakeSWsProc** that specifies the subwindow layout of your window. In our example, this procedure is called **MakeSWs**. For example:

```
MakeSWs: Tool.MakeSWsProc = {
  logName: LONG STRING ← [10];
  Tool.UnusedLogName[unused: logName, root: "CommandCentral.log"L];
  data.msgSW ← Tool.MakeMsgSW[window: Window.Handle];
  data.formSW ← Tool.MakeFormSW[
    window: Window.Handle, formProc: MakeForm];
  data.fileSW ← Tool.MakeFileSW[window: Window.Handle, name: logName];
  };
```

This procedure is of type **Tool.MakeSWsProc**, which takes one argument, a window handle.

The window creation code makes extensive use of Tajo's *call-back procedures* to implement the design principle of "*Don't call us, we'll call you.*" When you run this "Command Central" code, the procedure **Init** will be called from the mainline code. **Init** then calls **Tool.Create**, passing in the **MakeSWs** proc, which describes the desired subwindow layout. The Tajo facilities then have a procedure that it can call whenever it needs to create the window for the Command Central interface; the client simply passes the **MakeSWs** procedure to **Tool.Create** and lets Tajo and the **Tool** facilities decide when the **MakeSWs** procedure should be called. For example, Tajo will call back to your **MakeSWs** procedure

each time that the window is re-activated by the user. This ensures that Tajo is in control, rather than an individual client program.

Within the **MakeSWs** procedure, the **Tool.UnusedLogName** procedure guarantees unique log file names among file and TTY subwindows by enumerating all file and TTY subwindows and checking that the name is not in use. Each individual subwindow is created by a call to the appropriate procedure in the **Tool** interface.

The calls to **Tool.MakeMsgSW** and **Tool.MakeFileSW** are straightforward. **Tool.MakeMsgSW** requires only the window handle as a parameter; **Tool.MakeFileSW** requires a window handle and a name, which indicates the name of the file that is to be used for the backing store. There are other possible parameters for each of these calls, which are assigned default values in the type declaration. You should take a look at the declarations of **MakeMsgSw** and **MakeFileSW** in the **Tool** interface so that you have some idea of the other parameters that are available. In most cases, however, you can just default these additional parameters.

When a tool has a form subwindow, things are a little more complex; you must also provide a procedure of type **FormSW.ClientItemsProcType** to set up the items in the form subwindow. This procedure is another example of a call-back procedure: the client passes the description of the desired form subwindow to the **FormSW** interface, which is responsible for actually creating that form subwindow. Our Command Central "tool" has the following example of this kind of procedure:

```
FormItems: TYPE = {expand, compile, bind, run, go , options, compile,
   bind, run, log};

MakeForm: FormSW.ClientItemsProcType = {
    OPEN FormSW;
    nItems: CARDINAL = FormItems.LAST.ORD + 1;
    log: ARRAY[0..2) OF Enumerated ← [
      ["Compiler"L, 0], ["Binder"L, 1]];
    items ← AllocateItemDescriptor[nItems];
    items[FormItems.expand.ORD] ← CommandItem[
      tag: "Expand"L, place: [0, line0], proc: Expand];
    ...
    items[FormItems.options.ORD] ← CommandItem[
      tag: "Options"L, place: [294, line0], proc: Options];
    ...
    items[FormItems.compile.ORD] ← StringItem[
      tag: "Compile"L, place: [0, line1], inHeap: TRUE, string: @data.compile];
    ...
    items[FormItems.log.ORD] ← EnumeratedItem [
      tag: "Log"L, place: [0, line4], choices: DESCRIPTOR[log], value:
    @data.log];
    RETURN[items: items, freeDesc: TRUE];
    };
```

A procedure of type **FormSW.ClientItemsProcType** returns an array descriptor; each element within the array is a record describing one of the items in the form subwindow. The call to **FormSW.AllocateItemDescriptor** allocates an item descriptor for **nItems**. (It is important to allocate your item descriptor through the **FormSW** interface so that Tajo handles the

automatic allocation and deallocation of this storage during state transitions. If you don't use the standard system routines and data types to create your subwindows, you will have to explicitly allocate and deallocate that storage.)

The complete description for each type of item is given in a **FormSW.ItemObject**, which is a variant record with a different arm for each possible type of form item.

Two common fields of this variant record are **tag** and a **place**; every form item, regardless of its type, has a **tag** and a **place**. A **tag** is a **LONG STRING** that you supply to be used as the name of the item; this is the value in the **Tag:** field of the FormSWLayoutTool. **place** has two integer fields, **x** and **y**. The **x** field specifies the number of bits that an item is shifted to the right, starting from 0 at the left edge of the window. The **y** field specifies the number of lines that an item is shifted down from the top of the subwindow. [0, line 0] places an item in the top left corner of the subwindow.

There is a third common field in the **FormSW.ItemObject** variant record which is assigned default values in the type declaration, and is omitted in our example. We include it here for the sake of completeness; you will not often have to change the default values for this field. This field is called **flags**, and is of type **FormSW.ItemFlags**, which is declared as :

```
FormSW.ItemFlags: TYPE = RECORD [
  readOnly: BOOLEAN ← FALSE,
  invisible: BOOLEAN ← FALSE,
  drawBox: BOOLEAN ← FALSE,
  hasContext: BOOLEAN ← FALSE,
  clientOwnsItem: BOOLEAN ← FALSE,
  modified: BOOLEAN ← FALSE];
```

This record maintains state bits for an item in a form subwindow. The fields in this record are the parameters that you see when you invoke propertires on an item in the FormSWLayoutTool window. See the declaration of **FormSW.ItemFlags** in the **FormSW** chapter of the *Mesa Programmer's Manual* for an explanation of the fields in this record.

In addition to these three common fields, a **FormSW.ItemObject** has a variant arm for each type of form item that contains various pieces of information specific to that type of form item. Many of these parameters have default values and should be ignored for now; an **ItemObject** is a complex structure that allows flexibility when necessary but much of its flexibility is used only in special cases. However, you should take a look at the declaration of **FormSW.ItemObject** in the *Mesa Programmer's Manual* so that you have an idea of the kinds of parameters that are available for the various types of items.

This example illustrates three types of form objects: command, enumerated, and string, each of which is discussed below.

17.1.3.1 Command items

Command items have only one extra piece of information: a procedure (of type **FormSW.ProcType**) that is to be called when the command is invoked. The FormSWLayoutTool generates a "template" for each such procedure. For example:

```
Expand: FormSW.ProcType = {
  Put.Line[data.fileSW, "Expand called"L]};
```

This procedure does nothing other than output a comment telling the user that the command has been called. You are responsible for writing the actual code for this procedure. Tajo will call this procedure when the user invokes the **Expand!** command.

17.1.3.2 String items

String items can have several other parameters. The FormSWLayoutTool, however, generates a simple **StringItem** that has only two parameters: **inHeap** and **string**. When **inHeap** is **TRUE**, the backing string will be automatically allocated and deallocated (by a procedure in the **FormSW** interface) when necessary. **string** is a **LONG POINTER TO LONG STRING** that is used as the backing store for the characters entered by the user.

There are several other possible parameters for a string item; you should check the definition of a **FormSW.ItemObject** to get an idea of the other options that are available to you.

17.1.3.3 Enumerated items

The declaration of a **FormSW.ItemObject** has the following arm for enumerated items:

```
enumerated => [
  feedback: FormSW.EnumeratedFeedback,
  copyChoices: BOOLEAN,
  value: LONG POINTER TO UNSPECIFIED,
  proc: FormSW.EnumeratedNotifyProcType,
  choices: FormSW.EnumeratedDescriptor]
```

feedback determines how the choices for the enumeration are displayed; the choices are **one** and **all**. (These choices correspond to those in the options sheet of the FormSWLayoutTool.) This option is not illustrated in the above example.

The items in **choice** are the items that you entered in the properties sheet of the FormSWLayoutTool; they are the possible values that the enumeration can assume. When a value is selected, that value is stored in the location pointed to by **value**. **value** points to an **UNSPECIFIED** so that its possible values can be of any type.

proc is a procedure that is called whenever the user changes **value**. This procedure is of type **FormSW.EnumeratedNotifyProcType**, which is declared as follows:

```
FormSW.EnumeratedNotifyProcType: TYPE = PROCEDURE [
  sw: Window.Handle ← NIL,
  item: FormSW.ItemHandle ← NIL,
  index: CARDINAL ← FormSW.nullIndex,
  oldValue: UNSPECIFIED ← FormSW.nullEnumeratedValue];
```

sw is the subwindow containing the item; **item** is the **ItemHandle** of the enumerated item; **index** is the index of the item in the **ItemDescriptor** for the subwindow; **oldValue** is the value of the emumerated item before it was changed by the user. The example above does

not provide an example of such a proc; you can always write one if you find that you need one.

17.1.4 Window state transitions

A tool can be in one of three states: inactive, tiny, and active. Changes in state are usually made at the request of the user, via the commands on the window manager menu. These state changes should be accompanied by an associated change in resources: when a tool is deactivated, it should free all its resources; when a tool is tiny, it needs most of the resources that it requires when active, but should free any resources used exclusively for window display.

Tajo provides the window management for these transitions; that is, it allocates and deallocates the resources needed for standard windows and menus. However, you as the tool writer are responsible for managing any other resources that your tool creates (for example, closing any open files and deallocating any other private data) by writing a procedure that is called each time the window state is changed. This procedure is then passed as one of the parameters to **Tool.Create** (see section 17.1.2). The code generated by the FormSWLayoutTool includes such a transition procedure, which you should modify and expand as you write your actual tool. The **ClientTransition** procedure generated by the FormSWLayoutTool looks like this:

```
ClientTransition: ToolWindow.TransitionProcType = {
  SELECT TRUE FROM
    old = inactive = >
      IF data = NIL THEN data ← zone.NEW[Data ← []];
    new = inactive = >
      IF data # NIL THEN {
          zone.FREE[@data]};
    ENDCASE;
};
```

data is a **DataHandle** (see section 17.1.1 above.) The relevant declarations from the **ToolWindow** interface are:

```
ToolWindow.TransitionProcType: TYPE = PROC[
  window: Window.Handle, old, new:ToolWindow.State];

ToolWindow.State: TYPE = {inactive, tiny, active}
```

This **ClientTransition** procedure is yet another example of a call back procedure. In this procedure, you write the code that you want to have executed when the window state changes. You then pass your transition procedure to Tajo via **Tool.Create**; Tajo will then call your transition procedure when a state transition is about to occur.

17.2 Summary

A window is created with a call to **Tool.Create**. This procedure takes two primary parameters: a procedure of type **Tool.MakeSWsProc** that describes the subwindow structure of the tool; and a procedure of type **ToolWindow.TransitionProcType** that allocates and deallocates resources as the tool is activated and deactivated. The main part of the

MakeSWsProc is a procedure of type **FormSW.ClientItemsProcType** that describes the format of the form subwindow.

These procedures are passed to Tajo via the **Tool.Create** procedure; Tajo is then responsible for calling those procedures when it is time to create the window or to change the window state.

The FormSWLayoutTool generates code that has simple examples of these procedures; you need not change the FormSWLayoutTool code at all. However, if you want additional flexibility, you can modify the code that the FormSWLayoutTool produces by adding additional parameters or subwindow handles; you can also write your own window creation code if you like.

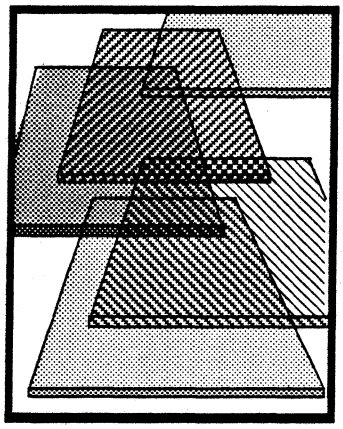
17.3 Exercise

As an exercise, you will implement the two commands **Acquire** and **Release** in the AcquireTool from the last chapter. The **Acquire** command should acquire the file with **MFile.Acquire**, using the parameters that the user enters into the tool's fields. The **Release** command should simply perform an **MFile.Release** on the previously acquired file. You should catch errors when you access the file since you as a programmer do not have control over the user's input to the tool. The solution for this exercise is stored on AcquireTool.mesa and LayoutAcquire.mesa.

Note: To output text to a subwindow (such as a message subwindow or file subwindow), you should use procedures defined in the **Put** interface. (This interface is documented in the *Mesa Programmer's Manual.*) The most commonly used procedures from this interface are **Put.Char**, **Put.Line**, and **Put.Text**, which are declared as follows:

```
Put.Char: PROCEDURE [h: Window.Handle ← NIL, char: CHARACTER];

Put.Line: PROCEDURE [h: Window.Handle ← NIL, s: LONG STRING];

Put.Text: PROCEDURE [h: Window.Handle ← NIL, s: LONG STRING];
```

18



Tool building

In the last few chapters, you have written tools that run from the Executive and created window interfaces using the FormSWLayoutTool. This chapter will complete your introduction to basic tool building in the XDE. We will use the Example Tool to illustrate how to create a tool that is fully integrated with the Tajo environment.

The Example Tool is on the release directory. Retrieve this tool, and run it to familiarize yourself with its interface. The Example Tool does not execute any useful commands; it is merely a sample of how tools are written in the XDE. Figure 18.1 is an illustration of the Example Tool.

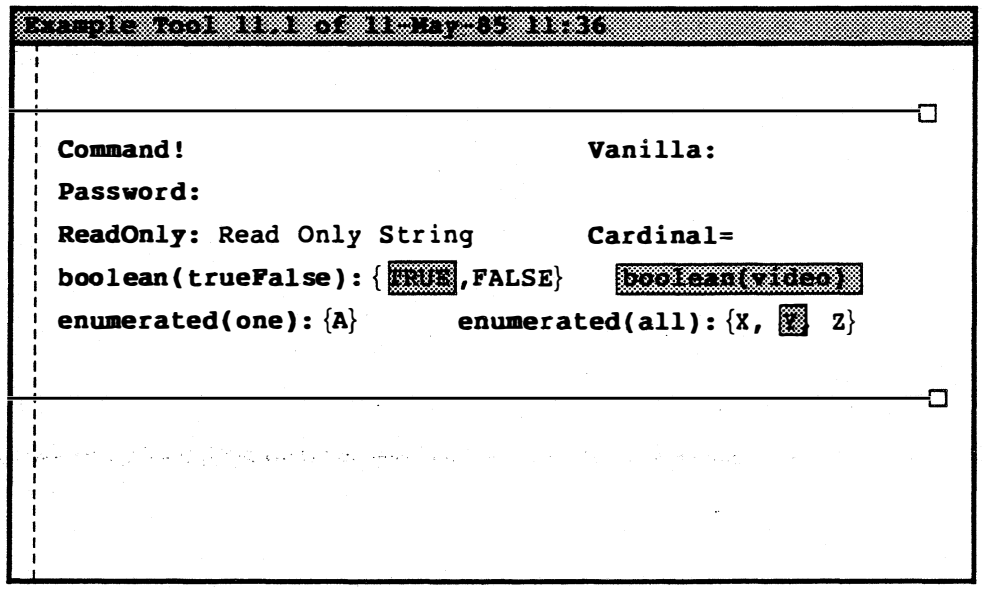


Figure 18.1 The Example Tool

18.1 Discussion

This chapter is divided into four sections: how to read the user.cm file, how to associate pop-up menus with your tool, how to register your tool with the tool driver, and how to use the Supervisor facility.

18.1.1 Reading the user.cm

When you write a tool, you can include a procedure that reads a section in the user.cm to determine initial values for the tool. To do this, you will need to write a procedure called **ProcessUserDotCM** or **ProcessUserCM**, or the like, which you call from your transition procedure to read and process the user.cm each time that the tool is activated.

The Example Tool has the following procedure and associated declarations:

```
Enum1Options: TYPE = {A, B, C};
Enum2Options: TYPE = {X, Y, Z};

ProcessUserDotCM: PROCEDURE =
  BEGIN
  CMOption: TYPE = {EnumOne, EnumAll};
  cmOptionTable: ARRAY [0..1] OF LONG STRING ← ["EnumOne"L, "EnumAll"L];
  cmIndex: CMOption;
  index: CARDINAL;

  cmFile: CmFile.Handle ← CmFile.UserDotCmOpen[
    ! CmFile.Error = > IF code = fileNotFound THEN GOTO return];
  IF CmFile.FindSection[cmFile, "ExampleTool"L] THEN
    DO
      index ← CmFile.NextValue[
          h: cmFile, table: DESCRIPTOR[cmOptionTable] !
          CmFile.TableError = > CONTINUE]
      IF [index = CmFile.noMatch) THEN EXIT

      ELSE
      SELECT (cmIndex ← VAL[index]) FROM
        EnumOne = >
          BEGIN
          enum1Table: ARRAY [0..2] OF LONG STRING ← ["A"L, "B"L, "C"L];
          --note that this is case sensitive
          e1Index: CARDINAL;
          value: LONG STRING = Token.Item[cmFile];
          e1Index ← StringLookUp.InTable[
              key:value, table: DESCRIPTOR[enum1Table], caseFold: FALSE,
              noAbbreviation :TRUE];
          IF e1Index # StringLookUp.noMatch THEN
              toolData.enum1 ← VAL[e1Index];
          [] ← Token.FreeTokenString[value];
          END;
```

```
        EnumAll => 
            BEGIN
            enum2Table: ARRAY [0..2] OF LONG STRING ← ["X"L, "Y"L, "Z"L];
            e2Index: CARDINAL;
            value: LONG STRING = Token.Item[cmFile];
            e2Index ← StringLookUp.InTable[
                key:value, table: DESCRIPTOR[enum2Table], caseFold: FALSE,
                noAbbreviation :TRUE];
            IF e2Index # StringLookUp.noMatch THEN
                toolData.enum2 ← VAL[e2Index];
            [] ← Token.FreeTokenString[value];
            END;
        ENDCASE;
    ENDLOOP;
[] ← CmFile.Close[cmFile];
EXITS return => NULL;
END;
```

This procedure declares an enumerated type (**CMOption**) that lists all options for which the user can have a user.cm entry. (In this case, we allow user.cm entries for the two enumerateds, called **enumerated(one)** and **enumerated(all)**.) This type is then used to build a table (**cmOptionTable**) of the exact strings that are acceptable user.cm entries (in this case, "EnumOne" and "EnumAll"). This structure is standard; you will have to declare similar types each time that you write a **ProcessUserCM** procedure.

After the types and variable declarations, the first two lines of code in this procedure call **CMFile.UserDotCMOpen** and **CMFile.FindSection** to open the user.cm file and ensure that it has an [ExampleTool] section. If the user.cm is present on the search path, is successfully opened, and has an [ExampleTool] section, you will enter a loop that reads through every entry in that section and processes it.

Within the **DO** loop, there is an **IF** expression that calls **CMFile.NextValue** to read the next value in the section. **NextValue** is passed a descriptor for the option table, and searches for a match in the table. Thus, for example, if the user.cm section had as its first entry **EnumOne: B**, **NextValue** would read **EnumOne** from the file, and check it against the values in **cmOptionTable**. Since there is a match, it returns the index of that match (1); if a match is not found, it will return the special value **cmFile.noMatch**, and the loop will be exited.

Within the **SELECT** statement, the **EnumOne** arm constructs a table of possible values that the user can enter ("A", "B", or "C") and then uses the procedure **Token.Item** to read the user.cm file. **Item** just returns the next token after the entry, where a token is delimited by white space. For example, if the entry was "EnumOne: A", **Item** would return "A". The long string returned by **Item** is then passed to **StringLookUp.InTable** to see if it is a valid value for that option. Since it is, this procedure returns the index of the element, and that index is then converted back into its enumerated value with the **VAL** operator. (The code within the **EnumAll** arm is essentially identical.)

In general, the structure of your user.cm procedure will be very similar to this one. To write a procedure to process a user.cm, you need to construct an option table, check to see if there is a user.cm file on disk and open it, verify that there is a section for your tool, and

then read the values in that section using the procedures in the Token interface. Basically, you can copy the structure from this procedure and alter the code in the **SELECT** statement.

18.1.2 Pop-up menus

You can access the Window Manager menu from virtually every subwindow in the environment; many tools have several other menus available as well. If you want your tool to have any menus other than the Window Manager menu, you will have to create and manage those menus through the **Menu** interface. This interface lets you determine which menus the user will see and the actions that each menu item will perform.

18.1.2.1 Creating a menu

You can create a menu to be associated with your window with a call to **Menu.Make**. This procedure is declared as:

```
Menu.Make: PROCEDURE [
   name: LONG STRING,
   strings: LONG DESCRIPTOR FOR ARRAY OF LONG STRING,
   mcrProc: Menu.MCRType,
   copyStrings: BOOLEAN ← TRUE,
   permanent: BOOLEAN ← FALSE]
RETURNS [Menu.Handle];
```

This procedure makes a menu named **name** that has the elements contained in **strings**. **mcrProc** is the procedure that is called when the user selects one of the items on the menu; this procedure is described more fully in section 18.1.2.3. The **copyStrings** flag indicates whether **strings** should be copied into the system heap. When interfaces exchange resources, clients must be very careful about who is responsible for the resource. Thus, all interfaces involving resources must state explicitly whether ownership of the resource is transferred. For example, if your strings are allocated in a local frame, and therefore will be destroyed when you exit form the procedure, you will want to have **copyStrings** true. **permanent** indicates whether the created object can subsequently be destroyed; you will usually want this to be **FALSE**.

Menu.Make returns a **Menu.Handle**, which is a long pointer to a **Menu.Object**:

```
Menu.Object: TYPE = RECORD [
   permanent: BOOLEAN,
   nInstances: CARDINAL [0..77777B],
   name: LONG STRING,
   items: Menu.Items];

Menu.Items: LONG DESCRIPTOR FOR ARRAY OF Menu.ItemObject;

Menu.ItemObject; TYPE = RECORD [
   keyword: LONG STRING, mcrProc; Menu.MCRType];
```

A **Menu.Object** is the basic data structure of the menu.

18.1.2.2 Instantiations of a menu

Once a menu has been created with a call to **Menu.Make**, you need to call **Menu.Instantiate** to indicate the window, windows, or subwindows with which the menu should be associated. **Menu.Instantiate** is declared as:

```
Menu.Instantiate: PROCEDURE [menu: Menu.Handle, window: window.Handle]
```

An unlimited number of menus may be associated (instantiated) with your tool window or with any of its subwindows. The menu mechanism maintains a ring of menu instances (pointers to associated menus) for each subwindow (if there is at least one associated menu). One of these associated menus is taken to be the "current" menu for that subwindow.

Some menus (such as the Window Manager) need to be available from virtually every subwindow. One way to accomplish this is to create an **Object** for each use, but the primary memory cost of multiple copies of an **Object** is large. This leads to the use of a level of indirection: Tajo never copies a client's **Object**; instead, it always keeps a pointer to that **Object**. It is your responsibility to guarantee that the **Object** is valid as long as Tajo has a pointer to it. You should only **Make** a menu once, but you may **Instantiate** that single menu over as many windows as you like. The **nInstances** field of a **Menu.Object** keeps a count of the number of windows with which the menu is associated. **Object**s are created and destroyed by the menu implementation.

Menus are normally created in the procedure that creates the subwindows for a tool (**MakeSWsProc**). (If you have a complicated menu to set up, you should write a procedure to create the menus, and call it from your **MakeSWs** procedure.) For example:

```
--Example Tool Menu support routines
MenuIndex: TYPE = {postMessage, aCommand, bCommand};
MakeSWs: Tool.MakeSWsProc =
  BEGIN
  menuStrings: ARRAY MenuIndex OF LONG STRING ← [
    postMessage: "Post message"L, aCommand: "A Command"L,
    bCommand: "B Command"L];
  toolData.menu ← Menu.Make[
    name: "Tests"L,
    strings: DESCRIPTOR[menuStrings.BASE, menuStrings.LENGTH],
    mcrProc: MenuCommandRoutine];
  ...
  Menu.Instantiate[toolData.menu, toolData.formSW];
  ...
  END;
```

toolData is a pointer to the **MACHINE DEPENDENT** record that contains the data (window handles, menus, booleans, strings, etc.) for the tool. In this example, two of the parameters to **Menu.Make**, **copyStrings** and **permanent**, are omitted; they have the default values assigned in the type declaration (see section 18.1.2.1 above).

