Lisa

Development System

INTERNALS

DOCUMENTATION

apple computer inc

Katie Withey
Development Tools Group
February 1984

memo

To: Development Tools Group, Operating Systems Group, Numerics Group, Eric Harslem, Larry Tesler, Pete Cressman, Steve Luckau, Paul Williams, Barry Haynes, Susan Keohan, Chris Espinosa, Caroline Rose, Jerome Coonen

From: Katie Withey, x3596

Date: 15 February 84

Re: Internals Documentation

Attatched is the first draft of the *Lisa Development System Internals Documentation.* Please note that this is a living document; changes will be made, and no part of it is guarenteed to be accurate. If you have any changes or corrections, PLEASE don't just mark them in your copy; tell me about them. Suggestions for inclusions in the next release are also welcome.

Preface

The purpose of this document is to explain the internal structures and algorithms used by the Lisa's run-time environment and development tools, and the internal library units (such as OBJIOLIB and SULIB) that are related only to Lisa systems software. It is actually a collection of documents and memos, any of which can be used separately, all relating to different aspects of the system.

This is a reference document for programmers working on the following:

- Maintaining or enhancing existing Lisa development software.
- Writing compilers or utilities for the Lisa Workshop, either on contract with Apple or as third-party independants.
- Writing assembly-language programs that will interface with our compiled code.

How will they benefit from this document?

- It will save the people maintaining tools the trouble of looking through the code themselves to find information.
- It will save outside programmers, who don't have access to the code, from calling us to ask questions about things that *we* have to look up in the code.
- Parts of it will be included as a reference section in technical contracts that we assign to outside programmers.
- It will provide assembly-language programmers with such specifics as register conventions, parameter-passing techniques, and memory layouts used by the compiler for different types of arrays and structures.
- It can be used to train new systems software programmers on the existing internals of the system.

Contents

Lisa Development Software Documentation: A Road Map

Introduction

This road map was designed to help you to find your way around the various documents describing program development for the Lisa. It will help you decide which software you need to learn more about, which software you can ignore for the moment, and how you should proceed in studying the rest of the technical documentation.

General Overview of the Environments Available

There are as many ways of writing programs as there are creative programmers. However, Apple supports only three general styles of programs that you can write for the Lisa: those written for 1) the Workshop environment, 2) the QuickPort environment, and 3) the ToolKit environment. Programs written for any of these environments can use most of the same units and libraries, but there are some important differences of which you should be aware.

The *Workshop* (Figure 1) provides a simple non-window, character and graphic environment within which a program may run. Programs written to run in this environment may use Pascal's built-in I/O for both files and textual display to the console's terminal emulator, or they may directly utilize the Lisa OS's file system primitives. They may also use the QuickDraw unit for drawing bitmap graphics and displaying text in a variety of fonts with various attributes, and may utilize a variety of other useful library routines. These programs are not able to use the Lisa Desktop libraries dealing with windows, menus, and dialog boxes, nor do they have easy access to Lisa Office System documents.

In addition to providing these run-time facilities, the Workshop also includes a command shell which makes available to users an extensive set of facilities for: 1) Interactive program development in Pascal, Assembly, BASIC, and COBOL; 2) File and device manipulation; and 3) Interactive and batch program execution and control.

QuickPort (Figure 2) provides the simplest Desktop environment, at least from the programmer's viewpoint. In most respects, writing a program for the QuickPort environment is identical to writing one for the Workshop environment. Using Pascal's built-in I/O facilities, programs written for QuickPort may do textual display to a variety of window-based terminal emulators, and may also display graphics using QuickDraw. These programs do not directly use the Lisa Desktop libraries, and are, in fact, unaware of such things as the window environment, the mouse, and menus. They may, however, exchange information with Lisa Office System documents via the Cut/Paste mechanism.

The *ToolKit* (Figure 3) provides the most complete access to the Desktop facilities. From the programmer's viewpoint, it also requires the most knowledge of these facilities. Programs written using the ToolKit use the Generic Application and may use any of the ToolKit building blocks, which provide easy, controlled access to the Lisa Desktop libraries, the mouse, and menus. They may also exchange information with Lisa Office System documents via the Cut/Paste mechanism.