18.1.2.3 Menu command routines

One of the parameters to **Menu.Make** is a procedure of type **Menu.MCRType**. This type is declared as:

```
Menu.MCRType: TYPE = PROCEDURE [
  window: Window.Handle ← NIL, menu: Menu.Handle ← NIL,
  index: CARDINAL ← LAST[CARDINAL]];
```

A *Menu Command Routine (MCR)* is a procedure that is called when the user invokes the associated menu item. **index** indicates which menu item was selected. You can have different MCR procedures for each item on the menu, but clients typically have one MCR per menu, so that they can use one large catch phrase to accomodate common exception conditions. MCR procedures are another example of call-back procedures; you write the MCR for your menu, and pass it to the **Menu** interface, which calls that procedure when the user selects an item on your menu. Here is the MCR used in the Example Tool:

```
MenuIndex: TYPE = {postMessage, aCommand, bCommand};
MenuCommandRoutine: Menu.MCRType =
    BEGIN
    mx: MenuIndex = VAL[index];
    SELECT mx FROM
        postMessage = > Put.Line[toolData.msgSW, "Message posted."L];
        aCommand = > Put.Line[toolData.fileSW, "A Menu command called."L];
        bCommand = > Put.Line[toolData.fileSW, "B Menu command called."L]
    END;
```

18.1.2.4 Freeing a menu

Menus are like any other storage: you should allocate them when you need them and free them when you are through with them. **Menu.Free** is the complement of **Menu.Make**; **Menu.Uninstantiate** is the complement of **Menu.Instantiate**:

```
Menu.Free: PROCEDURE [menu: Menu.Handle, freeStrings: BOOLEAN ← TRUE];

Menu.Uninstantiate: PROCEDURE [menu: Menu.Handle, window: Window.Handle];
```

These procedures should be called from your window state transition procedure. For example:

```
...
    new = inactive = >
        Menu.Uninstantiate[menu: toolData.menu, window: toolData.formSW];
        Menu.Free[toolData.menu];
```

18.1.3 Registering a tool with the Tool Driver

The Tool Driver is a tool that provides a mechanism for automatically performing repetitive, routine tasks in batch mode. The Tool Driver does not automatically have access to every tool, however; you must include code to register your tool with the Tool Driver. Every tool that performs some "generally useful function" should include code to

register itself with the Tool Driver. You should give the user the option of using the Tool driver with your tool.

The **ToolDriver** interface provides primarily two procedures: **NoteSWs** and **RemoveSWs**. These two procedures are used to notify the Tool Driver of the existence of a tool's subwindows and to remove that notification, respectively. The subwindow registration should be done when the subwindows are created, in the **MakeSWs** proc; the removal should be done in the **TransitionProc**, where the window resources are destroyed.

The subwindows for a window are described in an array of type **ToolDriver.Address**, which is declared as follows:

```
ToolDriver.Address: TYPE = RECORD [name: LONG STRING, sw: Window.Handle]
```

For example:

```
MakeSWs: Tool.MakeSWsProc =
  BEGIN
  addresses: ARRAY [0..3) OF ToolDriver.Address;
  ...

  addresses ← [
    [name: "msgSW"L, sw: toolData.msgSW],
    [name: "formSW"L, sw: toolData.formSW],
    [name: "fileSW"L, sw: toolData.fileSW]];
  ToolDriver.NoteSWs[tool: "ExampleTool"L, subwindows: DESCRIPTOR[addresses]]
  ...
END;

ClientTransition: ToolWindow.TransitionProcType =
  BEGIN
  SELECT TRUE FROM
  ...
    new = inactive = >
      BEGIN
      ...
      ToolDriver.RemoveSWs[tool: "ExampleTool"L];
      END;
  ENDCASE;
  END;
```

18.1.4 The Supervisor facility

Supervisor is a Pilot interface that provides a way to broadcast information to a collection of interested clients (within a single processor). This facility lets you register interest in a particular *event* or class of events. When you register interest in an event, the Supervisor will notify you each time that the event occurs or is about to occur. For example, your program might want to be notified when a world swap is about to occur, when a window is about to be deactivated, or when the user's credentials have just changed.

For example, consider the case of a world swap. The client transition procedures discussed in the last chapter do not contain any special provisions for a world swap. If you are

editing a file, for example, the editor should abort the world swap. It cannot do this through the client transition procedure, however; all this procedure can do is free storage, close files, and the like. Thus, the editor registers to be notified by the Supervisor when a world swap is about to occur, and aborts that swap if the user is in the middle of editing a file. Similarly, the File Tool might want to be notified about a world swap so that it can close any connections to servers. The File Tool does not want to abort the world swap; it just wants advance notice so that it can prepare itself. Thus, the Supervisor allows tools to be notified about an event "before it is too late" and to take specific action regarding that event.

The Supervisor models the entire client system as a collection of *subsystems* that depend on some basic resource. A client program can *register a dependency* on any subsystem; that is, it can register itself as a client of a particular subsystem, which means that it directly uses the services of that subsystem. The Supervisor maintains a database that describes dependency relationships among these subsystems, and provides a way to invoke them in clients-first or implementors-first order. Thus, when an event occurs that involves several subsystems, the Supervisor can either notify the clients of that subsystem first, or its implementors, depending on which is the logical direction.

Each subsystem that wants to use the Supervisor facilities should obtain a subsystem handle from the Supervisor and export it to its clients. The clients then use these handles to declare the subsystems on which they depend. A **Supervisor.SubsystemHandle** may be thought of as a class of related events. The **Event** interface in the *Mesa Programmer's Manual* contains **Supervisor.SubsystemHandles** on which a client may add dependencies. A client specifies interest in a particular class of events by registering a dependency on the **Supervisor.SubsystemHandle** obtained from **Event**. The interface **EventTypes** provides the specific **Supervisor.Events** that are raised. A client that has been registered to be notified about a class of events uses the **Supervisor.Event** to determine which element of that class has actually occurred.

Each subsystem also registers an *agent procedure*. When an interesting event happens, the Supervisor is invoked to notify the agent procedures that are interested in that event. This notification can take place in either order (clients-first or implementors-first).

18.1.4.1 Using the Supervisor

To register interest in an event, you would find the event definition in the **EventType** interface and add a dependency (**Supervisor.AddDependency**) on the **Supervisor.SubsystemHandle** in **Event** that corresponds to the event. (An event is defined by a pair of items, one from **Event** and the other from **EventType**.)

Here is an example from the Example Tool that uses the Supervisor to abort deactivation of a tool if the tool is still running:

```
agent: Supervisor.SubsystemHandle =
  Supervisor.CreateSubsystem[CheckDeactivate];

CheckDeactivate: Supervisor.AgentProcedure =
  BEGIN
  IF event = EventTypes.deactivate AND
    wh # NIL AND wh = eventData
    AND toolData.commandIsRunning THEN {
    Put.Line[toolData.msgSW, "The tool is still processing a
      command: aborting deactivation"L];
    ERROR Supervisor.EnumerationAborted};
  END;

-- main code add the event
Supervisor.AddDependency[client: agent, implementor: Event.toolWindow];
```

The signal **Supervisor.EnumerationAborted** refers to the enumeration of subsystems that need to be notified of a particular event. This signal is raised whenever an enumeration is aborted for some reason. (In this case, the Example Tool will abort the deactivation whenever it is in the middle of processing a command, and raise **EnumerationAborted** since there is no need to notify any other subsystems.)

As a second example, consider a world swap. When the user asks to leave CoPilot and world-swap to the client volume, CoPilot will notify on the event **Event.AboutToSwap**. If any tool is unwilling or unable to stop for a world swap, it should abort this event by raising **EnumerationAborted**. If no clients abort the swap, CoPilot will notify on the event **Event.swapping** with a swap-out reason (**EventType.abortSession**, **EventType.resumeDebuggee**, or **EventType.abortSession**). All tools are expected to stop when this event is notified. When CoPilot is re-entered for any reason, it raises the event **Event.swapping** with a swap-in reason (**EventType.newSession** or **EventType.resumeSession**) to let tools know that they can resume processing. Here is an typical example:

```
swapDone: CONDITION;
subsystemRunning, swapping: BOOLEAN ← FALSE;
aboutToSwapAgent: Supervisor.SubsystemHandle =
  Supervisor.CreateSubsystem[agent: AboutToSwap];
swappingAgent: Supervisor.SubsystemHandle =
  Supervisor.CreateSubsystem[agent: Swapping];

StartSubsystem: ENTRY PROCEDURE = {
  IF swapping THEN WAIT swapDone;
  subsystemRunning ← TRUE};

SubsystemStopped: ENTRY PROCEDURE = {subsystemRunning ← FALSE};

AboutToSwap: ENTRY Supervisor.AgentProcedure =
    BEGIN
    ENABLE UNWIND => NULL;
    IF subsystemRunning THEN {
        HeraldWindow.AppendMessage["MyTool busy: aborting swap."L];
        ERROR Supervisor.EnumerationAborted};
    END;
```

```
Swapping: ENTRY Supervisor.AgentProcedure =
    BEGIN
    ENABLE UNWIND => NULL;
    SELECT event FROM
      EventTypes.newSession, EventTypes.resumeSession, EventTypes.swapCancelled,
        EventTypes.bootPhysicalVolumeCancelled => {
          swapping ← FALSE; BROADCAST swapDone};
      EventTypes.abortSession, EventTypes.resumeDebuggee,
        EventTypes.bootPhysicalVolume =>
          swapping ← TRUE;
    ENDCASE;
    END;

-- mainline
Supervisor.AddDependency[client: aboutToSwapAgent, implementor:
Event.aboutToSwap];
Supervisor.AddDependency[
  client: swappingAgent, implementor: Event.swapping];
DO
  SubsystemStopped[];
  -- wait for user input from the Notifier
  StartSubsystem[];
  -- perform computation
  ENDLOOP;
  ...
```

18.1.5 Using the Executive interface

As you have seen, there are two basic styles of program invocation in the XDE: interactive (tool windows) and batch (the Executive). Typically, you provide an interactive interface by creating a form subwindow with command items for each procedure, and provide a batch interface by writing one or more **Exec.ExecProc**s that can be called from the Executive. In general, it is a good idea to make your tool facilities accessible in either style.

By taking some care in the design of your tools, you can support both invocation methods fairly easily. You should provide an interface that defines the function provided by your package. This functional interface can be called directly from programs, making it possible for client programs to use the package directly. You can then write two interface packages that invoke these functions: one package implements an **ExecProc**, and the other implements a tool window.

This means that the functional interface should make no assumptions about where its input comes from or where its output goes. If the package must interact with the user, it must require interface packages for the interaction. It must not assume that it has a window it can communicate through. The package should not assume it knows the location of input parameters. All input should be passed to the package explicitly by the interface packages, even if it is just in the form of a command line that must be parsed. An output procedure should be provided by the caller.

The Example Tool does not define an interface, but it does provide both sorts of interaction. Here are the relevant routines from the Example Tool:

```
Help: Exec.ExecProc =
  BEGIN
  OutputProc: Format.StringProc ← Exec.OutputProc[h];
  OutputProc[
    "This command activates the ExampleTool window. The ExampleTool is an
example of a 'Tool' that runs in Tajo. It demonstrates the use of a comprehensive set of
commonly used Tajo facilities. Specifically we present examples of the definition,
creation, use and destruction of the following:

  Windows and subwindows, Menus, Msg subwindows, Form subwindows and File
subwindows"L];
  END;

Unload: Exec.ExecProc =
  BEGIN
  IF wh # NIL THEN Tool.Destroy [wh];
  wh ← NIL;
  [] ← Exec.RemoveCommand[h, "ExampleTool.~"L];
  END;

InitHeap: PROCEDURE = INLINE
  BEGIN
  heap ← Heap.Create[initial: 1];
  END;

KillHeap: PROCEDURE = INLINE
  BEGIN
  Heap.Delete[heap];
  heap ← NIL;
  END;

Init: PROCEDURE =
  BEGIN
  Exec.AddCommand["ExampleTool.~"L, ExampleToolCommand, Help, Unload];
  END;

MakeHeraldName: PROCEDURE =
  BEGIN
  tempName: LONG STRING ← heap.NEW[StringBody [60]];
  String.AppendString [tempName, "ExampleTool "L];
  Version.Append[tempName];
  String.AppendString[tempName, " of "L];
  Time.Append[tempName, Time.Unpack[Runtime.GetBcdTime []]];
  tempName.length ← tempName.length - 3; -- gun the seconds
  heraldName ← String.CopyToNewString[tempName, heap];
  heap.FREE[@tempName];
  END;
```

```
MakeTool: PROCEDURE RETURNS[wh: Window.Handle] =
  BEGIN
  RETURN[Tool.Create[
  makeSWsProc: MakeSWs, initialState: default,
  clientTransition: ClientTransition, name: heraldName,
  cmSection: "ExampleTool"L, tinyName1: "Example"L, tinyName2: "Tool"L]]
  END;

ExampleToolCommand: Exec.ExecProc =
  BEGIN
  IF heap = NIL THEN InitHeap[];
  IF heraldName = NIL THEN MakeHeraldName[];
  IF (wh # NIL) AND inactive THEN ToolWindow.Activate[wh]
  ELSE IF wh = NIL THEN wh ← MakeTool[];
  END;

--Mainline code
Init[];
END.
```

18.2 References

The material in this chapter makes extensive use of the material in the following chapters of the *Mesa Programmer's Manual*: **CMFile**, **Event**, **EventTypes**, **Menu**, **Token**, **ToolDriver**. The Supervisor material is covered in the also read the Supervisor section in the *Pilot Programmer's Manual*.

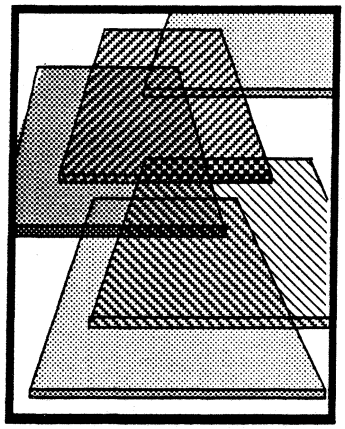
18.3 Exercises

The exercise for this chapter is to write a tool that reads information from the user.cm file and that uses the Supervisor facility. The tool should read a file name (**fileName**), and a file length (**Length**) from the user.cm file whenever the tool changes to an active state. In order to read this information you will need to write a procedure, called from the tool's **TransitionProc**, which uses the **CmFile** procedures.

The tool has only two commands, **AcquireFile** and **ReleaseFile**. **AcquireFile** acquires **fileName** with **readWrite** access and with initial length of **Length**. When the tool has acquired a file, you want to ensure that the user cannot deactivate the tool. To do this, you need to add a dependency to the **toolWindow** event (e.g. **deactivate**). Thus if someone attempts to deactivate the tool after the file is acquired, you should issue a message and abort the deactivation. However, if the user releases the file, the tool may be deactivated.

To do this exercise you should create a tool with the FormSWLayoutTool and add two procedures (**ProcessUserCM** and **CheckDeactivate**). In addition, you will need to make calls to **MFile.ReadWrite** and **MFile.Release** for the file manipulation. The solution for this exercise is located on UserCMtoolsolution.mesa.

19



Multiple instance tools

This chapter is the last of the tool building sequence; it assumes that you are familiar with the material in chapters 16 through 18. In this chapter, we discuss how to design a tool so that the user can have multiple copies of the tool on the screen simultaneously.

19.1 Definition of terms

Context — A *context* is data associated with a window or subwindow; such data has the same lifetime as the window with which it is associated.

Multiple instance tool — A *multiple instance tool* is a tool that allows the user to have more than one copy of the tool window on the screen at a given time. All instances of a multiple instance tool share the same global frame.

Notifier — The *Notifier* is the process that is responsible for processing user actions, such as mouse clicks and keystrokes.

19.2 Discussion

A multiple instance tool is a tool that can have more than one copy ("instance") of its window on the screen at any given time. For example, the Mail Send Tool is a multiple instance tool, but the File Tool is not. (A simplistic way of looking at it is that a multiple instance tool has the commands **Another!** and **Destroy!** in its form subwindow.)

All copies of a particular tool share the same global frame. Thus, there is an obvious problem: what does a multiple instance tool do with the data that it would normally store in its global frame? Data that must be replicated for each copy of the window (such as window and subwindow handles, and form subwindow items) can't reside in the global frame, since the global frame is shared amongst all copies of a tool.

The **Context** interface solves this problem by enabling you to associate data with a window handle rather than storing it in the global frame. Whenever you create a window, you allocate your data and associate that data with the window. You can then retrieve the data each time your tool is called.

19.2.1 Obtaining a context type

When you want to use the **Context** interface, you have to acquire a unique context type for your tool. The basic idea is that every client of the **Context** interface must have its own unique type; this type is how you identify yourself to the interface. To get such a type, you declare a global variable of type **Context.Type** and then call the procedure **Context.UniqueType**. (The context type is the same for all copies of the tool, so putting it in the shared global frame is the right thing to do)

Context.UniqueType returns a **Context.Type**, which is unique for each client of the **Context** interface. (**UniqueType** will raise an error if no more unique types are available.) You only need to call this procedure once, during initialization. For example,:

```
dataType: PUBLIC Context.Type ←Context.UniqueType[];
```

Figure 19.1 illustrates the idea behind contexts. Notice that all the windows have the same context type (they are instances of the same tool), but they each have different data.

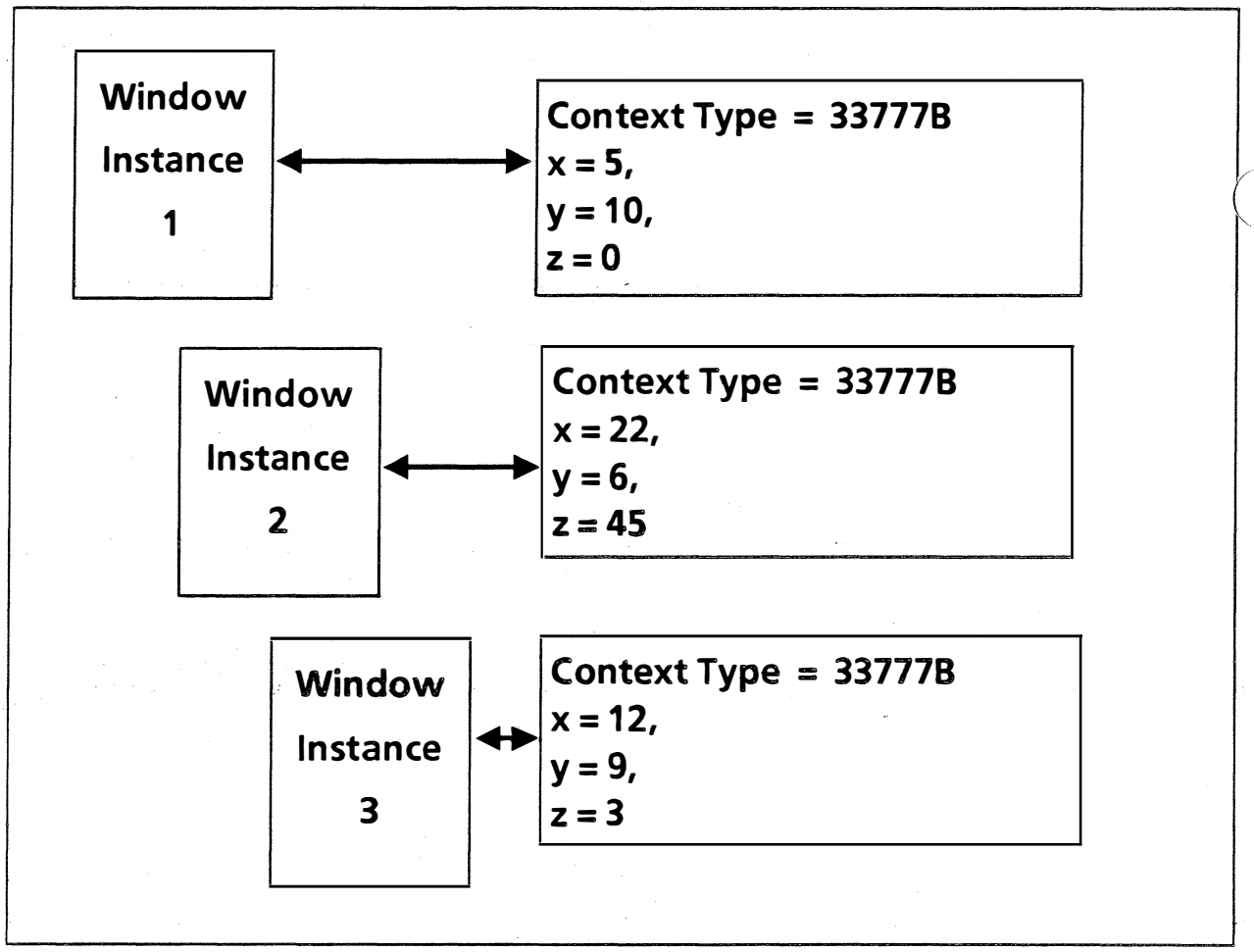


Figure 19.1 Associating contexts with windows.

19.2.2 Creating the context

Before you can use a context, you have to allocate it and "attach" it to the window. Since a context is basically just an alternative to global tool data, you should allocate the context in the same places in your code that you would otherwise have allocated the tool data. To do the allocation you need to call **Context.Create**:

```
Context.Create: PROCEDURE [
  type: Context.Type,
  data: Context.Data,
  proc: Context.DestroyProcType,
  window: Window.Handle];
```

Create creates a context of type **type** that contains **data**. **type** is your unique identification; **data** is a record that you define to specify what you want to store in your context. In this case, **data** is of type **DataHandle**, which is declared as follows:

```
DataHandle: TYPE = LONG POINTER TO Data;
Data: TYPE = MACHINE DEPENDENT RECORD [
  wh(0): Window.Handle ← NIL,          --handle to parent window
  msgSW(2): Window.Handle ← NIL,       --handle to message subwindow
  fileSW(4): Window.Handle ← NIL,
  formSW(6): Window.Handle ← NIL,
  string(8): LONG STRING ← NIL];       --for string in form subwindow
```

The **proc** parameter to **Create** is a call back procedure that you can use to deallocate the context data when the window is destroyed. **window** is the window or subwindow with which the context is to be associated.

Thus, continuing the example in section 19.2.1, the relevant parts of your **ClientTransitionProc** might look like this:

```
ClientTransition: ToolWindow.TransitionProcType =
  BEGIN
  ...--retrieve the context here
  SELECT TRUE FROM
    old = inactive = > BEGIN    --window being created; need to allocate context
        IF toolData = NIL THEN toolData ← heap.NEW[Data ← []];
        toolData.wh ← window;
        Context.Create[dataType, toolData, DestroyProc, window];
        END;
    new = inactive = >    --window being destroyed; need to destroy context
      IF toolData # NIL THEN
        Context.Destroy[type: dataType, window: window];
--Context.Destroy is discussed in section 19.2.4
    ENDCASE
  END;
```

Thus, the above call to **Create** creates a context of type **dataType**, associates that context with the window handle, and stores the information in the data record in that context.

19.2.3 Using the context

Once you have stored data in your context with **Create**, you can call **Context.Find** to retrieve that data. Basically, you have to call **Find** each time that you need to reference your data. This procedure is declared as:

```
Context.Find: PROCEDURE [type: Context.Type, window: Window.Handle]
  RETURNS [Context.Data];
```

Find retrieves the **data** field from the specified context. **Find** will return **NIL** if no such context exists on the window.

One example of when you need to use **Find** is when you create or reference the items in a form subwindow. In section 19.2.2, notice that the context includes a **string** parameter, which contains the value of a string in the form subwindow. Thus, when you create your form subwindow (in a **MakeForm** procedure) you need to have the context available so that you can specify it as the location where that string is to be stored.

However, to use **Find**, you need to pass the window handle as a parameter. How are you going to get that handle in order to pass it to **Find**? A **MakeForm** procedure is of type **FormSW.ClientItemsProcType**, which is declared as follows:

```
FormSW.ProcType: TYPE = PROCEDURE [
  sw: Window.Handle ← NIL, item: FormSW.ItemHandle ← NIL,
  index: CARDINAL ← FormSW.nullIndex];
```

Thus, when your **MakeForm** procedure is called, you are passed in a handle to the form subwindow as a parameter. Unfortunately, in order to retrieve the context, you need a handle to the parent window, and not just to the form subwindow. However, since you have the subwindow handle, you can make a call to the procedure **ToolWindow.WindowForSubwindow**: when you pass it the subwindow handle, it will return the handle for the parent window. Thus, the first few lines of your **MakeForm** procedure might look like this:

```
MakeForm: FormSW.ClientItemsProcType =
BEGIN
  OPEN FormSW;
  toolData: DataHandle ←
      Context.Find[dataType, ToolWindow.WindowForSubwindow[sw]];
```

Thus, you have retrieved your context into the variable **toolData**, and you can store to it or retrieve from it.

19.2.4 Destroying the context

The inverse of **Context.Create** is **Context.Destroy**, which you use to destroy a context of a given type on a given window. Again, you want to call **Context.Destroy** in the same places that you would normally free the **toolData** record. If the context exists, this procedure will call the **Context.DestroyProc** that you passed as a parameter to **Context.Create**. Here is the declaration of **Context.Destroy**:

```
Context.Destroy: PROCEDURE [
  type: Context.Type,
  window: Window.Handle];
```

Here is an example of what a **DestroyProc** might look like:

```
DestroyProc: PROC[data: DataHandle, window: Window.Handle] =
  BEGIN
  heap.FREE[@data];
  END;
```

19.3 Summary

In general, you should always make your tools multiple instance tools. To do so, you need to store replicated global data in a *context* rather than in the global frame. A context is essentially data that is associated with a window handle, and thus has the same life as the tool window. Since the global frame is shared among all copies of the tool, you should only use the global frame for data that is to be shared by all copies of the tool.

To use the context interface, you need to obtain a unique type (**Context.UniqueType**), create the context (**Context.Create**), use it (**Context.Find**), and destroy it (**Context.Destroy**). In general, you need to call **UniqueType** each time the *tool* is loaded, call **Create** and **Destroy** each time a *window* is created or destroyed, and call **Find** each time you need to access the data.

Here are the relevant portions of a basic tool that uses the **Context** interface:

```
-- File: MultiTool.mesa - last edit 27-Dec-84 15:59:00
-- Copyright (C) Xerox Corporation 1984. All rights reserved.

DIRECTORY
  Context USING [Create, Destroy, Find, Type, UniqueType],

MultiTool: PROGRAM
  IMPORTS Context, Exec, FormSW, Heap, Put, Tool, ToolWindow =
  BEGIN

  -- TYPES
  DataHandle: TYPE = LONG POINTER TO Data;
  Data: TYPE = MACHINE DEPENDENT RECORD [
    wh(0): Window.Handle ← NIL,
    msgSW(2): Window.Handle ← NIL,
    fileSW(4): Window.Handle ← NIL,
    formSW(6): Window.Handle ← NIL,
    string(8): LONG STRING ← NIL];

  -- create unique context for this tool and store it in global (shared) variable
  dataType: PUBLIC Context.Type ← Context.UniqueType[];

--This procedure is called when the window state is about to change
  ClientTransition: ToolWindow.TransitionProcType =
    BEGIN
    toolData: DataHandle ← Context.Find[type: dataType, window: window];
    SELECT TRUE FROM   --allocate context and associate it with the window
      old = inactive = > BEGIN
          IF toolData = NIL THEN toolData ← heap.NEW[Data ← []];
          toolData.wh ← window;
          Context.Create[dataType, toolData, DestroyProc, window];
          END;
      new = inactive = >    --deallocate context
        IF toolData # NIL THEN
        Context.Destroy[type: dataType, window: window];
      ENDCASE
    END;

--This call-back procedure deallocates the context
  DestroyProc: PROC[data: DataHandle, window: Window.Handle] =
    BEGIN
    heap.FREE[@data];
    END;
```

```
<<This procedure is called when a command in the form subwindow is invoked.
You need to retrieve the context so that you can access the information in the form
subwindow.>>
  FormSWCommandRoutine: FormSW.ProcType =
  BEGIN
  toolData: DataHandle ←
    Context.Find[dataType, ToolWindow.WindowForSubwindow[sw]];
  SELECT index FROM
    FormIndex.command.ORD = > CommandRoutine[toolData];
    FormIndex.another.ORD = > MakeTool[];
    FormIndex.destroy.ORD = > Tool.Destroy[toolData.wh];
  ENDCASE;
  END;

-- standard procedure to create items in the form subwindow
--storage for items in form subwindow found in context
  MakeForm: FormSW.ClientItemsProcType =
    BEGIN
    OPEN FormSW;
    toolData: DataHandle ←
    Context.Find[dataType, ToolWindow.WindowForSubwindow[sw]];
    formItems: LONG POINTER TO ARRAY FormIndex OF FormSW.ItemHandle ← NIL;
    items ← AllocateItemDescriptor[nItems: FormIndex.LAST.ORD + 1];
    formItems ← LOOPHOLE[BASE[items]];
    formItems↑ ← [
    command: CommandItem[
      tag: "Command"L, place: [0, line0], proc: FormSWCommandRoutine],
    -- Specify string field in context as storage for string
    vanilla: StringItem[
      tag: "Vanilla"L, place: [90, line0], string: @toolData.string, inHeap: TRUE],
    another: CommandItem[
      tag: "Another"L, place: [250, line0], proc: FormSWCommandRoutine],
    destroy: CommandItem[
      tag: "Destroy"L, place: [350, line0], proc: FormSWCommandRoutine]];
    RETURN[items: items, freeDesc: TRUE]
    END;

<<Retrieve the context, create the subwindow handles, and then store the
handles in the context.>>
  MakeSWs: Tool.MakeSWsProc =
    BEGIN
    toolData: DataHandle ← Context.Find[type: dataType, window: window];
    logName: STRING ← [40];
    Tool.UnusedLogName[unused: logName, root: "MultiTool.log"L];
    toolData.msgSW ← Tool.MakeMsgSW[window: window];
    toolData.formSW ← Tool.MakeFormSW[window: window, formProc:
      MakeForm];
    toolData.fileSW ← Tool.MakeFileSW[window: window, name: logName];
    END;

  END...
```

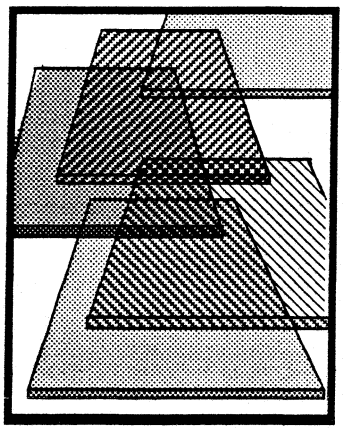
19.4 References

All of the procedures discussed in this chapter are documented in the **Context** chapter of your *Mesa Programmer's Manual.*

19.5 Exercises

The exercise for this chapter is to take your tool from the last chapter and make it a multiple instance tool.

20



Terminal interface package

In this chapter we discuss the Terminal Interface Package (TIP), which is responsible for recognizing user actions and producing corresponding program actions. TIP is specialized material; most of the code that you will write will not have to use the TIP facilities at all. You only need to learn about TIP if you want to change the way that a certain window handles user input, or if you are going to write a tool that uses a non-standard user interface. (We discuss how to write custom user interfaces in the next chapter.)

20.1 Definition of terms

Atoms — *Atoms* are unique keywords that are used in TIP tables as either actions or results of particular actions. Some predefined Atoms are **COPY**, **HELP**, **Point**, **COORDS**, **Video**, and **Word**.

Cursor — The *cursor* is the pointer that tracks mouse movements.

Input focus — The *input focus* is the location of the flashing caret that indicates the location of the next user input.

TIP table — A *TIP table* is used to map keystrokes and mouse actions to a list of results. TIP tables add an extra degree of indirectness in that you can edit them to change the interpretation of keystrokes. The structure of a TIP table is similar to a **SELECT** statement.

20.2 Discussion

In conventional computer systems you can only interact with one program at a time, so handling user input is not a big problem: the operating system interprets all user actions (keystrokes) and takes appropriate action. In the XDE, however, the problem is complicated by the fact that you can interact with many different windows simultaneously. TIP was designed to solve the additional complications created by a multi-window user interface. The following diagrams illustrate a simple single tasking user interface, a multitasking environment without TIP tables, and the XDE environment.

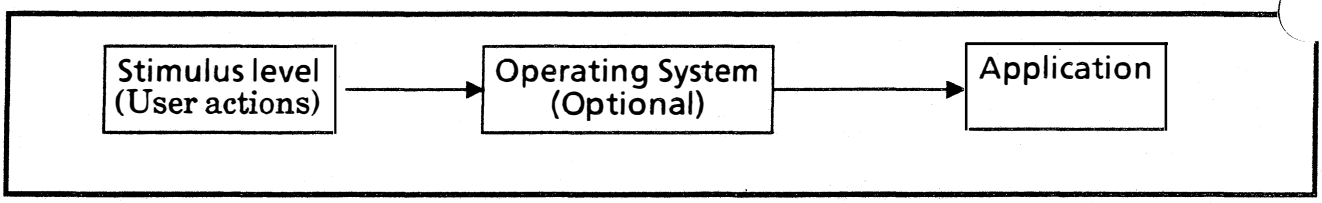


Fig. 20.1a Single task user environment

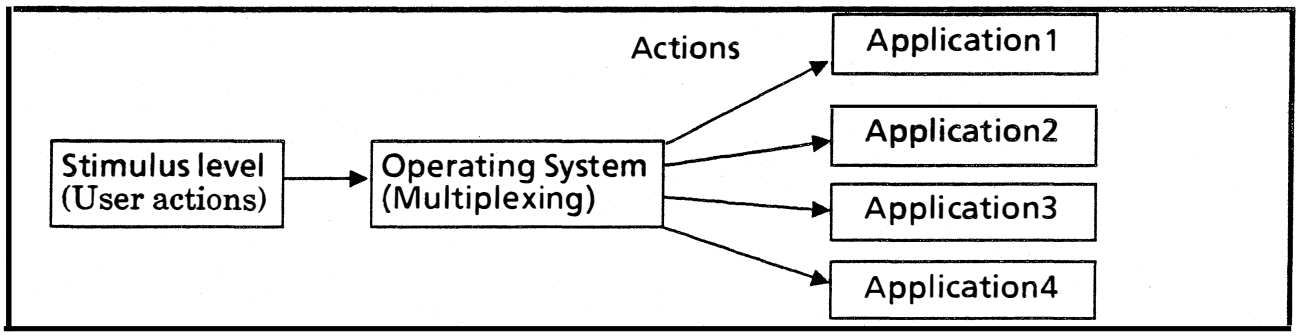


Fig. 20.1b Multitasking environment

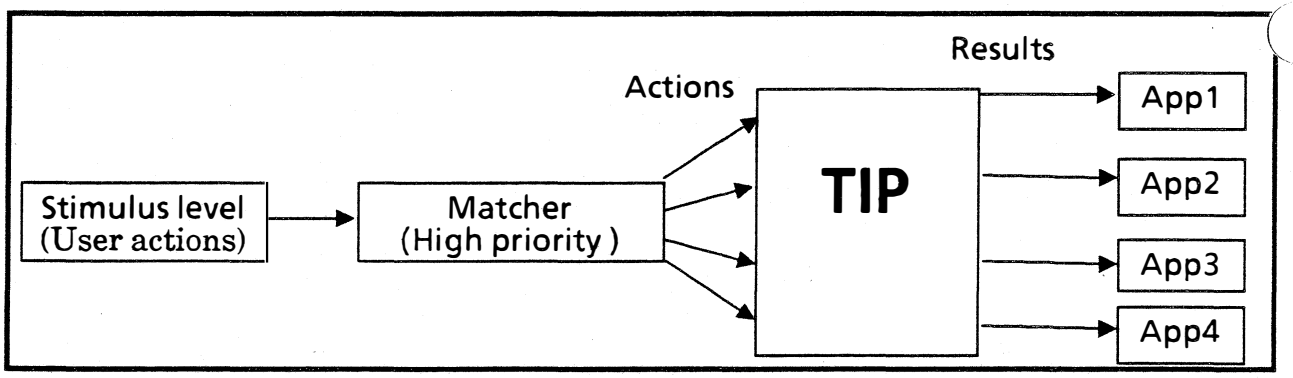


Fig. 20.1c XDE with TIP tables

TIP tables are thus the intermediary between user actions and program actions. A TIP table is basically just a list of user actions that the program is interested in, and a list of results associated with each user action. When the user does something, a high-priority process called the StimulusLevel (StimLev) puts that action on a *user action queue*. A second process called the Matcher dequeues each user action, figures out which window the action is intended for, and checks the TIP tables associated with that window. (If the action is a mouse click, the action is sent to the window with the current selection; any other action is sent to the window with the input focus.) The Matcher checks each TIP table associated with the appropriate window until it finds the action in a table, or until it runs out of TIP tables to check. If it doesn't find a match, the action is discarded; if it does find a match, it passes the associated results list to a special procedure called a NotifyProc.

The NotifyProc analyzes the results and produces the desired action. For example, when you select characters in a text window, the results from the mouse clicks are passed to a NotifyProc, which is responsible for doing the actual video-inversion (selection).

20.2.1 Default TIP tables

XDE provides seven default global TIP tables that are connected into a chain. Each of these tables corresponds to a particular type of window or subwindow, and that table acts as the head of the table chain for the user actions. Figure 20.2 shows the structure of the default chain of TIP tables. For example, if you are typing into the Executive the system would check tables in the following order: executive table, ttySW table, TextSW table, and finally Root table; if none of these tables provided a match, it would ignore the action.

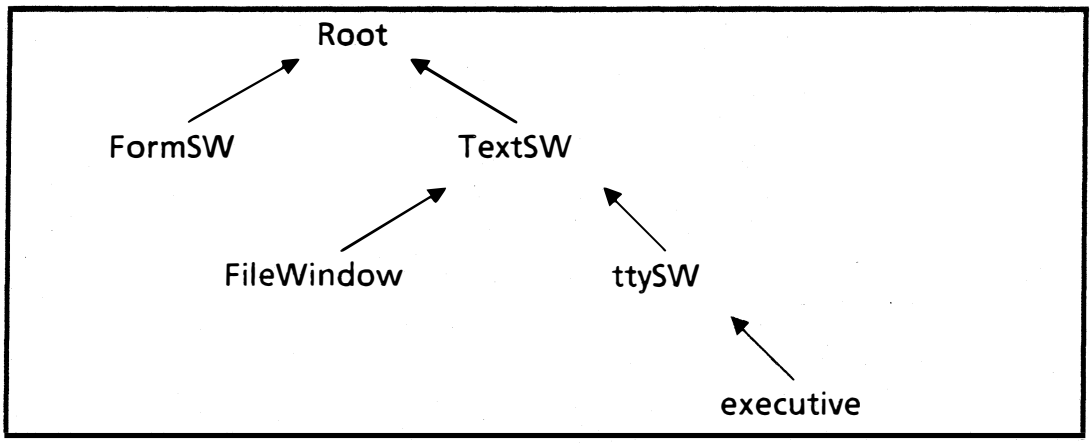


Fig. 20.2 Default TIP table chain

Generally you will have a chain of TIP tables associated with a given window and a NotifyProc associated with that same window. However, the relationships between tables, windows and NotifyProcs can become quite complex. You don't need to worry about all the gory details of how these three pieces interrelate, but you should be aware that the relationship is not always simple.

20.2.2 TIP table syntax

There are two parts to a TIP table: an options list and a "trigger" statement. We don't discuss the options list in this chapter; you'll have to consult the MPM to find out what options are available. The main body of a TIP table resembles a Mesa SELECT statement. The left hand side of the table contains various user actions; for example, **A down** means the "A" key was pressed down, and **Point Up** indicates that the left mouse button was released. (For a complete list of possible actions, see the TIP chapter of the *Mesa Programmer's Manual*.)

The right hand sides of TIP tables are *results* that are to be passed to the NotifyProc. A result must be one of the following seven variants;

```
TIP.ResultElement: TYPE = RECORD [
  SELECT type: * FROM
    char = >   [c: CHARACTER],          --character representation of last user action
    coords = >[place: Window.Place],-       -current bitmap position of mouse
    keys = >   [keys: LONG POINTER TO Keys.KeyBits],    --state of entire keyboard
    atom = >   [a: Atom.ATOM],    --unique string
    int = >    [i: LONG INTEGER],
    string = > [s:LONG STRING],
    time = > [time: System.Pulses], --time of last user action
  ENDCASE];
```

The most common of these results is the *atom*, which is basically just a unique character string used as a label. There are some standard system-defined atoms; mostly, you will define your own atoms to stand for actions that you want to recognize. Here is a sample TIP table:

```
SELECT TRIGGER FROM
  Point Down AND Point Up BEFORE 200 = >
    SELECT ENABLE FROM
      LeftShift Down = > COORDS, ShiftedClick
    ENDCASE = > COORDS, SimpleClick;
ENDCASE...    -- use three periods to indicate the end of your table
```

In this example we **SELECT** the case where the left mouse button goes down (**Point Down**) and then comes back up (**Point Up**) before 200 milliseconds has elapsed. If these actions occur the system checks to see if the left shift key is also down. If the shift key is down, the results **COORDS** and **ShiftedClick** are passed to the NotifyProc; otherwise, the results **COORDS** and **SimpleClick** are passed. **COORDS** is a system-defined result; **SimpleClick** is a client-defined atom. You can name the atom anything you like; it just has to be a label that you can recognize in your NotifyProc and act on accordingly.

The most common keywords in TIP tables are **TRIGGER**, **WHILE**, **AND**, and **ENABLE**. **TRIGGER** and **AND** refer to events that have just happened; that is, the event in question has just been dequeued from the user action queue. **ENABLE** and **WHILE** refer to the current state of something, regardless of whether or not it just reached that state. Thus, every TIP table must have at least one **TRIGGER** statement; this is the recent user action that has caused the Notifier to check the TIP table. Once a user action has been matched to a **TRIGGER** statement, you can use **ENABLE** statements to find out what else is true at the current time.

20.2.2.1 Modifying TIP tables

There are two ways that you can change the TIP tables associated with a window. You can modify an existing table, in which case the changes will affect all windows of the particular class, or you can write a new TIP table, and associate it with a particular window or window class. We discuss how to modify an existing table in this section; in the next section, we discuss how to integrate a new table into the existing structure.

Obviously, you can modify a TIP table either by changing the left hand side (thereby affecting which actions are recognized), or by changing the right hand side (affecting what happens once an action is recognized.) Once you have edited a system TIP file, you need to reboot if you want those changes to take effect. The reason is that the system uses a compiled version of the TIP tables, and that the compiled version will not be created until

you reboot. (System TIP tables live on the directory <CoPilot>TIP. The .TIP files are the text files; the .TIPC files are the compiled versions that are created whenever you boot.)

For example, suppose that you want to change the way mouse clicks are interpreted in text subwindows. Instead of the standard scheme (one click selects a single character, two clicks a word, and three clicks a line), suppose you always want to select a word, regardless of the number of clicks. You can do this by modifying the right hand side of the TIP table; in this case we add the result **Word**. Below is a portion of the TextSW TIP table after the modification:

```
SELECT TRIGGER FROM
  Point Down => 
    SELECT TRIGGER FROM
      Adjust Down BEFORE 200 => Time, COORDS, Menu;
    ENDCASE => 
      SELECT TRIGGER FROM
        CONTROL Down => Time, COORDS, MoveInsertion
        COPY OR MOVE => Time, COORDS, DoPrimary
      ENDCASE => Time, COORDS, Video, InsertToSel, Word, DoPrimary    -- add the result Word
          -- to the existing results list
  .
  .
ENDCASE...
```

Thus, any point click that is not part of a chord, a CONTROL-POINT or a copy/move will cause the atom **word** to be passed to the NotifyProc. To make this change really take effect, we obviously have to also change the NotifyProc so that it recognizes the new result **word**. We do this in section 20.2.3.

20.2.2.2 Writing new TIP tables

In addition to modifying existing tables, you can also write new TIP tables. When you write a new TIP table, you can make that table apply to all windows on the screen, to all windows of a particular class, or to just one window. You also have the choice of whether or not to attach it to the default chain. If you attach it to the chain, any actions not handled by your TIP table will go through the standard chain. If you don't attach it, any actions that you don't handle will be ignored.

The first step is to create a compiled version of the table that your program can use. To create a compiled version from a text file, you call **TIP.CreateTable**:

```
TIP.CreateTable: PROCEDURE [file: LONG STRING ← NIL,  -- source file for the TIP table
    opaque: BOOLEAN ← FALSE,                          -- don't search successive tables if TRUE
    z: UNCOUNTED ZONE ← NIL,                          -- allocate table from z
    contents: LONG STRING ← NIL]                      -- contents can contain the TIP table
    RETURNS [table: TIP.Table]                        -- in a character string as below
```

With **CreateTable** you either supply the TIP table text in **file**, or you can use the **contents** parameter and have the text within your program. The procedure below illustrates how to create a TIP table using the **contents** parameter to fill in the table code rather than using a file.

```
--excerpted from TIPExample2.mesa
myTip: TIP.Table ← NIL; -- declare the TIP table (usually as a global variable)

InitTip: PROCEDURE =
BEGIN
  tipContents: LONG STRING ←
  "-- This TIP table makes single click Point select a word
  -- Top-Level trigger select -- (these comments will go inside the TIP table)

  SELECT TRIGGER FROM
    Point Down WHILE CONTROL Up WHILE COPY Up WHILE MOVE Up = >
     SELECT TRIGGER FROM
       Adjust Down BEFORE 100 = > TIME, COORDS, Menu;
    ENDCASE = > TIME, COORDS, Video, InsertToSel, Word, DoPrimary;
  ENDCASE...
   "L; -- end of TIP table code - note that this is a 10 line string literal

  firstTime: BOOLEAN ← TRUE;
  myTip ← TIP.CreateTable[
    file: "TIPExample2.TIP"L, contents: tipContents !
    TIP.InvalidTable = >
      IF type # badSyntax THEN CONTINUE    -- file wasn't found
      ELSE {
      UserTerminal.BlinkDisplay[];    -- table has bad syntax so check log file
      IF firstTime THEN {firstTime ← FALSE; RESUME }}];
END;
```

If want to put your TIP table in a string rather than a file, you need to pay particular attention to the way the error **TIP.InvalidTable** is handled. The call to **CreateTable** will search first for a file of the specified name; if it can't construct a compiled table from the contents of the file, it will raise the error **InvalidTable**, with **type** = **badSyntax**. You must catch the signal and **RESUME** it; the second time, the **contents** string will be used as the table. If this error is raised a second time, there really is a syntax error in your table. (If you use a file as the source of your table, rather than a string, the file must be stored on the <>TIP local directory.)

Once you have created the table, you need to decide how it is to interact with the existing chain, and then you need to associate it with one or more windows. To hook your new TIP table into the existing chain, you call either **TIP.PushLocal** or **TIP.PushGlobal**:

```
TIP.PushGlobal: PROCEDURE [push: TIP.Table,
  onto: TIP.GlobalTable, opaque: BOOLEAN ← FALSE];

TIP.PushLocal: PROCEDURE [push, onto: TIP.Table, opaque: BOOLEAN ← FALSE];
```

PushGlobal inserts **push** after the global table indexed by **onto**; **PushLocal** pushes the table **push** in front of the table **onto**. If opaque is **TRUE**, any actions that your table does not recognize will be ignored; if **opaque** is **FALSE**, the system will continue to check the other TIP tables. You need only push the new table onto the chain once, so put the call to **PushLocal** or **PushGlobal** in your initialization code. After the new table is in place, call **TIP.SetTable**, which associates the table with a particular window:

```
TIP.SetTable: PROCEDURE [window: Window.Handle, table: TIP.Table]
  RETURNS [oldTable: TIP.Table];
```

Just pushing the new TIP table is not enough; you must associate the table with a window or the new table will be invisible. Below we push the TIP table **myTip** onto the global TextSW table and then associate **myTip** with a file subwindow. In this case, we aren't interested in the old TIP table, so we discard the results record returned by **SetTable**.

```
Init: PROCEDURE =
BEGIN
  IF myTip # NIL THEN TIP.PushLocal[push: myTip, onto: TIP.globalTable[textSW]];
  [] ← TIP.SetTable[window: toolData.fileSW, table: myTip];
END;
```

20.2.3 NotifyProcs

So much for the TIP tables themselves. The second big piece of the TIP mechanism is the NotifyProc, which analyzes the results passed to it and then performs the desired function. A **TIP.NotifyProc** is defined as:

```
TIP.NotifyProc: TYPE = PROCEDURE [window: Window.Handle, results: TIP.Results];

TIP.Results: TYPE = LONG POINTER TO TIP.ResultsList;

TIP.ResultsList: TYPE;
```

Thus, the results that are passed to the NotifyProc are opaque; you don't know anything about the structure of a **ResultsList**. The only way you can access them is with the procedures **TIP.Rest** and **TIP.First**:

```
TIP.Rest: PROCEDURE [results: TIP.Results] RETURNS [TIP.Results];

TIP.First: PROCEDURE [results: TIP.Results] RETURNS [TIP.ResultElement];
```

First returns a **Tip.ResultElement**, which is a variant record containing one of a number of different result types (see section 20.2.2). **Rest** returns the results less the first one in the list (**First** and **Rest** are similar to Lisp's Car and Cdr primitives.) The following NotifyProc taken from **TIPExample3.mesa**) displays the selected text (a single word) in a string field on the form subwindow. Thus, when you click Point on a word, the text is both selected and printed in the form subwindow.

```
--this call goes in the MakeSWs proc
toolData.oldNotifyProc ←
   TIP.SetNotifyProc[window: toolData.fileSW, notify: StuffSelection];

StuffSelection: TIP.NotifyProc =
BEGIN
   s: LONG STRING ← NIL;
   word: Atom.ATOM ← Atom.MakeAtom["Word"L];          -- declare atoms
   PointUp: Atom.ATOM ← Atom.MakeAtom["PointUp"L];
   toolData: DataHandle ← Context.Find[type: dataType, window:
      ToolWindow.WindowForSubwindow[window]];          -- find the context
   toolData.oldNotifyProc[window: window, results: results]; -- call the old NotifyProc
                                                       -- to interpret the user input
   FOR input: TIP.Results ← results, input.Rest[] UNTIL input = NIL DO
      WITH input.First[] SELECT FROM                   -- analyze the results one at a time
         z: TIP.ResultElement.atom = > SELECT z.a FROM --variant record syntax
         word = > toolData.selectOccurred ← TRUE       -- first action was Point down
         PointUp = > IF toolData.selectOccurred THEN   -- when Point comes up
         BEGIN                                               -- display text
            toolData.selectOccurred ← FALSE;
            s ← Selection.Convert[string];             -- convert selection to a string
            IF String.Empty[s] THEN RETURN
            ELSE
            BEGIN
               heap.FREE[@toolData.selection];
               toolData.selection ← String.CopyToNewString[s, heap];   -- display string to
               FormsW.DisplayItem[toolData.formSW, FormIndex.select.ORD]; -- window
               heap.FREE[@s];
               RETURN
            END;
         END;
         ENDCASE;
      ENDCASE;
   ENDLOOP;
END;
```

The above code is only interested in two user actions, the Point button going down and then going back up. When Point goes down, the oldNotifyProc selects a word; when Point comes back up, the text is displayed in a string field in the form subwindow.

Before the Notify code is called, we call **SetNotifyProc** in the initialization code. If you are associating a NotifyProc with a particular window, you must call **SetNotifyProc** each time your window is activated. (Remember, NotifyProcs are usually associated with windows rather than with TIP tables.) Thus, we put the call in **MakeSWs**, which is called each time the subwindow is created. **SetNotifyProc** associates the **StuffSelection** NotifyProc with the file subwindow; all user actions directed toward the file subwindow must first go through this procedure. **SetNotifyProc** returns the old notify proc for the window:

```
TIP.SetNotifyProc: PROCEDURE [window: Window.Handle,
      notify: TIP.NotifyProc]
      RETURNS [oldNotify: TIP.NotifyProc];
```

The first thing **StuffSelection** does is "make" the necessary atoms. A NotifyProc must call **Atom.MakeAtom** for every Atom that it wants to recognize, regardless of whether its a "standard" atom. (If you have a lot of atoms to initialize, you should do it in a separate procedure.) **MakeAtom** returns the atom corresponding to the string, creating one if necessary.

StuffSelection next finds the context for the subwindow to which the action belongs. It then calls the **oldNotifyProc** (which is returned by **SetNotifyProc**). **oldNotifyProc** interprets the Point Down motion by selecting the entire word where the cursor is located (this example uses the TIP table from section 20.2.2.) The **oldNotifyProc** also interprets most other user actions.

When the **oldNotifyProc** returns, **StuffSelection** also interprets the same results list. **StuffSelection** is only interested in the two Atoms **PointUp** and **word**. When **word** is encountered we know that a select (**PointDown**) has occurred, since **word** is one of the results returned by the modified TIP table. Thus we set a **BOOLEAN** to indicate that a word was selected, and return.

The next time **StuffSelection** is entered is when the the mouse button comes up; when this occurs the selected text is converted into a string. The string is then displayed in the form subwindow and the storage that the string occupied is released.

20.2.4 The GPM: Macro Package

The system TIP tables are all coded in macro format. To help you to understand this language we will show an example of a system TIP table and discuss its features. For a more complete explanation refer to the References at the end of this chapter.

```
-- TextSW.TIP; created by System
-- Version of 25-Jan-83 15:30:21

[DEF,ChordTime,(100)]    -- define ChordTime to be 100 milliseconds

[DEF,CopyMove,(COPY Down | MOVE Down)]  -- define CopyMove to be either COPY or MOVE

[DEF,TC,(TIME COORDS)]-- define an abbreviation for the Atoms, TIME and COORDS

[DEF,Chord,(SELECT TRIGGER FROM   -- define the Chord macro
  ~1 Down BEFORE [ChordTime] => { [TC] Menu };
ENDCASE => ~2)]

-- Top-Level trigger select
SELECT TRIGGER FROM
  Point Down => [Chord,Adjust,
  SELECT ENABLE FROM
      CONTROL Down => { [TC] MoveInsertion };
      [CopyMove] => { [TC] Video DoPrimary };
  ENDCASE => { [TC] Video InsertToSel DoPrimary }];

  Adjust Down => [Chord,Point,
  SELECT ENABLE FROM
      [CopyMove] => { [TC] ExtendPrimary };
  ENDCASE => { [TC] InsertToSel ExtendPrimary }];
```

The TIP table begins with a series of definitions. Each definition consists of the keyword **DEF**, followed by the macro-name, followed by the macro-definition body. Items in parentheses are taken as literals; thus, **[DEF, ChordTime, (100)]** sets **ChordTime** to 100 milliseconds.

The **Chord** macro's body performs the **SELECT** operation on the first argument (denoted **~1**) of the actual parameter list. If there is no match in the **SELECT** statement, the result returned is the second argument of the macro. For example, when a Point Down occurs in the Top-Level select statement, the **Chord** macro takes **Adjust** as its first argument and selects its second argument depending on which other keys are currently depressed. Thus, if **Point** goes down and **Adjust** goes down before **ChordTime** then the results passed to the NotifyProc are **{ [TC] Menu }**. If **Adjust** does not go down, then other actions are checked such as **CONTROL Down** or **CopyMove**.

20.3 Summary

The TIP mechanism translates keyboard and mouse actions into program actions. TIP performs this translation by matching user actions in a TIP table and passing the corresponding results to a NotifyProc. The NotifyProc then interprets the results and takes appropriate actions.

You can easily change the global interpretation of keystrokes and mouse movements by editing the global TIP tables. You need only edit the table and reboot to make the changes take effect.

You can create a new TIP table either within your program or in a separate text file. You must create the TIP table using **TIP.CreateTable**, push the table on the TIP table chain, and finally call **TIP.SetTable** to associate the table with a window.

You can also affect the way actions are interpreted by modifying a NotifyProc or by writing a new one.

20.4 References

The TIP chapter in the *Mesa Programmer's Manual* describes the TIP interface and gives the BNF for TIP tables.

The "General Purpose Macrogenerator" in the October 1965 Computer Journal is the basis on which the TIP macro package is based. If you intend to do a great deal of modifications to the system TIP tables this article is helpful.

The *Workstation Programmer's Manual* provides several examples of TIP tables and NotifyProcs.

20.5 Exercise

The exercise for this chapter is to write a program that transposes two characters in the current selection. The user should be able to switch any two adjacent characters by selecting them and then pressing the PROP'S key.

In order to write this program you will need to learn about another TIP procedure: TIP.**SetNotifyProcForTable**:

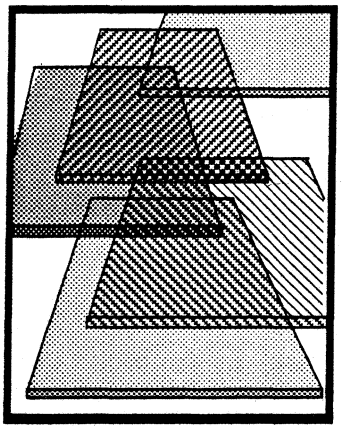
```
TIP.SetNotifyProcForTable[table: TIP.Table,
  notify: TIP.NotifyProc] RETURNS [oldNotify: TIP.NotifyProc];
```

SetNotifyProcForTable associates **table** with **notify** and returns the previous NotifyProc, if any. This is different than **SetNotifyProc**, which associates a NotifyProc with a window. **SetNotifyProcForTable** enables you to channel all actions through a global TIP table and then to your own NotifyProc. If you do not recognize a particular action (its not in your TIP table), it will be interpreted by other tables further down the chain. Thus you can add another level to the interpretation of user commands for a particular window class.

For this exercise you will need to write a TIP table that recognizes one action (the PROP'S key) and passes a result to your NotifyProc. Your NotifyProc must recognize the results that your TIP table generates and take actions to transpose the characters. Since the screen manipulation requires interfaces that you haven't used yet, we have provided a procedure that returns the current selection as a character string (**GetSelection**), a procedure that deletes the current selection (**DeleteSelection**), and a procedure that inserts a character string into a Text subwindow (**InsertText**). The interface for these procedures is called TransposeDefs.mesa.

Notes:

21



Creating subwindows

In the past five chapters you have been layering applications on top of Tajo's collection of predefined subwindows. Using these standard subwindow types insulates you from actually displaying information on the screen; until now, you've only had to worry about what was being displayed on the screen, not how it got there.

For most applications, you can blithely use the standard subwindow types and never have to worry about the low-level details. However, if your application requires an unusual user interface, such as a graphics-based interface, you will not be able to use a standard subwindow. Instead, you will have to implement your own subwindow type to support your desired functionality. This chapter discusses some of the strategies and mechanisms for accomplishing this.

A word to the wise: creating and supporting your own subwindow is not easy. You may want to postpone reading this chapter until you really need to implement your own subwindow.

21.1 Definition of terms

Clip

To *clip* a window is to cut off any information that the window attempts to paint outside of its established boundaries. Thus, Tajo will *clip* a subwindow if it attempts to paint beyond the boundaries of its parent window.

Clipping window

A *clipping window* is a window that prevents a subwindow from painting outside of its boundaries. In Tajo, a subwindow can only paint within the boundaries of its parent window; a clipping window is a window that is placed just within the confines of the parent window, thereby ensuring that a subwindow does not paint too close to the border.

21.2 Discussion

In this chapter, we demonstrate subwindow creation with an example called the BoxedTool, which has one graphics subwindow. The subwindow is basically just a grid (electronic graph paper); when the user selects a box in the grid with the mouse, the box video-inverts. We use this tool to illustrate how to register a new subwindow type, how to draw the grid on the screen, and how to perform the video-inversion of the boxes. We use a

second example, called ScrollBoxedTool, to illustrate how to add a scrollbar to the tool, and how to adjust things when the user changes the size of the tool window.

You might want to run BoxedTool in Tajo now, to familiarize yourself with how it works before you find out how it is implemented.

21.2.1 Registering a subwindow type

When you want to create a custom subwindow, the first step is to register a new subwindow type with Tajo. To do so, you call **Tool.RegisterSWType**, passing in some call back procedures to handle size adjustments and window state transitions. Tajo returns a unique subwindow type.

```
Tool.RegisterSWType: PROCEDURE [
   adjust: ToolWindow.AdjustProcType ← Tool.SimpleAdjustProc,
   sleep: ToolWindow.SWProc ← Tool.NopSleepProc,
   wakeup: ToolWindow.SWProc ← Tool.NopWakeupProc]
RETURNS [uniqueSWType: Tool.SWType];
```

The **adjust** procedure is called whenever the user moves the subwindow or changes the subwindow size. The **sleep** procedure is called whenever the window in which the subwindow lives becomes tiny. The subwindow is then expected to throw away any data that it uses only to display its contents. The **wakeup** procedure is the inverse of the **sleep** procedure.

You only need to write an AdjustProc when the information in your window is dependent on the size of the window. For example, if you had a tool with a 5 by 5 grid and you always wanted the grid to exactly fill the window, then you would need an AdjustProc. The BoxedTool, however, does not require an AdjustProc; the boxes are of fixed size and nothing depends on the size of the window. If the window is big, you will see more of the boxes than you do when the window is small, but the boxes do not have to be scaled to fit the window. Similarly, you do not always have to provide **sleep** and **wakeup** procedures. Instead, you can use the default procedures, or bypass the step completely by using the **Tool.SWType** of **vanilla** instead of getting your own subwindow type. For example, BoxedTool has the following global variable:

```
mySWType: Tool.SWType ← Tool.RegisterSWType[];
```

21.2.2 Creating a subwindow

Once you have a subwindow type, you use it to create an instance of that class. As with predefined subwindows, you do this in a **Tool.MakeSWsProc**. For example:

```
MakeSWs: Tool.MakeSWsProc =
   BEGIN
   data.sw ← Tool.MakeClientSW[window, MyCreateSWProc, NIL, mySWType];
   END;
```

This creates a subwindow "shell"; **MyCreateSWProc** is a call back procedure that creates the functionality for the subwindow. **mySWType** is the unique subwindow type that we obtained from **RegisterSWType**, and **window** is a handle to the parent window (**window** is a parameter to **MakeSWs**.)

21.2.3 Making a subwindow do something useful

The job of **MyCreateSWProc** is to associate functionality with the subwindow.In this procedure, we associate a DisplayProc and a NotifyProc with our subwindow. The DisplayProc is a call back procedure stored in the window object; Tajo will call this procedure whenever the information displayed in the window needs to be updated. We discuss our DisplayProc in section 12.2.1.3, and our NotifyProc in section 21.2.1.4.

```
MyCreateSWProc: PROC [sw: Window.Handle, clientData: LONG POINTER] =
  BEGIN
  [] ← TIP.SetNotifyProc[window: sw, notify: MyNotify];
  boxArray[0][0] ← black; --make corner box black
  [] ← Window.SetDisplayProc[sw, DisplayProc];
  END;
```

The call to **SetNotifyProc** returns the old notify proc associated with the window. In this case, we don't care about the old notify proc, so we discard the results record. Similarly, **SetDisplayProc** returns the old display procedure, which we also discard.

21.2.3.1 DisplayProcs

A DisplayProc is a call back procedure that is responsible for displaying information on the screen. This procedure is called once to initialize the tool's display, and then again each time the display needs to be updated. Once a window is active, Tajo oversees its display state largely by managing an invalid box list. This list represents those regions of the window that no longer have valid contents and therefore need to be repainted. When an operation (like moving a window off of another) invalidates portions of the screen, the operation must tell Tajo to validate its window structures. During validation, Tajo calls the DisplayProc for each window that has invalid regions.

Before calling the client's DisplayProc, however, Tajo creates a bad phosphor list. This list consists of the visible portions of the window's invalid areas. While a bad phosphor list exists for a window, any painting done to that window will be clipped to the bad phosphor list (i.e., only visible invalid areas will be repainted.)

A DisplayProc can implement one of two methods for repainting a window. First, it can call **Window.EnumerateInvalidBoxes**, supplying a call-back procedure, which will be called for each invalid box in the window's invalid box list. By repainting each invalid box, the DisplayProc can updates the display to reflect the current state. **EnumerateInvalidBoxes** should be called only from within a DisplayProc.

The other option for repainting a window is to ignore the invalid boxes and simply repaint the entire window. Since there is a bad phosphor list, only those areas that need repainting will be refreshed; the rest is ignored. Since this is the easier of the two operations, it is preferred. Only use **EnumerateInvalidBoxes** if there is no bad phosphor list (the window has never been validated) or if it takes too long to try and paint the whole window. (The latter case should hopefully not happen very often.) The following code fragment illustrates these two methods.

The DisplayProc gets the current window box and then loops through the pixels along the x-axis painting vertical lines, and then along the y-axis painting horizontal lines. This is

an example of just repainting the entire window, without looking at the invalid list. To put the boxes within the grid, however, we use the **EnumerateInvalidBoxes** method.

Once the lines are drawn, we call **EnumerateInvalidBoxes**, passing **DProc**. Thus, **DProc** will be called for each invalid box on the window's invalid list. **DProc** just converts from bitmap coordinates into row/column coordinates, and then calls **DisplayBox** to do the actual painting. Inside **DisplayBox**, we loop through the appropriate portions (i.e., the invalid ones) of a two-dimensional array representing the boxes in the grid. A box may be either white or black. If the data structure indicates the box as black, we call **Display.Black** to draw the box. We don't have to do anything for the white boxes, since Tajo has already painted them white.

```
DisplayProc: ToolWindow.DisplayProcType =
  BEGIN
  box: Window.Box ← Window.GetBox[window];
  --vertical. Starting at zero, increment by width until we reach edge of box.
  --In the loop, call Display.Line. Each line starts at box.place.x + i, 0
  --and ends at box.place.x + i, box.dims.h.
  FOR i: INTEGER ← 0, i + width UNTIL i > = box.dims.w DO
    Display.Line[window, [box.place.x + i, 0], [box.place.x + i, box.dims.h]]
    ENDLOOP;
  --horizontal
  FOR i: INTEGER ← 0, i + height UNTIL i > = box.dims.h DO
    Display.Line[window, [0, box.place.y + i], [box.dims.w, box.place.y + i]]
    ENDLOOP;
  Window.EnumerateInvalidBoxes[window, DProc];
  END;

--called once for each box on the window's invalid box list
  DProc: PROC [window: Window.Handle, box: Window.Box] =
    BEGIN rc: RC ← RCForBox[box]; DisplayBox[window, rc] END;

  DisplayBox: PROC [window: Window.Handle, rc: RC] =
    BEGIN
    FOR i: CARDINAL IN [rc.br..rc.er) DO
      FOR j: CARDINAL IN [rc.bc..rc.ec) DO
        IF boxArray[i][j] = black THEN
          Display.Black[
            window, [
            [(j - firstColumn) * width, (i - firstRow) * height], [
            width, height]]];
        ENDLOOP;
      ENDLOOP;
    END;
```

A window's location is defined in terms of its parent window. Thus, in **DisplayProc**, **Window.GetBox** returns the coordinates of the subwindow relative to its parent. However, the operations used to paint the bits (**Display.Line**, **Display.Black**, etc.) use window-relative coordinates; thus, the place specified in the box represents the xy offset from the window's origin. [**[box.place.x + i, 0]** are the window relative coordinates that specify where the line is to start; **[box.place.x + i, box.dims.h]** specify where the line is to stop. Because of

this difference, you should always use a **box.place** of [0,0] in calls to the **Display** interface. This accounts for the 0 in the lines:

Display.Line[window, [box.place.x + i, 0], [box.place.x + i, box.dims.h]] and
Display.Line[window, [0, box.place.y + i], [box.dims.w, box.place.y + i]].

21.2.3.2 The NotifyProc

The NotifyProc associated with a window is responsible for recognizing "interesting" user actions and acting upon them. The only user action that BoxedTool cares about is PointUp, or releasing the left mouse button. When this happens, the NotifyProc needs to video-invert the box under the cursor. When it recognizes a mouse click, the NotifyProc calls **InBox** to convert the current mouse position (a **Window.Place**) to a **Window.Box**, and then calls **RCForBox** to convert the **Window.Box** to row/column coordinates. It then toggles the color of the box in the box array. To make the display consistent with the data structure, it invalidates the appropriate region on the display and then tells Tajo to validate its windows. The call to **Window.ValidateTree** eventually calls **DisplayProc**, which does the actual painting. This is an example of the standard way to update display information.

Another way of doing this would be to paint the boxes directly from the NotifyProc. Thus, instead of invalidating the boxes and then calling **ValidateTree**, you could just perform a **Display.Black** right here. The choice of whether to do the display directly from the **NotifyProc** or to do an **Invalidate** to force your **DisplayProc** to be called is basically an efficiency issue. Doing the display directly from the **NotifyProc** is probably slightly faster for easy operations, such as displaying a single box, but if the operation is complicated, you shouldn't lock up the Notifier to do the display from the NotifyProc. Doing an invalidation and a validation to force the system to call your DisplayProc is never wrong.

```
MyNotify: TIP.NotifyProc =
  BEGIN
  wp: Window.Place ← [0, 0];
  pointUp: Atom.ATOM ← Atom.MakeAtom["PointUp"L];
  FOR input: TIP.Results ← results, input.Rest[] UNTIL input = NIL DO
    WITH z: input.First[] SELECT FROM
        atom = >
            IF z.a = pointUp THEN
                BEGIN
                box: Window.Box ← InBox[wp]; --convert to window.box
                rc: RC ← RCForBox[box];   -- convert to row/column
                IF boxArray[rc.br][rc.bc] = black THEN boxArray[rc.br][rc.bc] ← white
                ELSE boxArray[rc.br][rc.bc] ← black;
                Window.InvalidateBox[window, box];
                Window.ValidateTree[window];
                END;
        coords = > wp ← z.place; --represents current mouse position
        ENDCASE;
    ENDLOOP;
  END;
```

21.2.4 Implementing scrolling

ScrollBoxedTool is just like BoxedTool except that it can be scrolled vertically. We have added procedures to create the scrollbar, to calculate how much should be scrolled, to perform the scrolling, and to adjust the scrollbar window so it shares part of the subwindow's space. The rest of the code is the same as in BoxedTool. (Although the DisplayProc looks different, it works the same way; it first displays the grid lines, then it enumerates the invalid boxes to repaint the sections of the boxedArray.)

21.2.4.1 Creating the scrollbar

You create a scrollbar on a subwindow by calling **Scrollbar.Create**. Scrollbars are a slight anomaly in Tajo. The size of a scrollbar is tied to the subwindow it appears with, which might lead you to believe that the scrollbar should be a child of the subwindow. However, though children obscure parent windows, they do not clip them: thus, if a scrollbar were the child of a subwindow, it would obscure some of the subwindow's information, not adjust it to the right (top). Therefore, a scrollbar window is a sibling of its subwindow. This means that we have to create the scrollbar in **MakeSWs** and not **MyCreateSWProc**:

```
MakeSWs: Tool.MakeSWsProc =
  BEGIN
  data.sw ← Tool.MakeClientSW[window, MyCreateSWProc, NIL, mySWType];
  Scrollbar.Create[data.sw, vertical, Scroll, Therm];
  END;
```

The parameters of **Scrollbar.Create** include two procedures: **Scroll** and **Therm**. **Therm** is used to get the scrollbar data from the client in order to display it to the user. Its primary purpose is to calculate which portion of the available plane of information is being displayed in the window, and what percent of the plane this viewing portion represents. Tajo uses this information to display the dark areas in the scrollbar window. **Scroll** is called when the user makes a scroll request. It is responsible for shifting the display and adjusting the underlying data structures so the DisplayProc can repaint the area.

21.2.4.2 Calculating scrolling information

Tajo manages the visual cues in the scrollbar window that indicate the percentage and portion of the plane that the user is viewing. Tajo calls **Therm** to get this information.

```
Therm: Scrollbar.ScrollbarProcType =
  BEGIN
  wbox: Window.Box ← Window.GetBox[window];
  rows: INTEGER ←  --number of rows currently displayed
    IF wbox.dims.h MOD height = 0 THEN wbox.dims.h / height
    ELSE wbox.dims.h / height + 1;
  IF rows + data.firstRow > maxRows THEN rows ← maxRows;  --bottom of grid
  RETURN[
    [[0, 0], wbox.dims], (100 * data.firstRow) / maxRows,
        (100 * rows) / maxRows];
  END;
```

ScrollBoxedTool maintains two variables, **firstRow** and **firstColumn**, to represent the first row and first column being displayed at the top of the window. By calling **Window.GetBox** to get the dimensions for the window, and using the known width and height of a box, **Therm** calculates how many rows are currently displayed and subsequently the percentage of rows being displayed. Adjustments are made if the bottom of the grid has been scrolled into the viewing area.

21.2.4.3 Scrolling

Scrolling involves calculating how much of the window to scroll, shifting the display, and validating the window (so the DisplayProc will be called). The ScrollProc performs these functions:

```
Scroll: Scrollbar.ScrollProcType =
  BEGIN
  box: Window.Box ← Window.GetBox[window];
  rowsToScroll: INTEGER ← 0;
  rows: INTEGER ←     --rows currently displayed
    IF box.dims.h MOD height = 0 THEN box.dims.h / height
    ELSE box.dims.h / height + 1;
  IF rows > maxRows THEN RETURN;  --do not scroll
  SELECT direction FROM
    forward = >
      BEGIN     --percent is passed as a parameter
      rowsToScroll ← (rows * percent) / 100;
      IF (rowsToScroll + data.firstRow + rows) > maxRows THEN
        BEGIN
        rowsToScroll ← (maxRows) - (rows + data.firstRow);
        IF box.dims.h MOD height # 0 THEN rowsToScroll ← rowsToScroll + 1;
        END;
      Display.Shift[window, [[0, rowsToScroll * height], box.dims], [0, 0]];
      data.firstRow ← data.firstRow + rowsToScroll;
      Window.ValidateTree[window];
      END;
    backward = >
      BEGIN
      rowsToScroll ← (rows * percent) / 100;
      IF rowsToScroll > data.firstRow THEN rowsToScroll ← data.firstRow;
      Display.Shift[window, [[0, -(rowsToScroll * height)], box.dims], [0, 0]];
      data.firstRow ← data.firstRow - rowsToScroll;
      Window.ValidateTree[window];
      END;
    relative = > NULL;
    ENDCASE;
  END;
```

In the case of ScrollBoxedTool, scrolling requires calculating the number of rows to scroll. We maintain the row currently at the top of the subwindow as a variable to help check boundary cases. Once we have figured out the number of rows to scroll, we shift the display the appropriate number of lines forward or backward, then then validate the

window. Notice that scrolling relative to some point in the window ("thumbing") is not implemented.

21.2.5 Adjusting subwindows with scrollbars

In BoxedTool, we did not have to write an AdjustProc, since our grid image did not depend on the size of the window. Now, however, we have added a scrollbar, and life is suddenly more complicated. The AdjustProc associated with all windows created using **Tool.Create** does not know how to redistribute new box sizes to subwindows with scrollbars. Thus, ScrollBoxedTool has an AdjustProc to handle window changes and readjust the scrollbar.

You can associate an AdjustProc with a window that was created using **Tool.Create** by calling **ToolWindow.SetAdjustProc**:

```
[] ← ToolWindow.SetAdjustProc[wh, MyAdjust];
```

This operation supplants Tajo's AdjustProc with **MyAdjust**. Here is **MyAdjust** from **ScrollBoxedTool**:

```
MyAdjust: ToolWindow.AdjustProcType =
  BEGIN
  SELECT when FROM
    before = > NULL;
    after = >
        BEGIN
        clientBox, vBox, hBox: Window.Box;
        vWindow, hWindow: Window.Handle;
        [clientBox, vWindow, vBox, hWindow, hBox] ← Scrollbar.Adjust[
            window: window, box: [[0, 0], box.dims]];
        IF vWindow # NIL
            THEN Window.SlideAndSize[window: vWindow, newBox: vBox];
        IF hWindow # NIL
            THEN Window.SlideAndSize[window: hWindow, newBox: hBox];
        Window.SlideAndSize[window:data.sw, newBox: clientBox];
        END;
    ENDCASE;
  END;
```

MyAdjust will be called whenever the size of the tool window changes. It is called twice for each change; once before the window is adjusted (with the old box size as a parameter), and once after the window is adjusted (with the new box size as a parameter). Usually little, if anything, is done in the "before" call. This call exists to give applications the opportunity to save the bitmap for currently visible portions of the screen before they disappear. You might want to do this when you know you will be redisplaying a region shortly, and you know that it involves lengthy computations.

The "after" case is more interesting. If the window needs to display an image that depends upon the size of the window, the necessary calculations are done here. Also, if the window has any children, it must redistribute the regions to its children and adjust the scrollbar and subwindow within its new space. As illustrated above, this is done by calling **Scrollbar.Adjust** with the dimensions of the window box dimensions. This operation returns the new sizes for the scrollbars and the associated subwindow. **clientBox** describes the area

that the subwindow (minus the scrollbar) should actually occupy. **verticalWindow** is the window used to display the vertical scrollbar, and **verticalBox** is the region that **verticalWindow** should occupy. (**horizontalWindow** and **horizontalBox** are similar.) Finally, we call **Window.SlideAndSize** to adjust the regions these windows occupy.

21.3 Summary

There are four basic cornerstones of subwindow implementation: a DisplayProc, an AdjustProc, a NotifyProc and a ScrollProc.

A DisplayProc is responsible for painting bits in a window. The DisplayProc only displays the information, however; it does not do any calculation to figure out what should be displayed. Instead, an operation that changes something on the display updates the internal data structures, and then invalidates the appropriate portion of the screen. When it is through invalidating things, it calls **Window.Validate** to ask Tajo to update the display. Tajo then calls the DisplayProc, which updates the display to correspond with the internal data structures. A DisplayProc is always called as the result of a **Window.Validate** (or **Window.ValidateTree**); you should never call your DisplayProc directly.

The NotifyProc, the AdjustProc, and the ScrollProc are the procedures that are responsible for readjusting the internal display data structures. A subwindow's NotifyProc is called whenever TIP determines that user actions should be directed to that subwindow. The NotifyProc implements the functionality that is dependent upon mouse and keyboard actions. For example, a simple editor would define selection, insertion, and deletion. (If you like, you can do simple display actions directly from the NotifyProc.)

An AdjustProc is called whenever the size of the window changes. It is called twice: before the window is adjusted and after it is adjusted. An AdjustProc readjusts the internal data structures to reflect the new size of the window. You don't always have to write an AdjustProc; you only need one if the information you are displaying depends upon the size of the window (i.e. scaling of pictures to a new window size).

In addition to managing the information displayed in the subwindow, the AdjustProc must also divide the window among the subwindows. Thus, the AdjustProc is usually associated with the main window; this AdjustProc can then call the AdjustProcs associated with the individual subwindows (if there are any).

A ScrollProc is called whenever the user initiates a scrolling operation; it then calculates the amount of movement necessary. If this movement only involves shifting the contents of the screen, the ScrollProc also performs the actual shifting.

21.4 Exercises

21.4.1 Exercise 1: horizontal scrolling

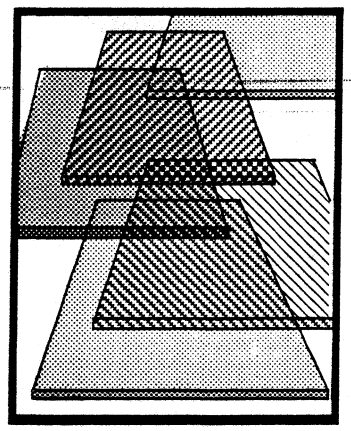
The first exercise for this chapter is to modify ScrollBoxTool so that it implements horizontal scrolling as well as vertical scrolling. This is an easy "warm-up" exercise; you should complete it before attempting the next one.

21.4.2 Exercise 2: the crossword puzzle tool

For this exercise, you are to create the "crossword puzzle tool", using BoxedTool as a starting point. Basically, you need to modify BoxedTool so that each box can contain a

number and a letter. You should set things up so that the tool does the numbering, at the request of the user. (The algorithm for numbering is that a box should be numbered if there is a black box or an edge of the grid to its left or above it.)

There are a lot of possible features for you to add to this tool; how far you take it is entirely up to you. One thing that you might want to do is modify the NotifyProc so that the mouse tracks the selection. That is, when the user presses Point and holds it down, the screen cursor should follow the mouse movements (as it does in standard text subwindows.) Currently, BoxedTool recognizes PointUp, so it does not track the selection.



Correcting compilation errors

A big part of learning a new programming language is learning its syntax. Mesa's advanced concepts make it an exciting language in which to program, and learning the syntax is but a hurdle to be cleared. To expedite your learning, we suggest you follow along with the compiling session presented here, which shows the Compiler's output for several common errors.

A.1 Discussion

The Compiler translates a Mesa program into executable code. This process consists of six passes over the source file, during any of which the Compiler may detect and report errors. Syntax errors, *per se*, are detected during one of the first passes. Later passes check for other types of errors. For example, if your program references an interface that the Compiler cannot find on your local disk, this is not a syntax violation and it will not be reported until one of the later passes.

When the Compiler finds an error, it gives an indication of the error (an error message), the position of the error in the source file, a listing of the offending line, and the fix the Compiler assumed in order to continue (when possible).

The Compiler will not go on to the next pass after it detects an error. This means that you must fix the errors and compile the program again. If it finds more errors, then you must fix them and repeat the cycle.

The Compiler also gives warning messages when it discovers a problem (or potential problem) that is not an out-right error. These do not cause the Compiler to stop at the end of the pass, and compilation proceeds.

To demonstrate this process, we have supplied three versions of a sample program. The first version contains several errors. The second version contains the corrections for those errors, but reveals some new errors that went unnoticed in the original version. The third version contains the final set of corrections and compiles successfully.

Please retrieve `SyntaxErrors1.mesa`, `SyntaxErrors2.mesa`, `SyntaxErrors3.mesa`, and `InterfaceForSyntaxErrors.mesa` from the `>AppendixA>Programs>` folder on the course's file directory.

A.2 Early-pass errors

`SyntaxErrors1.mesa` is a program intentionally laced with several common errors. It looks like this:

```
DIRECTORY
  InterfaceForSyntaxErrors USING [CreateFactorialTool FactType, tooBig];
SyntaxErrors1: PROGRAM =
  BEGIN

  Fact: PROCEDURE [n: LONG CARDINAL] RETURNS[factorial: LONG CARDINAL ← 0] =
    BEGIN
    factorialOfZero: CARDINAL = 1;
    SELECT n FROM
      0 = > RETURN[factorialOfZero]
      IN [1..12] = > RETURN[n*Fact[n - 1]
      ENDCASE = > RETURN[InterfaceForSyntaxErrors.tooBig]
    END; --of procedure Fact

  --mainline code
  InterfaceForSyntaxErrors.CreateFactorialTool[Fact];
  END
```

Use Command Central to compile `SyntaxErrors1.mesa`. You can see the results in the Compiler log, which is displayed in the bottom subwindow of Command Central. (The log displayed in Command Central is stored on your disk as `Compiler.log`.)

The text below is from the Compiler log with italic annotations added. The number in square brackets following each error message is the character position in the source file where the error was found. If you load the source into a window, you can use the **Position** command to scroll the file to the error. (After you have edited the file these positions will be a little off, since they refer to the position of the error *before* you began editing.)

```
Mesa Compiler 11.1 of 24-Sept-84 11:45:20
17-Dec-84 16:48:42

Command: SyntaxErrors1
  InterfaceForSyntaxErrors USING [tooBig];
  ↑ Syntax Error [198]
Text inserted is: ,
```

Error 1: Items in DIRECTORY *clauses must be separated by commas.*

```
    IN [1..12] = > RETURN[n*Fact[n - 1]
    ↑ Syntax Error [433]
Text inserted is: ;
```

Error 2: The line preceding this one must be terminated with a semicolon because it is in a SELECT *statement. Note that the Compiler displays the line* following *the line that is erroneous; 433 is the character position of the i in* IN.

```
    ENDCASE = > RETURN[InterfaceForSyntaxErrors.tooBig]
    ↑ Syntax Error [424]
Text inserted is: ]
```

Error 3: Not enough brackets enclose the RETURN *statement on the preceding line.*

```
END;
    ↑ Syntax Error [633]
Text deleted is:  ;
Text inserted is:  ..
```

Error 4: *The last line in a program must be* **END** *followed by a period. (The Compiler inserts two periods. The extra one doesn't hurt, and used to be a good idea on an old version of the Compiler.)*

```
SyntaxErrors1.mesa aborted
4 err, time: 5
```

Note: *The compilation did not proceed to completion, but was cancelled at the end of the pass. No object file was produced.*

A.3 Later-pass errors

`SyntaxErrors2` is a modified version of `SytnaxErrors1` that fixes the problems in `SyntaxErrors1`, but introduces some new problems. `SyntaxErrors2` splits the interface into two pieces (`InterfaceForSyntaxErrors` and `InterfaceForSyntaxErrors2`) just so that it can illustrate more errors. Here is the code for SyntaxErrors2:

```
DIRECTORY
  InterfaceForSyntaxErrors USING [tooBig],
  InterfaceForSyntaxErrors 2 USING [CreateFactorialTool, FactType];
SyntaxErrors2: PROGRAM  =
  BEGIN

  Fact: PROCEDURE [n: LONG CARDINAL] RETURNS[factorial: LONG CARDINAL ← 0] =
    BEGIN
    factorialOfZero: CARDINAL = 1;
    SELECT n FROM
      0 = > RETURN[factorialOfZero]
      IN [1..12] = > RETURN[n*Fact[n - 1]
      ENDCASE = > RETURN[InterfaceForSyntaxErrors.tooBig]
    END; --of procedure Fact

  --mainline code
  InterfaceForSyntaxErrors2.CreateFactorialTool[Fact];
  END
```

Compile `SyntaxErrors2.mesa`. Here is another annotated compiler log:

```
Mesa Compiler 11.1 of 24-Sept-84 11:45:20
17-Dec-84 16:50:42

Command: SyntaxErrors2
InterfaceForSyntaxErrors cannot be opened, at [133]:
DIRECTORY
```

Error 5: *The interface mentioned in the* **DIRECTORY** *clause,* `InterfaceForSyntaxErrors.bcd`*, is not on your local volume and cannot be opened by the Compiler. To fix this, compile* `InterfaceForSyntaxErrors.mesa`.

CreateFactorialTool must come from an imported interface, at SyntaxErrors2[616]:
InterfaceForSyntaxErrors2.CreateFactorialTool[Fact];
Error 6: **CreateFactorialTool** *must be imported by your module. To fix this, add an* **IMPORTS** *list that includes* **InterfaceForSyntaxErrors2.**

Fact has incorrect type, at SyntaxErrors2[616]:
InterfaceForSyntaxErrors2.CreateFactorialTool[Fact];
Error 7: *The procedure we are passing to* **CreateFactorialTool** *is of incorrect type.* **CreateFactorialTool** *expects* **n** *to be a* **CARDINAL** *but our procedure defines it as a* **LONG CARDINAL.**

tooBig is not valid as a field selector, at Fact[524]:
ENDCASE = > RETURN[InterfaceForSyntaxErrors.tooBig];
Error 8: *This error could mean one of two things: the symbol was not included in the* **USING** *clause; or the Compiler could not open the interface in which the symbol is defined. Since we already know that the Compiler could not open the interface* **InterfaceForSyntaxErrors,** *the latter case applies. This error will go away when you have compiled* **InterfaceForSyntaxErrors.**

warning: FactType is never referenced, at [145]:
InterfaceForSyntaxErrors2 USING [CreateFactorialTool, FactType],
Error 9: *The Compiler is warning you that you have included* **FactType** *in your* **USING** *clause, but never used it.*

SyntaxErrors2.mesa aborted
4 err,1 warn, time: 4
Note: *Again, the compilation did not proceed to completion, but was terminated at the end of the pass. No object file was produced.*

A.4 Successful compilation

`SyntaxErrors3.mesa` is a version of the program with all the previously found errors corrected. Remember to first compile `InterfaceForSyntaxErrors.mesa` and then `SyntaxErrors3.mesa`. The Compiler log should show no errors, like this one:

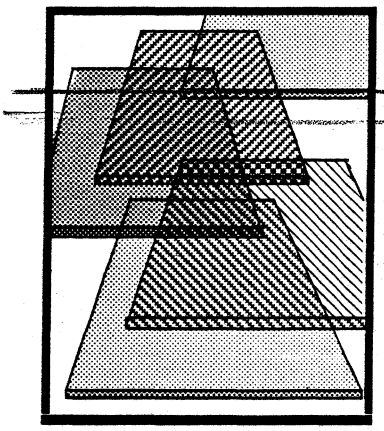
Mesa Compiler 11.1 of 24-Sept-84 11:45:20
17-Dec-84 16:51:13

Command: SyntaxErrors3
SyntaxErrors3.mesa
lines: 24, code: 47, links: 1, frame: 1, time: 21
Note: *This compilation* was *successful. The object file* `SyntaxErrors3.bcd` *was produced.*

A.5 Let the Compiler help you

The Compiler helps you find syntactic and semantic errors, and often helps expose logical errors as well. Thus, you can view the Compiler as an aid to better programming.A final piece of advice, though: don't stare at erroneous source code for too long trying to find a syntax error. Ask one of your colleagues to take a look. Sometimes, an *outside* person can spot a syntax error more quickly than the person who wrote the code, because the code is fresh to the outsider.

B



Using the Debugger

Muddling around in appendices can be a frustrating experience. You're not likely to find any interesting material in them...appendices, by nature, are pretty boring. However, this is your lucky day, because this appendix has some hot stuff in it. This chapter introduces you to some of the more important features of the debugger.

B.1 Discussion

Your goal for this chapter (and the next two) should be to become familiar with the debugger so that you can take advantage of its power and debug programs quickly. To help you achieve this familiarity, this chapter concentrates on using the interpreter and setting breakpoints; appendices C and D will introduce some other debugging techniques.

In this chapter, we assume that you are familiar with the debugger's user interface. If you are not, you should read the section of the debugger chapter of the *XDE User's Guide* that discusses the user interface.

B.1.1 The interpreter

CoPilot has a built-in interpreter that enables you to evaluate Mesa expressions. The interpreter allows you to display and re-assign variables (simple or complex), dereference pointers, call procedures, and convert types. The interpreter handles a subset of the Mesa language; mostly you'll be making assignment statements and simple queries of variables. You invoke the interpreter by typing a space at the beginning of a line in the debugger, followed by whatever it is you want to interpret.

B.1.2 Sample debugger session

In this sample debugging session, you'll use the interpreter to assign variables, call procedures, and dereference pointers.

Retrieve **MiscProcs.mesa** and **MiscProcs.bcd**. Run **MiscProcs/d** from CommandCentral in CoPilot (The "/d" means that the module will be loaded in Tajo but not started). Once you have returned to CoPilot, follow along with the script presented below (underlined text in the script indicates something you type in to CoPilot; *italic* text indicates commentary):

B.1.2.1 Using the interpreter

```
1. >SEt Module context: MiscProcs   --type a return at the end of the line
2. > A;j                            --type a space, then "A;j" followed by a return
3. > Factorial[5]                   --type a space at the beginning of lines 3 through 9
4. > 170B?
5. > A;j
6. > A[3] ← 30; A[7] ← 70
7. > A
8. > InterChange[3,7]
9. > A
```

On line 1, you tell the debugger the module that you are interested in. For the debugger to be able to find the code for a procedure, you must explicitly tell it where to focus its attention. "SEt Module context" tells the debugger that subsequent commands pertain to the specified module. (When you are using the debugger, you may find that the debugger will occasionally tell you that it can't find a specified symbol. This usually indicates that the debugger is looking in the wrong place. Use the SEt Module context command to refocus the debugger's attention.) In this case, the debugger found the specified module in Tajo. When it can't find a specified module, the debugger will issue an error message.

On line 2, you use the interpreter to examine the variables "A" (an array) and "j" (a long cardinal). Their values will look unfamiliar; they aren't initialized because the module hasn't been started (which explains the warning "{*global frame number*} is not started").

On line 4, you make an interpreted call to the procedure Factorial in module MiscProcs. You pass the necessary parameter (in this case, a cardinal), and it returns an answer (the factorial of your number). This number may be in octal format (denoted by the "B" after the number). (Note again that you are warned that the module had not been started).

On line 4, you interpret the number "170B" by typing a "?" after the number. The reason for this is that the answer returned by procedure Factorial was in octal format; this allows you to see the answer in octal, hexidecimal, decimal, ascii, and other formats (See the *XDE User's Guide* to find out how to set the default format using the Options Window.)

On line 5, you re-examine the variables "A" and "j"; you should find that they have been initialized ("A" initialized to all zeroes and "j" being set by the Factorial call to 120). The global variables in the module were initialized when you made your call to Factorial.

On line 6 you stuff new values into the 3rd and 7th spots in the array "A."

On line 7, you examine "A" to make sure that the array contain your new values.

On line 8, you make an interpreted call to the procedure InterChange, which interchanges the two values in the spots in the array that you specified (in this case, the third and seventh spot).

On line 9, you re-examine "A" to check that the values for the 3rd and 7th spot have been interchanged.

Your debugger should look similar to the one on the next page:

```
Nub: "MiscProcs.bcd" loaded.
>SEt Module context: MiscProcs
> A; j
112560B is not started!
A = (13)[1, 2, 6400B, 17B, 20B, 20156B, 67564B, 20146B, 67565B, 67144B, 20141B,
67144B, 20000B]
112560B is not started!
j = 46406504418
> Factorial[5]
112560B is not started!
170B
> 170B?
170B = 78X = 120 = 'x = 7:8
> A; j
A = (13)[0,0,0,0,0,0,0,0,0,0,0,0,0]
j = 170B
> A[3] ← 30; A[7] ← 70
> A
A = (13)[0,0,0,30,0,0,0,70,0,0,0,0,0]
> InterChange [3, 7]
> A
A = (13)[0,0,0,70,0,0,0,30,0,0,0,0,0]
>
```

B.1.2.2 Setting breakpoints

Here is another series of debugger commands for you to type into your debugger window:

```
10. >Break Entry procedure: Factorial Breakpoint #1.
11. > Factorial[4]
12. >Display Stack
      >v
      >q
13. >CLear Break #: 1
14. >Break Xit procedure: Factorial Breakpoint #2.
15. >Proceed [Confirm]
16. >Display Stack
      >v
      >s
      >n
      >v
      >n
      >v
      >n
      >v
```

```
      >n
      >v
      >q
17. >ATtach Condition #:2 Condition: number = 4
18. >Proceed [Confirm]
19. >Display Stack
      >v
      >q
20. >Proceed [Confirm]
```

On line 10, you set a breakpoint at the entry to procedure Factorial. Breakpoints are numbered sequentially throughout a debugging session, even if you remove earlier numbered settings. The debugger allows you to set a breakpoint at entry or exit of a procedure; you can also set a breakpoint at a specific line by selecting that line in the source code and invoking the Break command in the DebugOps menu.

On line 11, you again make an interpreted call to Factorial.

On line 12, after returning to CoPilot at your breakpoint, you display the first element in the run-time stack. This command provides the name of the current procedure, the local frame number, the program counter state, the current module, and the global frame number. Once you are inside Display Stack, there is a new set of single character sub-commands available. The sub-commands you can now use include: variables, parameters, next, source and quit. You will continue in stack viewing mode until you ask to quit or hit <DELETE> to the prompt. (The full set of subcommands is in the *XDE User's Guide*). When you quit out of Display Stack mode, the full set of commands will once again be available. **Note**: the variable "(anon)" is the unnamed return parameter for that procedure.

On line 13, you clear your breakpoint. It is no longer in effect.

On line 14, you set a breakpoint at the exit of procedure Factorial.

On line 15, you allow the process that was executing your procedure to proceed.

On line 16, you again display the stack, and look at some of the other entries on the stack. The stack is loaded with calls to Factorial since Factorial is a recursive procedure.

On line 17, you make breakpoint #2 conditional (it will only be effective if n = 4.)

On line 18, you proceed again.

On lines 19 and 20, after hitting the breakpoint, you display the stack to check the value of your variables at that point. Since they look good, you should proceed again and the procedure will return with the correct answer.

Your debug log should now look like the one on the next page.

```
>Break Entry procedure: Factorial Breakpoint #1.
> Factorial[4]
Break #1 at entry to Factorial, L: 65150B, PC: 117B (in MiscProcs, G: 114340B)
>>Display Stack
Factorial, L: 65150B, PC: 117B (in MiscProcs, G: 114340B) >v
 number = 4
 (anon) = 14110B
 >q
>>CLear Break #: 1
>>Break Xit procedure: Factorial Breakpoint #2.
>>Proceed [Confirm]
Break #2 at exit from Factorial, L: 3020B, PC: 163B (in MiscProcs, G: 114340B)
>>Display Stack
Factorial, L: 3020B, PC: 163B (in MiscProcs, G: 114340B) >v
 number = 0
 (anon) = 1
 >s at exit. Factorial: PROC [number: CARDINAL] RETURNS [LONG CARDINAL] = BEGIN
 >n
Factorial, L: 14120B, PC: 146B (in MiscProcs, G: 114340B) >v
 number = 1
 (anon) = 3745400001B
 >n
Factorial, L: 45364B, PC: 146B (in MiscProcs, G: 114340B) >v
 number = 2
 (anon) = 4000002B
 >n
Factorial, L: 14110B, PC: 146B (in MiscProcs, G: 114340B) >v
 number = 3
 (anon) = 15207200003B
 >n
Factorial, L: 65150B, PC: 146B (in MiscProcs, G: 114340B) >v
 number = 4
 (anon) = 4
 >q
>>ATtach Condition #: 2, condition: number = 4
>>Proceed [Confirm]
Break #2 at exit from Factorial, L: 65150B, PC: 163B (in MiscProcs, G: 114340B)
>>Display Stack
Factorial, L: 65150B, PC: 163B (in MiscProcs, G: 114340B) >v
 number = 4
 (anon) = 24
 >q
>>Proceed [Confirm]
24
```

B.1.2.3 Dereferencing pointers

Try the following in the debugger:

```
21. >CLear All Breaks
22. > MakeLinkedList[4]
23. > headNode
24. > headNode↑
25. > headNode.next↑
26. > headNode.next.next↑
27. > headNode.next.next.next↑
```

On line 21, you clear any currently set breakpoints.

On line 22, you make a call to the procedure MakeLinkedList, which creates a singly-linked list (the size of the linked list is specified by the caller; in this case, the size is 4.) The global variable **headNode** is a pointer variable that acts as the head of this linked list.

On line 23, you examine the value of **headNode** and find the *address* of the record that it points to. You know that it's an address by the up-arrow that follows the returned number.

On line 24, you *dereference* the pointer "headNode" and examine the contents of its referent. Notice the field "next" and the fact that it contains a number with an up-arrow after it. This field points to the next element in the linked list. (The other field in this record, "str," is a **LONG STRING** of length = 1 and maxlength = 1 [hence the "(1,1)"] that contains the text "D".)

On line 25, you examine the contents of what the "next" field points to. Notice that you do not have to type "headNode↑.next↑", only "headNode.next↑", due to the *auto-dereferencing* feature of the Mesa language (and interpreter).

On line 26, you examine the contents of what the next "next" field points to.

On line 27, you look at the final element in the linked list. Notice that the "next" field for this last element is **NIL**.

The last part of your debugger should look similar to the following.

```
> MakeLinkedList[4]
> headNode
headNode = 4021731B ↑
> headNode ↑
[str:4021736B ↑ (1,1)"D", next:4021742B ↑ ]
> headNode.next ↑
[str:4021747B ↑ (1,1)"C", next:4021753B ↑ ]
> headNode.next.next ↑
[str:4021760B ↑ (1,1)"B", next:4021764B ↑ ]
> headNode.next.next.next ↑
[str:4021771B ↑ (1,1)"A", next:NIL]
```

B.2 Style

There is no single technique for debugging a program, since everyone tends to develop a personal style. Yet, there always comes a time when staring at a listing of a program doesn't give you a clue as to what has gone wrong. In this case you may want to use a debugging technique that under most circumstances can track down any bug. This method is called *binary search debugging* and it has only one requirement: there must be an absolutely reproducible test case that causes the same problem every time. If the error can be made to occur on demand, then debugging using binary search is very straightforward.

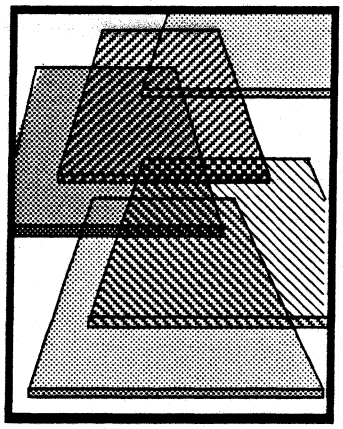
Like the standard searching algorithm of the same name, binary search relies on splitting the search space into two parts and then determining in which half to continue the search. In the case of debugging, the binary search range (or search space) starts at a point when everything is okay and ends at a point where something is not okay. You are searching for the instant when that something changes from okay to *not* okay. To start the search, split the range in half, and set a breakpoint in the middle. Proceed and check which came first, the bug or the breakpoint? If everything is still fine at the breakpoint, then split the second half, set another breakpoint and proceed again. If the problem occurred before the breakpoint, then start the program again, setting the breakpoint in the middle of the first section. After just a few tries you will narrow things down and find the offending code.

It is important to realize that an exact split is not as important as making sure you narrow the range with each iteration. As long as the search space shrinks each time, you will eventually find the error. It is also useful to pin the problem down to a specific procedure call or a particular DO loop since this provides a very specific area to search for the problem. You should avoid setting breakpoints inside of loops or too close together since those breakpoints will occur too often without significantly shrinking the search space.

B.3 References

Chapter 24 of the *XDE User's Guide* is the reference source for information about the debugger.

Notes:



Translating uncaught signals

Unexpected events can interrupt the execution of a program. For example, suppose that you gave a program input that was not in the expected range. The most common solution to this problem is for the programmer to write code to ensure that the input is acceptable before the body of the program is executed. However, in Mesa, you can use a mechanism called *signals* instead.

The *signal* mechanism was designed to allow you to anticipate and deal with unusual occurrences during program execution. Signals are like procedure calls, except that the code to be executed for a signal is determined dynamically. Thus, when an excception occurs, control transfers to a runtime program called the Signaller, which searches up the call stack looking for a procedure that has code to handle the exception. If no procedure has code to handle the exception, the debugger is called to inform you of the error; the signal is considered to be *uncaught* .In this appendix, you will see three examples of uncaught signals and find out what to do when you get one. Chapter 8, Signals, presents a thorough explanation of the signal mechanism, and discusses how to write programs that use signals.

C.1 Definition of terms

Signal — A *signal* is a Mesa language construct used to help handle exceptional conditions encountered during program execution. Signals are like procedures, except that the code to be executed for a signal call is determined at runtime.

Uncaught signal — An *uncaught signal* occurs when no module in the call stack handles a *signal* that has arisen. If a signal is uncaught, control transfers to the debugger.

C.2 Discussion

When you get an uncaught signal, you need to know what exception caused the signal, and how to fix the problem. This appendix discusses primarily how to determine the name of the signal that caused your problem. As you will see, however, learning the name of the signal will often go a long way toward finding out why the signal was raised. You will also find that this often gives information which allows you to prevent the signal from being raised in the future. If you cannot figure out how to prevent a signal from being raised

again, consult someone for help. In most cases, rebooting the volume in which the uncaught signal was generated will get you back working again.

C.2.1 Example A: Pre-translated uncaught signal

In this example, the uncaught signal message is already translated into a human-readable form when it first appears in the debugger. Run the program `UncaughtSignal` from Command Central. When you get to Tajo, fill in 14 in the `number=` field and invoke **CalculateFactorial!**. This will cause an uncaught signal and return control to the debugger, because the program is coded to only accept input in the range [0..12]. Once CoPilot has fully instantiated itself, take a look in the debugger window. Instead of seeing something you're used to, like

18-Dec-84 10:05:38
**** interrupt ****

you should see something like

18-Dec-84 10:05:38
**** uncaught SIGNAL InputTooBig[input: 16B] (in module UncaughtSignalImpl, G: 71404B).*

This is what a fully translated uncaught signal looks like in the debugger. You can read the message as follows: when you used 14 (16 octal) as input, the signal **InputTooBig**, which is declared in the module `UncaughtSignalImpl`, was raised and not handled by the program. Instead, the signal was sent to the debugger so you can handle the problem.

Instead of using a boolean to check if the input is too big, this program uses a signal to handle the exception. However, for the purposes of this example, it deliberately mishandles the signal. Since you will not learn how to handle signals until Chapter 8, you cannot fix the code in the program, however.

C.2.1.1 Why the debugger translated the signal

You may have noticed a delay after the ****uncaught SIGNAL* part of the debugger's message was displayed. CoPilot was making an automatic interpret call to determine the signal's value (i.e. its symbol name and parameter). As with any interpret call, CoPilot is successful only if it can find the symbol table for the interpreted symbol. Remember, a symbol resides in the symbol table of the module in which it was declared and in the configuration file in which it was bound. For CoPilot to be able to translate this signal either `UncaughtSignalImpl.bcd` or `UncaughtSignal.bcd` (or both) has to be on your search path.

When CoPilot can't find a module whose symbol table includes the signal, it does not have enough information to translate the signal and you'll see something like

18-Dec-84 10:08:07
**** uncaught SIGNAL [50501B] msg = ?[5367B] (in module SecondUncaughtSignalImpl, G: 71404B)*

when you get to the debugger. The first bracketed number is the system's representation of the signal. In Mesa, all signals are represented as unique numbers. This is fine for the system, but it doesn't give you much information. The next two sections explain different

ways to make sure that the appropriate symbol table is on the local volume so that CoPilot can translate the signal. First, however, a few words on how to get out of the debugger:

C.2.1.2 Returning from an uncaught signal

There are three ways to leave the debugger:

- Type P. The debugger will fill in "roceed" and ask you to confirm that you want to proceed. This command is comparable to a return from a procedure. When you Proceed, you will return to Tajo and continue execution from where you left off. An example of when you might do this is when the signal is only a warning message. Generally, though, you will not be able to proceed from an uncaught signal.

- Type Q. The debugger will fill in "uit" and ask you to confirm that you want to quit. Quitting a program is similar to hitting the ABORT key except that it raises a signal for notification. Chances are that this signal will also go uncaught and you will find yourself back in the debugger. If this happens, try Quitting a second time. This almost always gets you back to the volume in which the original uncaught signal was raised.

- Reboot the volume that generated the uncaught signal. This is appropriate if you wish to get back and work on the volume that crashed with the uncaught signal and proceeding or quitting did not work. You will often need to reboot if the problem occurred during the use of someone else's program. If you are testing your own code, and you have learned how to handle signals (chapter 8) you will most likely want to skip proceeding and quitting. Instead, you can alter your program to handle the signal or fix the problem, and then you can reboot the volume by running from CommandCentral.

C.2.2 Retranslating an untranslated uncaught signal: Method 1

Run the program **SecondUncaughtSignal**. Once you are in Tajo, try again to calculate factorial with **number** = 14. As with the first example, the debugger will be called. But this time, instead of a translated signal, you'll see an untranslated one. Follow the script below to learn one way to translating an untranslated signal. (Underlined text in the script indicates something you type to the debugger; *italic* text indicates commentary):

```
18-Dec-84 10:09:41
*** uncaught SIGNAL [50501B] msg = ?[5367B] (in module
  SecondUncaughtSignalImpl, G: 71404B)
```

This is the message form of an untranslated signal. This message tells you that a signal whose symbol's value is 50501B has gone uncaught. This symbol is declared in the module SecondUncaughtSignalImpl. CoPilot attempted to translate this signal when you entered the debugger but it was unable to find the symbol table for the module SecondUncaughtSignalImpl. You must supply the symbol table module and then ask CoPilot to reinterpret the signal.

```
>Current context
  Module: SecondUncaughtSignalImpl, G: 71404B, L: 21134B, PSB: 77B
  Configuration:SecondUncaughtSignal
```

One way is to get the symbol from the symbol table that was generated when the configuration file that includes SecondUncaughtSignalImpl was bound. To do this you

need to know the name of that configuration file. To get this information, you ask CoPilot to tell you the current context (the current referencing environment). It tells you that the configuration that was executing when the signal was raised is SecondUncaughtSignal. But you have this file on your volume. Why didn't CoPilot find the symbol in its symbol table?

When binding a configuration, it is common to separate the symbols (by putting them into a separate file) from the rest of the configuration file. In this case, the symbols are in a separate file (which is not on your local disk), so CoPilot was unable to translate the signal. To make the symbol information available, you have to retrieve the file SecondUncaughtSignal.symbols. Do this now. You should find the file on the release course directory (subdirectory >AppendixC>Symbols). Once you have retrieved the file, you can ask CoPilot to try again to do the translation.

```
>ReDisplay swap reason
SIGNAL InputTooBig[input: 16B] (in module UncaughtSignalAgainImpl,
G: 71404B)
```

Redisplay Swap Reason tells the debugger to make another attempt at translating the signal. This time, since you have retrieved the symbols file, the debugger is able to tell you the name of the signal. You can now see that the problem was caused, as in the last example, by our program's failure to handle the signal InputTooBig.

C.2.3 Retranslating an untranslated uncaught signal: Method 2

You've now seen one method for translating a signal. We will use this same example to illustrate the other method for translating a signal. Delete **SecondUncaughtSignal.symbols**. By doing this you are removing the means for CoPilot to translate the signal. Try retranslating the signal. You should see

```
18-Dec-84 10:11:59
*** uncaught SIGNAL [50501B] msg = ?[5367B] (in module
SecondUncaughtSignalImpl, G: 71404B)
```

Symbols reside in the module in which the symbol was declared, as well as in any bound configuration that includes that module. In the previous example, you translated the signal using the configuration's symbols. In this example you'll use the module in which the signal was declared. Retrieve SecondUncaughtSignalImpl.bcd from the release directory (subdirectory >AppendixC>Symbols), and then do another Redisplay Swap Reason.

```
>ReDisplay swap reason
SIGNAL InputTooBig[input: 16B] (in module UncaughtSignalAgainImpl,
G: 71404B)
```

C.2.4 If you want more information

When you have translated a signal, you will often find that you need more information to determine what went wrong. For example, you may want to know which procedure raised the signal and why, or you may want to see where you were in your program when the signal was raised. In method 1 you saw that when CoPilot instantiated itself, it was set to the context that resulted in the uncaught signal. By doing a Display Stack you can step through the chain of calls that resulted in the uncaught signal.

```
> Display Stack
Fact, L: 4070B, PC: 50B (in SecondUncaughtSignalImpl, G: 112004)  >s
ENDCASE => <> SIGNAL InputTooBig[n];
  >n
CallToCalculateFactorial, L: 12674B, PC: 442B (in ToolFactorialImpl,
G: 112034B)   >s
<>SELECT fact ← FactorialProc[toolData.number] FROM

>n
No symbols for L: 5764B, PC: 1343B (in FormSWsB, G: 32734B)   >n
No symbols for L: 5300B, PC: 1147B (in FormSWsJ, G: 32734B)   >n
No symbols for L: 5030B, PC: 3126B (in TIPMatchImpl, G: 32734B)  >n
No symbols for L: 60530B, PC: 2776B (in TIPMatchImpl, G: 32734B)
>n
No symbols for L: 4030B, PC: 564B (in TIPMatchImpl, G: 32734B)   >n
No symbols for L: 21434B, PC: 25B (in TajoControl, G: 32734B)   >n
No previous frame!
```

The first entry displayed is the code that actually raised the signal. By doing successive n(exts) you can see the procedures that had a chance to handle the signal but did not. While stepping through the stack, you may find that CoPilot is unable to translate a symbol's value into its procedure name (the last six calls in the stack shown above). CoPilot is unable to translate these symbols for the same reason it was unable to translate the signal. It cannot find the corresponding symbol table for the value it wants to translate. By retrieving the symbols (the object file for the module mentioned in the parentheses) and causing CoPilot to retranslate the line, by either starting a new Display Stack or doing a j(ump) of 0 lines (see the XDE User's Guide if you do not know how to do this), the procedure name will be displayed. If you also retrieve the source file for the module you can look at the code that is making the call or causing the problem. Normally, however, you will not need the symbols or the source for these modules, which are part of the system. You will usually find the error in your own code.

C.3 Summary

Each time the debugger is called with an uncaught signal, CoPilot immediately tries to translate the signal's value into its name. CoPilot is successful only if it can locate a symbol table that includes the signal's symbol. There are three files that can contain this symbol table:

- The implementation module where the signal was declared. The name of this module appears as part of the untranslated message.

- The symbols file that was generated when the implementation module in which the signal was declared was bound into a configuration file. To find out the name of this file, you need to do a Current context command to discover your context. The symbols file is the name returned by this command with `.symbols` appended.

- If the symbols were not copied out at bind time into a separate `.symbols` file, they will be in the configuration file. In this case, if the signal comes from your program, you only need to retrieve the configuration file you ran. If it comes from somewhere else, it is easier to retrieve the implementation module mentioned in the untranslated message (because you know its name).

If CoPilot is unable to translate the signal, retrieve one of these files and ask CoPilot to reinterpret the signal (Redisplay Swap Reason).

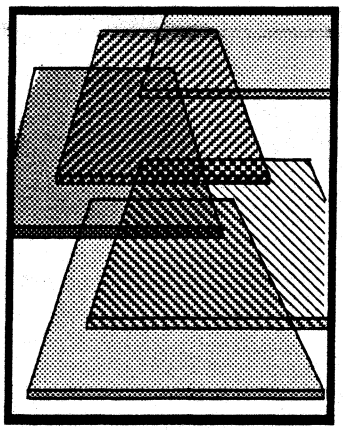
We have not discussed how to handle signals that are under program control. You will learn more about that in Chapter 8, Signals. For now, you just need to understand that when signals are not recognized by the program, the debugger is called so you can do something appropriate. Expect to see more uncaught signals. They are not a cause for panic. They are undesirable, but you should assume the attitude that they are aids to your efforts to write error-free programs.

C.4 References

A condensed version of the information provided in this chapter can be found in "Interpreting Signals," *XDE User's Guide*, §III.6.5

A discussion of the rationale for the signal mechanism in Mesa and several more examples of its use are provided in *Mesa: A Designer's User Perspective*, §4.

D



Debugging an address fault

This appendix examines a debugging session that helps to determine the reason for an address fault. Code involving pointers never seems to work on the first try, so you should get a little practice at debugging the most common pointer programming problem: the address fault.

In this appendix we use several debugger commands without fully explaining them. If we use a command that is unfamiliar to you, you should look it up in the *XDE User's Guide*.

D.1 Definition of terms

@	The @ is the prefix "address of" operator. @x generates a reference to the expression x.
↑	is the Mesa dereferencing operator. ↑ is the opposite of @.
Address Fault	An *address fault* occurs when you attempt to dereference an invalid address.

D.2 Discussion

The source code for the sample program you will run is in the module **AddressFaultImpl**. The first thing that you should do is retrieve **AddressFaultImpl.mesa** and **AddressFaultImpl.bcd** from the course directory, if you don't already have them on your local disk.

Inside **AddressFaultImpl**, the **AppendText** procedure has been written in an attempt to make the code more efficient. A good way of doing this is to create a **TextBody** that has room for four extra characters to begin with, and then just copy the characters into the **text** array. With this technique, only one **TextBody** need be created and only one freed, no matter how many characters are being appended.

Unfortunately, the program **AddressFault** does not work. To start testing this program, do the following:

1. Run **AddressFault.bcd** from CommandCentral.

2. When Tajo is booted, bring up the Executive and type

```
AddressFault Moo unto ye!
```

This will invoke the debugger. To find out what to do next, follow the debugging session below.(You might want to have `AddressFaultImpl.mesa` loaded in an open file window while viewing the Debugger log.)

D.3 Start of the debugging session

```
19-Dec-84 10:46
*** Address Fault, PSB: 137B, at 415116B, in AppendChar, L: 14134B,
PC: 126B (in AddressFaultImpl, G: 114274B) ***
```

This message is displayed in the debugger whenever a program crashes due to an address fault. Let's look at the call stack.

```
>Display Stack
AppendChar, L: 3604B, PC: 153B (in AddressFaultImpl, G: 71444B) >s ELSE
BEGIN <>onto.text[onto.length] ← from; onto.length ←onto.length + 1;
END;
```

We crashed while trying to execute the line **onto.text[onto.length] ← from**. *Let's look at the variables involved in that statement to see if they are correct.*

```
 > onto
onto = 403733B↑
 > onto↑
4703325B↑
 > onto↑↑
[length:340B, maxlength:54B, text:(0)[]]
```

A **TextBody** *with a* **length** *greater than its* **maxlength**! *Not only that, but the* **length** *seems awfully large. Undoubtedly this* **TextBody** *has caused the address fault. Let's take a look at its contents to see if they can give us a clue as to what has gone wrong. Note that the debugger claims that the* **text** *field has a zero length and no contents. This is not really true, but the debugger thinks this is true because the* **text** *array was declared as a* **PACKED ARRAY [0..0)**. *The debugger, looking at the declaration, decides that the* **text** *array is an array with a fixed size of zero, and refuses to show us anything inside the array. However, you can look inside the array yourself by getting the address where it starts and then reading the contents of memory directly.*

```
 > @onto↑↑.text
4703327B↑
 >q
>AScii Read: 4703327B, n(10): 340B
```

This command reads n bytes starting at the address specified and displays them as ASCII characters.

```
ye! ▮▮▮▮ye!▮▮ffl▮▮▮▮moo unto ye!▮ffi▮▮▮▮moo unto ▮▮ ▮▮▮▮moo unto▮
▮▮▮▮moo ▮▮▮'▮▮▮▮¿ !Invalid Address [4703400B]
```

Well, some of the contents of this array look familiar. But somehow the array is being overwritten so that the last few elements of the array lie outside the legal address space. This explains why you get an address fault when you try to access **onto.text[onto.length]**. *Since you had to stop displaying the stack in order to do an Ascii Read, display the rest of the stack now to try to get more information about the bug.*

```
>Display Stack
AppendChar, L: 3604B, PC: 153B (in AddressFaultImpl, G: 71444B) >n
Reverse, L: 3730B, PC: 564B (in AddressFaultImpl, G: 71444B)    >s
<>AppendChar[onto: @reversed, from: Space];
 > reversed↑
[length:340B, maxlength:54B, text:(0)[]]
```

Looks like the **TextBody** *of* **reversed** *was already wrong when* **AppendChar** *was called.*

```
 >n
Main, L: 6060B, PC: 707B (in AddressFaultImpl, G: 71444B) >n
No symbols for L: 21350B, PC: 717B (in AddressFaultExports, G: 71420B) >n
No symbols for L: 11004B, PC: 5763B (in ExecsA, G: 32770B) >n
No symbols for L: 6640B, PC: 2122B (in ExecImpl, G: 32744B) >n
No previous frame!
```

Well, the address fault occurred because a **TextBody** *record is being damaged. You can't tell where or when the damage occurs, though. Let's run the program again with the same input, but this time with some breakpoints in strategic locations. Hopefully, we will be able to see the trouble as it develops.*

D.4 Running then setting breakpoints

In order to set breakpoints inside **AddressFaultImpl**, you need to have **AddressFaultImpl.bcd** and **AddressFaultImpl.mesa** on your CoPilot search path and you need to have the **AddressFault** program loaded in Tajo. Run **AddressFault.bcd** from CommandCentral, and then interrupt into CoPilot after Tajo is booted but **before** typing anything to the Tajo executive. Running a program from CommandCentral will load it in Tajo, thus enabling you to set breakpoints.

```
19-Dec-84 11:04
*** interrupt ***
```

Here we have just SHIFT-STOPped into CoPilot from Tajo after hitting Run! in CommandCentral. The module **AddressFaultImpl** *has been loaded in Tajo, so you can set breakpoints in it.*

```
>SEt Module context: AddressFaultImpl
```

Set the module context so that the debugger knows where to look for the procedure that you are going to set entry and exit breakpoints in. If you didn't set the module context, the debugger would have no way of knowing what module to look in to find the **AppendText** *procedure.*

```
>Break Entry procedure: AppendText Breakpoint #1.
>Break Xit procedure: AppendText Breakpoint #2.
```

In addition to the entry and exit breakpoints, set two other breakpoints inside the body of AppendText, one each before the lines **onto ← AddressFaultDefs.FreeTextNil[onto]** *and* **onto ← s.**

```
>List Breaks
1 -- Break at entry to AppendText (in AddressFaultImpl, G: 71444B).
2 -- Break at exit from AppendText (in AddressFaultImpl, G: 71444B).
3 -- Break in AppendText (in AddressFaultImpl, G: 71444B).
  <>onto ← AddressFaultDefs.FreeTextNil[onto];
4 -- Break in AppendText (in AddressFaultImpl, G: 71444B).
```

```
<>onto ← s;

>Proceed [Confirm]
```

Having set your breakpoints, you now can go back to Tajo and run the program again with the same input as before.

```
Break #1 at entry to AppendText, L: 4024B, PC: 170B (in AddressFaultImpl,
G: 71444B)
```

You have returned to the debugger by encountering the breakpoint at the entry to **AppendText**.

```
>Display Stack
AppendText, L: 4024B, PC: 170B (in AddressFaultImpl, G: 71444B) >p
 onto = 4703332B ↑
 from = 4703337B ↑
 startingAt = 11B
 endingAt = 14B
  > onto↑
[length:0, maxlength:3, text:(0)[]]
  > from↑
[length:14B, maxlength:16B, text:(0)[]]
```

All of these **TextBody** *records look fine.*

```
  >n
Copy, L: 3714B, PC: 360B (in AddressFaultImpl, G: 71444B) > copy↑
[length:0, maxlength:3, text:(0)[]]
  > copy
copy = 4703332B ↑
```

This is the same as the **onto** *variable in* **AppendText**, *as it should be.*

```
  > s
s = 4703337B ↑
```

This is the same as the **from** *variable in* **AppendText**, *as it should be.*

```
  > s↑
[length:14B, maxlength:16B, text:(0)[]]
  > @s.text
4703341B ↑
  >q
>AScii Read: 4703341B, n(10): 14B
moo unto ye!
```

This is the text you typed in on the command line. So far, everything looks fine. Continue execution of the program.

```
>Proceed [Confirm]
Break #2 at exit from AppendText, L: 4024B, PC: 341B (in AddressFaultImpl,
G: 71444B)
```

Now you've reached the exit of **AppendText**, *and can check to see if the* **TextBody** *records, which looked fine when you entered* **AppendText**, *look right now.*

```
>Display Stack
AppendText, L: 4024B, PC: 341B (in AddressFaultImpl, G: 71444B) > onto↑
```

```
[length:3, maxlength:3, text:(0)[]]
 >@onto.text
4703334B↑
 >q
>AScii Read: 4703334B, n(10): 3
ye!
```

Well, everything looks fine here. Maybe the bug isn't in **AppendText** *after all. Let's let the program run some more to find out.*

```
>Proceed [Confirm]
Break #1 at entry to AppendText, L: 4024B, PC: 170B (in AddressFaultImpl,
G: 71444B)
```

Here you are back at the beginning of the **AppendText** *procedure, which must have been called again.*

```
>Display Stack
AppendText, L: 4024B, PC: 170B (in AddressFaultImpl, G: 71444B) >p
onto = 4703325B↑
from = 4703337B↑
startingAt = 4
endingAt = 10B
 >onto↑
[length:4, maxlength:4, text:(0)[]]
 >from↑
[length:14B, maxlength:16B, text:(0)[]]
 >q
```

Once again, everything looks fine here. Let's continue execution.

```
 >Proceed [Confirm]
Break #3 in AppendText, L: 4024B, PC: 300B (in AddressFaultImpl, G: 71444B)
```

Ahh. This time the **onto TextBody** *had to be re-allocated in order to hold the new text, so, you've hit your other breakpoints inside of* **AppendText**.

```
>Display Stack
AppendText, L: 4024B, PC: 300B (in AddressFaultImpl, G: 71444B) >s
<>onto ← AddressFault.FreeTextNil[onto];
 >s
s = 4703316B↑
 >s↑
[length:10B, maxlength:10B, text:(0)[]]
```

This **s** *is the string allocated inside of the* **THEN** *block. Looks good.*

```
 >onto
onto = 4703325B↑
 >onto↑
[length:4, maxlength:4, text:(0)[]]
 >@onto.text
4703327B↑
```

onto *looks fine*

```
 >@from.text
4703341B↑
```

```
 > @s.text
4703320B↑
 >n
Reverse, L: 3620B, PC: 575B (in AddressFaultImpl, G: 71444B) >s
<>AppendText[
```

So, the call to **AppendText** *was made from* **Reverse**. *Let's see what the string to be appended onto looks like right now.*

```
 > reversed
reversed = 4703325B↑
```

This is the same as **onto** *inside of* **AppendText**, *as it should be.*

```
 > reversed↑
[length:4, maxlength:4, text:(0)[]]
 >q
```

Before you let the program continue, check up on the contents of all the **text** *fields. Do this by using the ".text" values from above.*

```
>AScii Read: 4703327B, n(10): 4
ye!
```

This is the **text** *of* **onto** *and* **reversed**.

```
>AScii Read: 4703341B, n(10): 14B
moo unto ye!
```

This is the **text** *of* **from**.

```
>AScii Read: 4703320B, n(10): 10B
ye! unto
```

This is the **text** *of* **s** *inside of the* **THEN** *block of* **AppendText**.

```
>Proceed [Confirm]
Break #4 in AppendText, L: 4024B, PC: 303B (in AddressFaultImpl, G: 71444B)
```

Now you have hit the breakpoint set just after the **onto TextBody** *has been freed by a call to* **FreeTextNil**.

```
>Display Stack
AppendText, L: 4024B, PC: 303B (in AddressFaultImpl, G: 71444B) >s
<>onto←s;
 > onto
onto = NIL
```

As expected, **onto** *is* **NIL**. *Since* **onto** *is the same as* **reversed** *inside the* **Reverse** *procedure,* **reversed** *should be* **NIL** *now, too.*

```
 >n
Reverse, L: 3620B, PC: 575B (in AddressFaultImpl, G: 71444B) >s
<>AppendText[
 > reversed
reversed = 4703325B↑
```

Uh oh! It's not **NIL**!

```
 > reversed↑
[length:340B, maxlength:54B, text:(0)[]]
```

Sure enough, the **TextBody** *of* **reversed** *is now garbled. Now you can see what went wrong. In* **AppendText, onto** *starts out pointing to the same* **TextBody** *record that* **reversed** *points to. Then we free the record that* **onto** *points to, and set* **onto** *to point at a new* **TextBody** *record. But* **reversed** *is never changed! So,* **reversed** *still points to its original referent;, that is,* **reversed** *points at the* **TextBody** *that has been freed. No wonder* **reversed** *is garbage! This bug is a typical pointer bug: only the local copy of the pointer (***onto***) is being changed in the procedure call; the actual pointer variable passed to the procedure is not modified.*

So we have tracked down the bug: **AppendText** needs to be rewritten the way **AppendChar** is written, so that it takes a pointer to a **AddressFaultDefs.Text** variable as a parameter. That way, any changes made to **onto↑** inside of **AppendText** will affect **reversed** inside of **Reverse**.

D.5 Summary

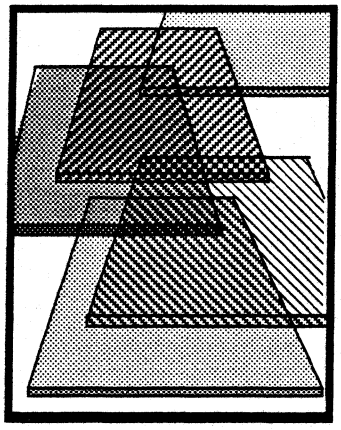
This debugging session provided an example of how to debug programs with pointer problems. Using the Mesa operators ↑ and @, you can dereference a pointer and find the address of a variable. Using the Ascii Read command, you can convert a sequence of values into their corresponding characters. This is a very important command when you need to see the value of a string variable.

By setting breakpoints, you were able to stop execution at key points and examine the values of variables. This was very important in this example since you are not sure when **reversed** was garbled. The address fault occurs after it was garbled. By setting breakpoints at important places, you can determine when **reversed** changed, and therefore have a clue as to why.

The techniques you learned here can be applied to debugging other address faults. Since address faults are common, you should find these techniques very helpful.

Notes:

E



Answers to Questions

This appendix contains answers to the questions posed in the chapters.

E.1 Chapter 2: Interfaces

E.1.1 Question 1

These modules can be compiled in many orders. The only constraint on compilation is that any interface that is used by a module (one that appears in the module's **DIRECTORY** clause) must be compiled prior to the compilation of that module. Generally, this means that interfaces must be compiled before implementations. When one interface uses another, the same rule applies.

In this problem, **Program1** must be compiled after **Interface1**, **Interface2**, and **Interface3**. **Interface1** depends on no other modules and can therefore be compiled at any time. **Program2** cannot be compiled until **Interface1** and **Interface2** have been compiled. **Interface2** must be compiled after **Interface1**. **Program3** gets compiled after all three interfaces have been compiled. **Interface3**, like **Interface1**, has no dependencies and can therefore be compiled at any time.

You must run them in the order **Program2, Program2, Program3**. Remember, you don't have to run interfaces, since they don't contain any executable code, and you must run implementations before the clients that use those implementations.

E.2 Chapter 4: Pointers

E.2.1 Question 1

Examine the procedure declarations. **GetNextValue1** can't possibly work: it doesn't have a pointer to the variable that is passed as the **nextValue** parameter. Thus, since it can't change the value of its parameter, and since it doesn't return a value, **GetNextValue1** has no way to communicate the next piece of input data to whoever calls it.

GetNextValue2 takes a pointer to a CARDINAL variable as its parameter, so the call **DataIn.GetNextValue2[@i]** is the correct one for this procedure.

The expression **@i** is not a variable, and so it cannot appear on the left hand side of the assignment operator. Thus the statement **@i ← DataIn.GetNextValue3[]** makes no sense. **i ← DataIn.GetnextValue3[]** is the correct call.

The call to **GetNextData3** is the best one from the viewpoint of good style, since it does not require the use of the **@** operator. As discussed in §4.2.4, you should avoid passing around addresses of variables whenever possible.

E.2.2 Question 2

Procedure **AverageData1** is a ponderous no-op. Not only must it copy a large record into the local variable **dataToAverage**, none of the changes that are made to the parameter **dataToAverage** will be visible to a caller of **AverageData1**.

AverageData2 handles parameters correctly. Since it is passed a pointer to a record, only a two-word pointer need be copied into **dataToAverage**. Consequently, an actual procedure call executes much more quickly than does a call to **AverageData1**. Even better, the changes made to **dataToAverage.data** by **AverageData2** are visible to a caller.

E.3 Chapter 5: Dynamic Storage Allocation and Management

E.3.1 Question 1

The **OurFreeNode** procedure is an invitation to disaster. It appears to be nice shorthand that allows us to both free a **Node** and set the **NodePtr** to NIL in one operation. Unfortunately, only the local variable **nodeToFree** of **OurFreeNode** gets set to NIL and not the **NodePtr** passed as a parameter; the actual parameter will end up pointing to deallocated storage. The correct way to write **OurFreeNode** is as a function that returns NIL:

```
OurFreeNode: PROCEDURE [nodeToFree: NodePtr] RETURNS [NodePtr] =
  BEGIN
  Node.FreeNode[nodeToFree];
  RETURN [NIL];
  END;
```

E.4 Chapter 8: Signals

E.4.1 Question 1

The CONTINUE will branch to the statement **Write["Commands completed."L];**. Because the signal is defined in an ENABLE clause, the continue will cause a branch to the statement *following the one in which the signal was raised.* In this case, the outermost BEGIN-END block serves as that statement, so the continue will branch to the first statement after the one containing the catch phrase.

E.4.2 Question 2

```
Sig1: SIGNAL = CODE;
x: CARDINAL ← 0;
...
FOR counter: INTEGER IN [1..3] DO                    1
   ENABLE                                            5
      Sig1 = > RETRY;                                1
   <statement 1>                                     3
   IF counter = 2 THEN                               2
      BEGIN                                          1
      ENABLE                                         3
         BEGIN                                       4
         Sig1 = > <statement 2>;                     5
         UNWIND = > x ← 1;                           1
         END;                                        5
      <statement 3>;
      IF x = 0 THEN
         SIGNAL Sig1;
      <statement 4>;
      END;
ENDLOOP; ...
```

E.4.3 Question 3

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO                   1
      BEGIN                                          1
      ENABLE                                         2
         Sig1 = > LOOP;                              3
      <statement 1>;                                 4
      IF counter = 1 THEN
         SIGNAL Sig1;
      <statement 2>;
      END;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

E.4.4 Question 4

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO                    1
    BEGIN                                             3
    ENABLE                                            1
      Sig1 = > CONTINUE;                              2
    <statement 1>;                                    3
    IF counter = 1 THEN                               4
      SIGNAL Sig1;
    <statement 2>;
    END;
  <statement 3>;
ENDLOOP;
<statement 4>; . . .
```

E.4.5 Question 5

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO                    1
    BEGIN                                             4
    ENABLE
      Sig1 = > EXIT;
    <statement 1>;
    IF counter = 1 THEN
      SIGNAL Sig1;
    <statement 2>;
    END;
  <statement 3>;
ENDLOOP;
<statement 4>; . . .
```

E.4.6 Question 6

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO                    1
  ENABLE                                              1
    Sig1 = > LOOP;                                    2
  <statement 1>;                                      3
  IF counter = 1 THEN                                 4
    SIGNAL Sig1;
  <statement 2>;
  <statement 3>;
ENDLOOP;
<statement 4>; . . .
```

E.4.7 Question 7

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO                1
   ENABLE                                         1
      Sig1 = > CONTINUE;                          2
   <statement 1>;                                 3
   IF counter = 1 THEN                            4
      SIGNAL Sig1;
   <statement 2>;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

E.4.8 Question 8

```
Sig1: SIGNAL = CODE;
...
Proc1: PROCEDURE =
   BEGIN
      SIGNAL Sig1;
   END;
...
IF TRUE THEN                                      1
   BEGIN                                          2
   ENABLE                                         3
      Sig1 = > RESUME;                            4
   <statement 1>;
   Proc1[!Sig1 = > CONTINUE];
   <statement 2>;
   Proc1;
   <statement 3>;
   END;
<statement 4>;
```

E.4.9 Question 9

```
Sig1: SIGNAL = CODE;
...
BEGIN                                                    1
ENABLE                                                   2
  Sig1 = > RESUME;                                       4
<statement 1>;                                           5
IF TRUE THEN
  BEGIN
  ENABLE
    Sig1 = > GOTO TheEnd:
  <statement 2>;
  SIGNAL Sig1;
  <statement 3>;
  EXITS
    TheEnd = > <statement 4>;
<statement 5>;
EXITS
  TheEnd = > <statement 6>;
END;     ...
```

E.4.10 Question 10

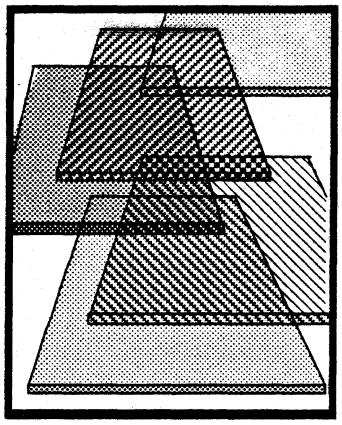
Proc1[0] calls **OtherProc[0]**, (in block b), which calls **StillOtherProc[0]** ,which raises **sig1**. **Catch phrase-4** sees the signal first, and we have assumed that it rejects it. Next **Catch phrase-3** is presented with the signal, but it rejects it implicitly since there is no catch-case for **Sig1**. Next **Catch phrase-2** catches **Sig1**, and rejects it (by assumption). Finally **Catch phrase-1** catches **Sig1** and jumps to the label **punt**.

Beore executing this jump, the Signaller raises **UNWIND** in every catch phrase that had rejected **Sig1**: **Catch phrase-4,** then **-3** then **-2** (but not **Catch phrase-1**, because it didn't reject the signal.). Thus, the program will execute the statements in the order given below:

```
Stmt1
Stmt2
Stmt3
Stmt4
ERROR Sig1          -- Sig1 is raised in StillOtherProc
Catch-case-7        -- in Catch phrase-4
Catch phrase-3      -- Does not catch Sig1
Catch-case-4        -- In Catch phrase-2
GOTO punt           -- In Catch phrase-1/Catch-case-1. UNWIND is raised
Catch-case-9        -- In Catch phrase-4
Catch-case-6        -- In Catch phrase-3
Catch phrase-2      -- Does not catch UNWIND
Stmt6
```

E.4.11 Question 3

b will get the value **FALSE**. Within the catch phrase (**Sig = > c2 ← c1: RESUME**), the variables **c1** and **c2** refer to variables local to **Sig**, and not to **Proc**'s variables.



Training Liaison/Mentor Information

Trainers are an important part of the Mesa Course. Although the course is designed to be completed with a minimum of outside assistance, students are sure to have questions. This appendix provides information to those individuals who will be asked these questions.

Each XDE customer site will have a designated XDE *training liaison* with certain XDE training responsibilities. He will assign other, more experienced, Mesa programmers to act as *training mentors* for students who are beginning the Mesa Course. (It is entirely possible to have a student who is working on the latter part of the course act as a training mentor for one just beginning the course.) If a student has a question about an explanation in the course or difficulty with a programming assignment, the student should ask his training mentor for help. If the mentor cannot answer a question, he should refer it to the training liaison. If the liaison cannot answer a question, he should refer it to **XDESupport.osbunorth@Xerox.arpa** for customers "outside" Xerox corporation or **XDEConsultants:All Areas** for internal users. **Only the training liaison should submit questions to XDESupport**.

The training liaison is the owner of the local <MesaCourse> file drawer. Initially certain subdirectories of this file drawer will be private. The training liaison will determine access privileges appropriate for the installation. He is the person to contact if you wish access to a protected folder.

F.1 The machine

This is version 12.0 of the Mesa Course. It assumes that you are using a Dandelion or Daybreak processor running the Sequoia release (12.0) of the Xerox Development Environment with Tajo installed on a normal volume, CoPilot serving as a debugger for the volume on which Tajo is installed, and a User.cm that is set up for this configuration.

A possible volume configuration for students using a Shugart 42MB disk is:

Volume	Size	Type	Notes
CoPilot	25,000	Debugger	
Tajo	10,000	Normal	Tajo boot file
Scavenger`	3,900`	Normal	
User	26,376	Normal	ViewPoint boot file

Use Othello's Describe Physical Volume command to compare the student's volume structure with this one. This configuration is only a suggestion; other configurations are possible. There must be at least a Tajo and CoPilot volume, however.

F.1.1 User.cm entries

Certain sections of the User.cm should be tailored for the Mesa Course. A crucial entry is ClientVolume, which should correspond to the volume with an installed Tajo boot file (typically the Tajo volume). The significant entries are:

```
[Executive]
CompilerSwitches: eub-j
BinderSwitches: ec
ClientVolume:Tajo                    --Volume with installed Tajo bootfile.
CodeLinks: FALSE
UseBackground: TRUE
```

It is helpful to have editor symbiotes on file windows in the CoPilot volume so that the student can easily set breakpoints and position files to correct compilation errors. We suggest the following:

```
[FileWindow]
Menu:Load Edit Save Store Reset Empty Position Break Split Time Trace Destroy
```

Whenever a student starts the course, *check the User.cm.*

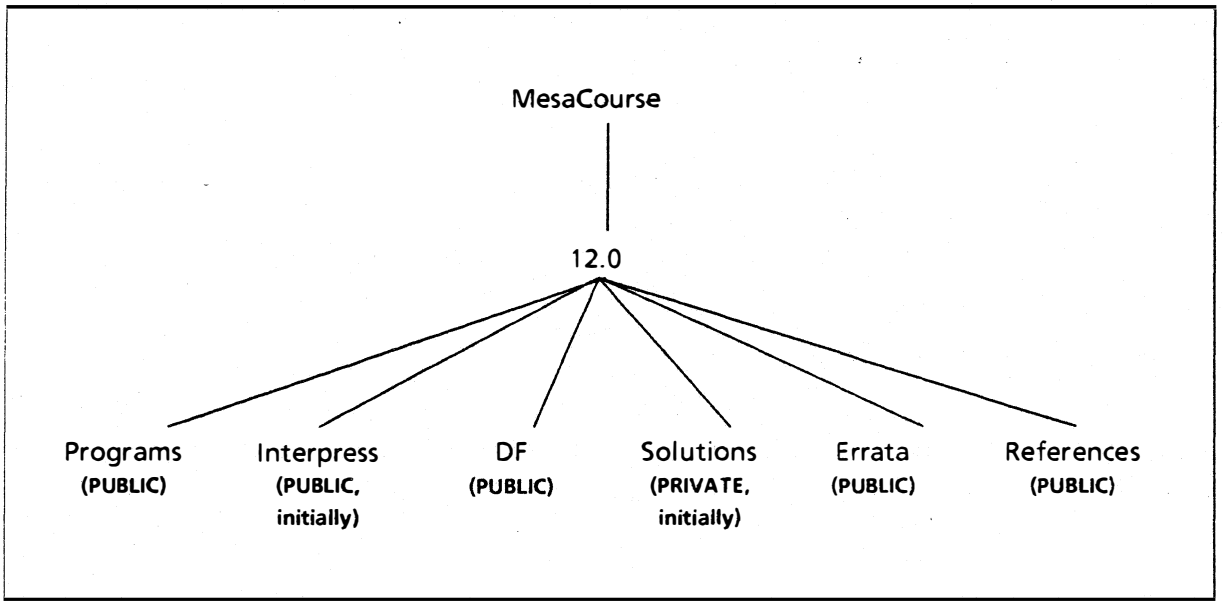
F.2 Location of course materials

Contact your local XDE training liaison to determine the location of the files that comprise the Mesa Course. Training liaisons within Xerox corporation should copy the course's release directory from [McKinley:OSBU North Xerox] <MesaCourse> onto a local file server.

The training mentor should make sure that the following system interfaces are on the student's machine:

Ascii.bcd, Environment.bcd, Exec.bcd, Format.bcd, FormSW.bcd, Heap.bcd, Inline.bcd, MFile.bcd, MStream.bcd, Process.bcd, Put.bcd, Stream.bcd, String.bcd, System.bcd, Time.bcd, Tool.bcd, ToolWindow.bcd, UserTerminal.bcd, and Window.bcd.

F.3 The Course's directory structure



The Mesa Course Directory Structure

Interpress masters for the course text are stored electronically in the folder [CustomerNSFileServer]<MesaCourse>12.0>Interpress>. Within that folder there there is an interpress master for each chapter. A student with proper authorization can print copies of the course from these folders if necessary (Universities may want to protect this folder, other sites would not). Bound copies of the Mesa Course should be available from your local documentation support group.

The programs discussed in the chapters are stored in the [...] <...> ...> Programs>ChapterName(ChapterNumber) folder for each chapter. The student should retrieve all files from this folder before starting a chapter, e.g., retrieve all the files in [CustomerNSFileServer]<MesaCourse>12.0>Programs>Interfaces(2) before starting Chapter 2.

Solutions to programming exercises are stored in the [...] <...> ...> Solutions> folder. The XDE training laison will decide who has access rights to this folder: it may be read protected (universities using the course may have reason to protect this folder; other users may not).

The two papers mentioned below can be found in [...] <MesaCourse> 12.0> References>

The Mesa Course is still under development, and we would appreciate your comments and corrections. We apologize for any inconveniences caused by inconsistencies or inaccuracies that have escaped our current review. Please check on [...] <...>...> Errata> for any update information.

F.4 References

The Mesa Course refers students to various XDE release documents and two papers. The release documentation is available from your local technical support group. It includes:

Xerox Development Environment User's Guide
Mesa Programmer's Manual
Pilot Programmer's Manual
Filing Programmer's Manual (contained in the Services 8.0 Programmer's Guide)
Mesa Language Manual

The papers can be found in the [...]<MesaCourse>12.0>References> folder. They are:

Impact of Mesa on System Design by Hugh Lauer
Mesa: A Designer's User Perspective by James Mitchell

F.5 Errors in course materials

Report all errors not acknowledged in the [...] <...>...> Errata> folder to your training liaison. He can forward them to XDESupport.osbunorth@Xerox.Arpa.

Internal Xerox students with access to the System Software Adobe data base can submit AR's directly There is a Mesa Course subsystem under the Documentation system for System Software

Glossary

Abstract machine: An *abstract machine* is a set of functions, provided by some combination of hardware and software, that forms the underpinnings of a system sitting above. Pilot, for example, provides an abstract machine that runs on a variety of machines.

Abort: To *abort* is to terminate a process abnormally, such as by using the **ABORT** key.

Accelerator: An *accelerator* is an easier or faster way of doing a common operation.

Active window: *An active window* is a window that is ready for interaction with the user. (Compare **Tiny window**, **Inactive window**.)

Actual parameters The variables and expressions that are supplied to the procedure to replace the formal parameters are called *actual parameters*.

Actual procedure: An *actual procedure* is a procedure initialized so that it's meaning (defined by it's body) cannot change. Actual procedures (as opposed to procedure variables) cannot be assigned to.

Address Fault: An *address fault* occurs when an attempt is made to reference an illegal address.

Adjective: An *adjective* is an identifier constant from an enumerated type, used to select one of the alternatives in a variant record.

ADJUST: *ADJUST* is the right mouse button, generally used to extend selections and for accelerators.

ALT B: *ALT B* is a boot button used to do alternate booting, such as booting from another device.

Argument: An *argument* is a piece of data upon which an operation is performed. For example, the argument to a **DELETE** command is the video-inverted text to be deleted.

Asynchronous call: An *asynchronous call* is a procedure call that initiates an operation, but returns control to its caller without waiting for the operation to complete.

Atom: An *atom* is a Mesa primitive providing a unique identifier in a global naming space. An atom has a property list associated with it.

Authenticate: To *authenticate* is to establish that a user or client is who he, she, or it claims to be. (See **Credentials**.)

Background process: A *background process* is a process that receives machine resources only if higher priority processes are idle or blocked.

BCD: A binary configuration description (*BCD*) is a compiled and possibly bound Mesa module, sometimes called an object file. (See **Configuration description**.)

Bind: To *bind* is to combine object modules into one executable unit (called a configuration) by resolving intermodule references.

Bitmap: A *bitmap* is a representation of an image as a sequence of bits, each of which represents the intensity of a point in the image. The display hardware and microcode convert a bitmap to a displayed image.

Block: A *block* is a construct used to associate declarations with statements. The names so declared have significance only within the block. The block is the scope of these names which are said to be local to the block. Since a block may appear as a statement, scopes may be nested.

Boot: To *boot* is to load and start a system on a machine whose main memory has undefined contents. The Dandelion can be booted by pressing the **B RESET** boot button. ("Boot" is short for "bootstrap", which is in turn short for "bootstrap load".)

Boot button: A *boot button* is a maintenance panel button used to boot the processor. The Dandelion has two buttons, labelled B RESET and ALT B.

Boot file: A *boot file* is a file that contains a bootable program.

Built-in types: The Mesa built-in types incluce serveral numeric types (**INTEGER, LONG INTEGER, CARDINAL, LONG CARDINAL, REAL** , and **NATURAL**) a type for logical values (**BOOLEAN**) a type for individual character values (**CHARACTER**), and a type for sequences of characters (**STRING**).

CALL DEBUG: *CALL DEBUG* is the action of pressing **SHIFT-ABORT** together, which transfers control to the debugger.

Call Stack: The *call stack* is a Mesa processor data structure containing a frame for each procedure invocation that has not yet returned. The call stack is ordered with the most recent invocation first.

Caret: The *caret* is a blinking pointer that indicates the type-in point.

Catch Phrase: A *catch phrase* is a Mesa construct that establishes code to catch one or more signals.

Channel: A *channel* is a low-level procedural interface for accessing and driving I/O devices.

Chord: To *chord* keys or buttons is to push them down at the same time, as when chording the mouse buttons.

Clearinghouse: A *clearinghouse* is a server for locating named objects in a distributed environment.

Click: To *click* a mouse button is to press down on it and let it up.

Client: A *client* is a program (as opposed to a person) that uses the services of another program or system. (See **User**.)

CoCoPilot: *CoCoPilot* is the name usually given to the debuggerDebugger volume used to debug CoPilot.

Command Central: *Command Central* is a tool for compiling and binding programs on a development volume and running them on a client volume.

Command file: A *command file* is a file containing commands, especially Executive commands.

Compile: To *compile* is to translate a source file into an object file (BCD).

Condition variable: A *condition variable* is a Mesa construct by which processes wait for or provide notification of an event. A condition variable is associated with a monitor.

Configuration description: A *configuration description* (*config* for short) is a C/Mesa source file that tells the Binder how to combine modules into a configuration. A *configuration* file is the bound code of one or more modules.

Constant: A *constant* is a name, an associated value, and a scope for the association. Within this scope, the value associated with the name may not change.

Continue: To *continue* a signal is to resume program execution at the statement following the one to which the catch phrase belongs. Thus, control is resumed in the procedure where the signal was caught, not the procedure that raised the signal.

CoPilot: *CoPilot* is the name of the debugger volume used to debug Tajo and other normal volumes. The boot file that contains the debugger, used on both the CoPilot and CoCoPilot volumes, is also called CoPilot.

Courier: *Courier* is the Network Systems remote procedure call facility. A remote procedure call causes a procedure to be executed in another machine over a network.

Create date: The *create date* is the date and time that the information contained in a particular version of a particular file was created. Since create dates are accurate to the nearest second, the pair <file name, file version's create date> serves as a unique identifier for the contents of a file.

Credentials: *Credentials* are the identification, such as name and password, presented by a client to a service for authentication.

Critical section: A *critical section* is a portion of a program in which only one process can be executing at a time. In Mesa, access to critical sections is arbitrated by monitors.

Current selection: See **Selection**.

Cursor: The *cursor* is an icon that tracks the mouse position: moving the mouse moves the cursor. The system may change the cursor shape to provide feedback about what it is doing.

Dandelion: The *Dandelion* is a processor supporting both the Xerox Development Environment and the Office System products.

Dangling Pointer: A *dangling pointer* is a pointer to an invalid memory location.

Data type: A *data type* is a set of objects and a set of operations on those objects that create, build-up, destroy, modify and pick apart instances of the objects. A data type may be either directly described in a declaration that uses it, or it may be referenced by a type name introduced in a type declaration.

Deactivate: To *deactivate* is to make a tool inactive, removing all windows associated with the tool from the display and discarding the state of the tool.

Debugger context: *A context* in the debugger is a referencing environment that determines the meaning of symbols. The current context identifies one of the executing processes (within a particular module within a particular configuration) that the debugger will use in interpreting other commands. For example, the current context determines which variables in which procedure invocations to use in evaluating an expression.

Debugger volume: A *debugger volume* is a logical volume that contains a debugger and is used to debug normal volumes. (See **normal volume, debuggerDebugger volume**.)

debuggerDebugger volume: A *debugger-Debugger volume* is a logical volume that contains a debugger and is used to debug debugger volumes.

Dereference: To *dereference* a pointer is to follow the pointer through one level of indirection toward the value it is referencing.

Device: A *device* is a peripheral unit (almost always hardware) that is separately accessible through its own channel.

Device driver: A *device driver* is a program that translates channel requests into physical device actions.

Directory: A *directory* is a named subdivision of a logical volume. A directory can in turn be divided into subdirectories. The top-level directory on a volume has the same name as the volume.

Discrimination: A *discrimination* statement provides access to the fields in the variant part of a variant record, based on the value of the tag.

Disk page: A *disk page* is a contiguous 256-word region of disk storage.

Dynamic allocation: *Dynamic allocation* acquires storage during program execution.

Dynamic variables *Dynamic variables* are generated by a special procedure (NEW) that yields a pointer or reference value that subsequently serves in place of a name to refer to the variable.

Error: An *error* is a Mesa language construct similar to a signal, except that a signal can return to where it was raised (like a procedure), whereas an error cannot.

Ethernet: The *Ethernet* is a communications system for carrying digital data among locally distributed computer systems. The Ethernet is implemented as a 10 megabit/second multi-access packet-switched network.

Exception: An *exception* is an unusual event that programs must be prepared to handle, such as I/O error. In Mesa, exceptions are associated with signals. (See **Signal**.)

Executive: The *Executive* is a tool with a simple teletype interface for loading and running Mesa programs. Some commands are already available.

Expression: *Expressions* are constructs describing rules of computation for evaluating variables and for generating new values by the application of operators.

Export: To *export* is to implement all or part of an interface for use by other modules. (See **Import, Interface**.)

Face: A *face* is a Mesa interface that embodies part of the abstract machine defined in the *Mesa Processor Principles of Operation*.

File: A *file* is a sequence of data pages located on some physical device and containing some common grouping of information. Files may be local or remote.

File extension: The *file extension* is the (possibly null) portion of a file name that follows a period. By convention some extensions indicate the format of the data in the file (although not all tools use default extensions consistently). Some common extensions are:

archiveBcd	Mesa object program module
bcd	Mesa object program module
boot	boot file
cm	command file
config	a C/Mesa source file (configuration description file)
doc	documentation file
errlog	error message file
log	history of program actions
mesa	Mesa source module
symbols	Mesa symbol table in binary format (for debugging)
tip	tip tables

File handle: A *file handle* is a data structure that identifies a file being accessed.

File service: The *file service* is a set of network facilities that provide file storage and retrieval. A machine implementing this service is called a file server.

File Tool: The *file tool* is a tool that allows the user to store and retrieve files on remote file servers.

File type: A *file type* is a file attribute provided by Pilot for the use of higher level software.

File window: A *file window* is a window whose main subwindow is a text subwindow for displaying and editing the contents of a file. A contiguous group of pages within a file into which a map unit is mapped is also called a file window.

Filter: A *filter* is a software entity that implements a stream for transforming, buffering, and manipulating data.

Font: A *font* is a set of characters of one size and style. Fonts come in different families (such as Classic or Gothic), different sizes (such as 10 point or 14 point), and different styles (such as plain, bold, or italic). This sentence is in Classic 10 plain font.

Formatter: The *Formatter* is a tool that transforms Mesa source files into a standard format for display.

Form subwindow: A *form subwindow* is a system-provided subwindow type that supports invoking commands and displaying or changing the values of data.

Frame: A *frame* is a PrincOps data structure allocated for the variables and internal data structures of a module or procedure while it is executing. Module frames are called *global frames*, and procedure frames are called *local frames*. Since Mesa supports recursion, there may be several frames for a given procedure.

Frame pack: A *frame pack* is a swap unit produced by the Packager that contains the global frames for a collection of modules.

Gateway: A *gateway* is a processor serving as a forwarding link between separate Ethernets. (See **Router**.)

Germ: The *germ* is the Pilot program that loads a boot file into memory and starts it executing. The germ also creates outload files and implements communication with remote debuggers. The germ is so named because it is the first program executed when a boot button is pushed.

Head: A *head* is an implementation of a face for some processor or device. A collection of heads provides a processor-independent environment in which Pilot and its clients execute.

Heap: A *heap* is a system-designated area of virtual memory used for dynamic allocation of storage. Heaps, which provide more automatic management of storage than zones, support the Mesa language operators **NEW** and **FREE**, which allocate and deallocate storage dynamically.

Herald Window: The *herald window* is a tool (usually a wide, short window at the top of the screen) that displays information about the state of the environment, has a menu to boot logical volumes, and allows tools to display messages.

Hint: A *hint* is information that is usually accurate and is easy for a program to use. A program can detect when a hint is inaccurate and find the truth in some other (usually less efficient) way.

Icon: An *icon* is a small picture on the display representing some entity.

Implementation module: An *implementation* or **PROGRAM** *module* is a program that codes (*implements*) and makes available to clients (*exports*) items in an interface. One implementation module can export all or part of one or several interfaces, and an interface can be jointly implemented by several implementation modules.

Import: To *import* is to make accessible to one module the procedures and variables exported by other modules. (See **Exports**.)

Input Focus: The *input focus* is the window to which keyboard commands and characters are sent. The input focus contains the type-in point.

Interface: An *interface* is a formal contract between pieces of a system that describes the services to be provided. A provider of these services is said to implement the interface; a consumer of them is called a client of the interface.

Interface module: An *interface* or **DEFINITIONS** *module* defines types, variables, constants, procedures, and signals, thus specifying the services to be provided by its implementation modules.

Interlisp: *Interlisp* is an interactive version of LISP with a large library of facilities.

Internet: An *internet* is a collection of networks mutually accessible via internet routing services.

Interpress: *Interpress* is a print file format standard.

Lister: The *Lister* produces listings of information in object files, such as dates of the interface modules used and cross references for procedure calls.

Literal: A *literal* is a constant whose value is given by its sequence of symbols.

Log file: A *log file* is a file containing a history of program actions. For example, compiler.log

contains summary statistics for each source file compiled by the most recent compile command.

Logical volume: A *logical volume* is a partition of storage for client files, including system data structures for manipulating those files. A physical volume is divided into one or more logical volumes. Each logical volume is largely protected from actions in other logical volumes.

Loophole: *Loophole* is a Mesa operator that coerces a value of one type into another type, thus circumventing Mesa's strong typing.

Machine: A *machine* is a hardware configuration consisting of a processor, main memory, and peripheral devices. Workstations and servers are machines.

Main data space: The *main data space (MDS)* is a subspace of virtual memory that provides the local execution environment for Mesa programs and holds the implicit Mesa data structures. The MDS can contain up to 64K words. Thus, only short (16-bit) pointers are needed to address any part of the MDS.

Maintenance panel: The *maintenance panel* is the front panel on a Mesa processor with boot buttons, a numerical display for maintenance panel codes, and an on/off switch.

Maintenance panel codes: *Maintenance panel codes* (MP codes) are three or four-digit status and error codes that indicate the current processor state.

Map: To *map* is to associate a region of virtual memory with a file window so that the contents of the file window appear to be the contents of the region.

Map unit: A *map unit* is a contiguous group of virtual memory pages that is the principle unit for allocating, mapping, and swapping virtual memory.

Menu: A *menu* is a list of available commands or data chosen by mouse selection. More than one menu may be associated with a tool window or subwindow or with the unused portion of the display.

Mesa: The *Mesa* language is a Pascal-like, strongly typed, system programming language that forms the basis of the Xerox Development Environment.

Message subwindow: A *message subwindow* is a system-provided subwindow type for posting messages (including errors).

MLM: The *Mesa Language Manual* describes the Mesa programming language.

Mode: A *mode* is a special state of a system in which user actions have special meaning.

Modeless: A *modeless* user interface is one that is free of modes. In such an interface, pressing a particular key always has essentially the same effect.

Module: A *module* is a Mesa program. A *source module* is a text file that can be compiled into an *object module*. There are three kinds of source modules: **PROGRAM**, **MONITOR**, and **DEFINITIONS**.

Monitor: A *monitor* module is a Mesa module that controls access to shared data, thus synchronizing interactions among processes.

Monitor invariant: A *monitor invariant* is a logical assertion about the state of monitored data whenever the monitor is unlocked, (i.e., exited). Every monitor has a monitor invariant.

Monitor lock: A *monitor lock* is essentially a hidden data item associated with each monitored record or program that indicates when a process has entered and not yet exited a critical section.

Mouse: The *mouse* is a pointing device that allows the user to direct the attention of the machine to a particular point on the display. A mouse usually has two buttons, **POINT** and **ADJUST**.

Mouse-ahead: Analogous to type-ahead, *mouse-ahead* is mouse clicks made before a program has asked for them.

Movable boundary: A *movable boundary* is a horizontal line with a small box on its right end that divides a window into subwindows and is

used to change the relative heights of adjacent subwindows.

MPM: The *Mesa Programmer's Manual* describes the interfaces that provide the framework and run-time system for writing Mesa programs in the Xerox Development Environment.

Name: A *name* (or identifier) is a sequence of alphabetic and numeric characters beginning with an alphabetic character. Identifiers in Mesa can be up to 256 characters long; character case is significant in Mesa identifiers.

Name lookup: *Name lookup* is the process of mapping a character string to a network address.

Name stripe: The *name stripe* is a rectangular region at the top of a window. It is usually black, with the window's name and other information in white.

Network: A *network* is a communication medium, such as an Ethernet, known to routers by a unique network number.

Network address: A *network address* consists of a network number, host number, and socket number. The network number identifies a network anywhere in the world. The host number identifies a machine, independent of which network it is on. A socket number identifies a particular socket on that host. (See **Socket**.)

Network stream: A *network stream* is a stream representing a connection over a network between two processes, often on different machines.

Node: A *storage node*, or *node* for short, is a block of allocated storage, often with a record structure.

Normal volume: A *normal volume* is a logical volume used to run client programs. (See **debugger volume**, **debuggerDebugger volume**.)

Notifier: The *Notifier* process in Tajo handles user actions, informing each tool of each user action directed to it. Because tools perform their work in the Notifier process, further user input is not acted on until the first operation is fnished.

NS: Network Systems (*NS*) are the Xerox standard protocols for using the Ethernet.

Othello: *Othello* is a utility for managing Pilot volumes, including initializing physical and logical volumes, installing and invoking boot files, and scavenging logical volumes.

Outload file: An *outload file* is a snapshot of the volatile state of a system (essentially the contents of memory and registers). Outload files are used by the debugger. (See **World-swap**.)

Package: To *package* is to group components of modules together into swap units to try to improve use of real memory.

Packet: An NS *packet* is the unit of information in the internet. A packet consists of a header and data, and has a maximum length of 576 bytes. The information in the header is specified by the Internet Datagram Protocol.

Page: A *page* is a block of 256 words of information in either virtual memory or a file. The page is the basic addressable unit of a file.

Path name: The *path name* is the complete name of a file, including the file server or workstation and directory or subdirectory on which it is stored. A path name is usually denoted by a machine name in square brackets followed by a directory name in angle brackets, optionally followed by one or more subdirectory names separated with right angle brackets, followed by the file name itself, such as [Iris]<Mesa>Doc>Compiler.doc.

Physical volume: A *physical volume* is the basic unit available for random access file page storage. A physical volume corresponds to a storage device, typically a disk.

Pilot: *Pilot* is the operating system for the Xerox Development Environment. Pilot provides a single-user, single-language environment including virtual memory, a large flat file system, network communication facilities, and Mesa run-time support (including concurrency facilities).

Pilot kernel: The *Pilot kernel* comprises the basic facilities of Pilot.

Pipeline: A *pipeline* is a sequence of concatenated filters that perform a series of transformations on the contents and properties of a stream.

POINT: *POINT* is the left mouse button, generally used to identify data and to invoke commands.

Pointer: A *pointer* is a data item containing the location of a value. The Mesa language has pointer types.

PPM: The *Pilot Programmer's Manual* describes the external structure and interfaces of Pilot.

Print service: A *print service* provides printing facilities, usually for files formatted in Interpress format.

PrincOps: The *Mesa Processor Principles of Operation* is a document that defines the abstract architecture of the Mesa processor. It specifies the processor's virtual memory structure, its instruction interpreter, and the Mesa instruction set. It is classified as Xerox Private Data.

Procedural abstraction: A *procedural abstraction* is a mapping from a set of inputs to a set of outputs that can be described by a specification. The specification must show how the outputs relate to the inputs, but it does not reveal or imply the way the outputs are to be computed.

Procedure: A *procedure* is comprised of four elements: its name, a list of identifiers called formal parameters, a body, and an environment.

Procedure body: A *procedure body* is a block.

Procedure environment: A *procedure's environment* consists of those variables that are declared outside of the body of the procedure, but which may be used ao altered at run-time by the procedure's statements.

Procedure results: A procedure can produce one or more values, called its *results*.

Procedure variable: A *procedure variable* is a procedure initialized in such a way that the procedure's value (body) can be changed by assignment.

Procedure statement: A *procedure statement* causes the application (invocation, call) of a designated procedure value (body) to the values of its arguments (actual parameters). Application of procedures that produce results may appear within expressions.

Process: A *process* is effectively a procedure activation that runs concurrently with its caller, allowing asychronous activities.

Processor: A *processor* is a computing engine (including its memory) in a workstation or server.

Raise: To *raise* a signal is to instruct the Signaller to look in each procedure on the call stack, starting with the most recently invoked, until it finds a procedure with a catch phrase for that signal.

Real estate: *Real estate* is any or all of the display screen.

Real memory: *Real memory* is the physical memory that holds software and data during processing.

Reference: A *reference* component of a variable identifies the area of stroage where a value will be kept.

Reject: A catch phrase *rejects* a signal when it is not prepared to resolve it. A catch phrase rejects a signal either by explicitly placing a **REJECT** statement in the code or by not specifying how to resolve the signal.

Release: A *release* is an official, consistent version of software produced and maintained by its developers.

Resume: To *resume* a signal is to return program control (and possibly values) to the statement immediately following the one that raised the signal. An **ERROR** cannot be resumed.

Retry: To *retry* a signal is to tell the Signaller to re-execute the statement containing the catch phrase.

Router: A *router* is a software package that sends packets between sockets. The path chosen by a router includes intermediate stops if the destination socket is on another network. A router that sends packets between networks is called an *internet router*.

RS-232-C: *RS-232-C* is a standard established by the Electronic Industries Association for serial binary data interchange between a machine and data communication equipment. An RS-232-C controller connects a machine to a modem, allowing data to be sent across telephone lines.

Scavenge: To *scavenge* is to check for damaged file structures and to attempt to repair them.

Scope The *scope* of a name is that part of the program text where all uses of the name are the same.

Scroll: To *scroll* is to reposition the data visible in a subwindow as though it were part of a long, continuous sheet of paper. Scrolling up, for example, moves the data near the bottom of the window toward the top.

Scrollbar: A *scrollbar* is a tall, narrow rectangle near the left border of a subwindow, used in scrolling and thumbing.

Search path: The *search path* is a sequence of directories (with subdirectories) used as prefixes to look up file names that are not fully specified; i.e., that do not start with a directory name.

Selection: The *selection* is a text string or icon that the user has caused to be highlighted. Many actions operate on the current selection, which need not be in the window associated with the action.

Server: A *server* is a machine dedicated to performing one or more services.

Service: A *service* is a related set of facilities provided for general use, such as a print service or file service.

Signal: A *signal* is a Mesa language construct used to help handle exceptional conditions encountered during program execution. Signals are like procedures except that the code to be executed is determined at run-time.

Signaller: The *Signaller* is the program that gets control when a signal is raised, attempts to find an associated catch phrase, and executes the code in the catch phrase.

Simple types: The *simple types* are the enumerated types, the subrange types, and the built-in types.

Size: To *size* a window is to switch its state either from active to tiny or vice versa. (See **Window state**.)

Smalltalk: *Smalltalk* is an object-oriented programming language (and its integrated programming system) developed by Xerox.

Snarf: To *snarf* is to copy files between logical volumes, especially from the CoPilot volume.

Socket: A *socket* is a source or destination of packets on a given machine. A socket is uniquely identified by a 16-bit socket number. Several streams of packets may share a single socket. A socket is accessed through a channel interface and is thus a logical input/output device. The clearinghouse and the time server, for example, each have their own socket.

Space: *Space* is the Pilot interface for managing virtual memory. Space often refers more generally to virtual memory.

Storage Leak: A *storage leak* occurs when a program neglects to free all the storage nodes it has allocated, thus reducing the total amount of space available for the system.

Stream: A *stream* is an abstraction for device- and format-independent sequential access to a collection of data. Some streams also provide random access to the data. A stream is a sequence of bytes, possibly marked by attention flags and possibly partitioned into identifiable subsequences.

Stream component manager: A *stream component manager* is the software entity that

implements a stream component—a transducer, filter, or pipeline.

Stream Handle: A *stream handle* is a pointer to the stream object that identifies the particular stream being accessed.

Stream Object: A *stream object* contains the data and procedures for operations on the stream.

String: A *string* is conceptually a sequence of characters, such as "that". A string is represented in Mesa as a pointer to a record containing an array of characters, the current length, and the current maximum length.

Strongly typed: The Mesa compiler uses static analysis to deduce the type of every constant, variable, and expression to ensure that all programs are type correct. Languages in which such type correctness is determined at compile time are called *strongly typed.*

Stub: A *stub* is a program that implements a Mesa interface in terms of Courier calls to a remote server or workstation.

Subdirectory: A file directory can be divided into a hierarchical collection of *subdirectories.* Subdirectory names are listed from the root of the tree down to the leaves, separated by ">". (See **Path name**.)

Subrange types: A *subrange type* is a type is a type created from a sunset of an existing enumerated type or type whose elements can be linearly ordered. The subrange takes on the characteristics of the enclosing type but are constrained to thke values within some interval.

Subwindow: A window is often composed of one or more rectangular *subwindows.* The Xerox Development Environment provides several standard subwindow types, including form subwindows and text subwindows.

Swap: To *swap* is to transfer data between memory and files, either in response to hints from the client program or upon demand. To *swap in* is to copy from a file window into real memory; to *swap out* is to copy from real memory to a file window.

Swap unit: A *swap unit* is a portion of a space to be swapped together. Proper choice of the size of swap units can improve use of real memory and reduce disk overhead.

Swat: To *swat* is to strike CALL-DEBUG to invoke the debugger.

Switch: A *switch* is a modifier to a command or subcommand, often preceded by a "/".

Symbiote: A *symbiote* is a subwindow that can be added dynamically to a text subwindow in an existing tool without changing the tool or Tajo. A symbiote provides extra facilities via stick-around menu items.

Synchronous call: A *synchronous call* is a procedure call that returns control only after the operation completes.

Tag: The *tag* is a field of a variant record whose value selects one of the alternatives of the variant part by matching one of the adjectives.

Tajo: *Tajo* is the user interface part of the Xerox Development Environment. The main client volume and its boot file are also often called Tajo.

Teledebug: To *teledebug* is to debug remotely, that is, to debug one machine from another over the internet.

Text subwindow: A *text subwindow* is a system-provided subwindow type with text display and editing capabilities.

Thumb: To *thumb* is to position the data in a file (usually text) to an arbitrary position for viewing on a display. The "thumb-index" in some dictionaries performs somewhat the same function: it gets you to roughly the right place quickly.

Timeout: *Timeout* is the failure to complete an operation within a specified amount of time.

Tiny window: A window is *tiny* if it is represented on the display by an icon. A tiny window is not ready for interaction with the

user, but maintains the state of the tool. (Compare **Active window**, **Inactive window**.)

TIP: *Terminal Input Processor (TIP)* is a system for interpreting keyboard and mouse actions and turning them into sequences of commands based on TIP tables.

Tool: A *tool* is a Xerox Development Environment applications program. A tool can run in parallel with other tools, including other instances of the same tool. Tools react to prompting and seldom carry out operations when not in use. A tool need not have a window associated with it, although they usually do.

Transducer: A *transducer* is a software entity that implements a stream, such as the **MStream** interface, connected to a specific device or medium through a Pilot channel.

Trash bin: The *trash bin* is the conceptual container of the most recently deleted selection, which can be retrieved to a different spot or a different window.

Type-ahead: *Type-ahead* is the ability to type characters to a program before that program has asked for them.

Type-in point: Typed characters are inserted at the *type-in point*. The type-in point is indicated by a flashing caret or box.

Type declaration: *Type declarations* collect together common properties of variables. The type name declared refers to these common properties. If one desires to change the properties then one need only change the type declaration.

Uncaught signal: An *uncaught signal* occurs when no module in the call stack handles a *signal* that has arisen. If a signal is uncaught, the Signaller transfers control to the debugger.

Unwind: *Unwind* is a special signal raised by the Signaller to allow procedures about to be deleted from the call stack to do clean up (such as deallocate storage and close files). When there is an unconditional branch out of the catch phrase, the Signaller raises the unwind signal at the point where the original signal originated.

User: A *user* is a person (rather than a program) who avails him or herself of the services of some program or system. (See **Client**.)

User.cm: *User.cm* is a file on the system volume used to set defaults for many of the tools in the Xerox Development Environment. This file allows users to customize their environment.

User Interface: The *user interface* is the man/machine interface. It is the manner in which information is presented to you on the display screen, and the way that you communicate with using keyboard and mouse.

User profile: A *user profile* is commonly accessed global information that identifies a user in the internet. A user profile includes name, password, and Clearinghouse domain.

Valid memory location: A location is *valid* if it is currently allocated. A location that has been freed is *invalid* and should not be referenced.

Value: A *value* is an immutable object that is not changed by computation.

Variant records: A *variant record* consists of an optional common part followed by a variant part. The common part contains components that are common to all records of this type. The variant contains the components of variants of the record. This allows different record values of the same types, or their names.

Version stamp: The *version stamp* is the date and time, accurate to the nearest second, at which a file was created. Different versions of a file are distinguished by their version stamps. Version stamps allow tools such as the binder and the debugger to ensure that proper versions of files are used.

Video-invert: To *video-invert* a region is to cause black areas of the region to become white and white areas to become black.

Virtual memory: *Virtual memory* is the large word-oriented address space of up to 2^{32} words that forms the execution environment.

Volume: See **physical volume**, **logical volume**.

Wedged: A program is *wedged* when there is no response to input from either the keyboard or the mouse. The whole system or some part may be wedged.

Window: A *window* is a rectangular region of the display in which text and graphics can be displayed. Most tools communicate via windows.

Window state: The *state* of a window is either active, tiny, or inactive. (See **Active window**, **Tiny window**, **Inactive window**.)

Word: A *word* is the basic 16-bit unit of information manipulated by Mesa processors.

Workstation: A *workstation* is a machine connected to the network and used as a personal computer. Most Dandelions are used as workstations. (See **Server**.)

World-swap: A *world-swap* is the process of writing out the complete state of a logical volume onto a disk file and reading in a different state. CoPilot normally works by world-swaps between the debugger and the program being debugged. (See **Outload file**.)

XDEUG: The *Xerox Development Environment User's Guide* provides an introduction to the Xerox Development Environment and describes how to use the tools that make up the environment.

Zoom: To *zoom* a window is to switch the size of an active window either from normal to full screen or vice-versa. Zooming a normal-sized window also puts it on top of all other windows.

Zone: A *zone* is a client-designated area of virtual memory used to allocate and free arbitrary-sized storage nodes. (See **Heap**.)