Overview of the Pieces

QuickPort is a set of units that are USEd and linked with a program which is to be run in the Desktop environment. QuickPort then provides the program with a "terminal window", to which the program's console I/O may be directed through the use of Pascal's built-in Text I/O facilities. The program simply makes ReadLn and WriteLn calls to display text or receive keyboard input. QuickPort code hides from the program such issues as cutting and pasting information from other Desktop applications, communicating with the Desktop shell, growing and shrinking the window, covering and uncovering the window, and activating or deactivating the program. For a program using QuickPort, such issues are of no concern.

The ***ToolKit*** is a set of libraries that provides standard Lisa application behavior, including windows that can be moved, resized, and scrolled, pull-down menus with standard functions such as saving and printing, and the Cut/Paste mechanism. The ToolKit defines the parts of an application common to all Lisa applications. The object-oriented structure of the ToolKit allows you to implement your application as extensions to the "Generic Application".

The ***Lisa Operating System*** provides the program with an environment in which multiple processes can coexist, with the ability to communicate and share data. It provides a device-independent file system for I/O and information storage, and handles exceptions (software interrupts) and memory management for both code and data segments.

PASLIB is the Pascal run-time support library. Most of the routines in PASLIB support the Pascal built-in facilities, including routines for initialization, integer arithmetic, data and string manipulation, sets, range checking, the heap, and I/O.

Floating Point Libraries provide numeric routines which implement the proposed IEEE Floating Point Standard (Standard 754 for Binary Floating-Point Arithmetic), and higher-level mathematical algorithms. **FPLib** provides Single (32-bit), Double (64-bit), and Extended (80-bit) floating-point data types, a 64-bit Integer data type, conversion from one arithmetic type to another (or to ASCII), arithmetic operations, transcendental functions, and tools for handling exceptions. **MathLib** provides, among others, algorithms such as extra elementary functions, sorting, extended conversion routines, financial analysis, zeros of functions, and linear algebra.

QuickDraw is a unit for doing bit-mapped graphics. It consists of procedures, functions, and data types you need to perform highly complex graphic operations very easily and very quickly. You can draw text characters in a number of fonts, with variations that include boldface, italic, underlined, and outlined; you can draw arbitrary or predefined shapes, either hollow or filled; you can draw straight lines of any length and width; or you can draw any combination of these items, with a single procedure call.

The ***Desktop Libraries*** provide window, graphics, mouse, and menu routines used by all Office System applications. They are not directly called by any programs written for the three run-time environments discussed here, but provide the hidden foundation for both the QuickPort and the ToolKit environments.

The ***Hardware Interface*** unit lets you access Lisa hardware elements such as the mouse, the cursor, the display, the contrast control, the speaker, the keyboard, the micro- and millisecond timers, and the hardware clock/calendar.

The ***Standard Unit*** lets you do string, character, and file-name manipulation, prompting, retrieval of messages from disk files, abort exec file processing, and conversions between numbers and strings.

The ***IOPrimitives*** unit provides you with fast, efficient text-file input and output.

The ***Program Communication*** unit allows programs to communicate with each other and with the Workshop shell.

LisaBug allows you to examine and modify memory, set breakpoints, assemble and disassemble instructions, and perform other functions for run-time debugging.

More Detail

QuickPort: A program which is to make full use of the capabilities of the Lisa Office System will be structured as an endless loop, within which the program continually polls the Window Manager for any events it should respond to. We will refer to such a program as an *Integrated Program*. An integrated program must handle such asynchronous events as the program's window being activated or deactivated, the window being opened, closed, moved, resized, or needing update, the mouse button going down or up, and a key going down or up. The program must also be a good citizen in Lisa's multi-tasking but non-preemptive scheduling environment by volunteering periodically to yield the CPU to any other process needing service. These are just a few of the important characteristics of an integrated program. The result of a program following these and other guidelines will be that it exhibits the same consistent, responsive behavior as other Apple-written programs like LisaDraw.

QuickPort is a collection of pieces which make writing programs for the Office System's window environment as easy as writing them for the Workshop's non-window environment. NOTE: In order to differentiate the QuickPort modules from the program which uses them, we will refer to the program itself as a *Vanilla Program*. QuickPort allows the vanilla program to be more traditionally structured, as if its user interfacing were being done through a smart text/graphics terminal; the vanilla program presents its display to the user by a combination of text I/O calls (e.g., WriteLn/ReadLn) and QuickDraw calls (e.g., DrawString/PaintRect). The QuickPort modules handle all events from the Window Manager, provide for yielding the CPU to competing processes at specific points, and in general shelter the program from the sometimes tricky requirements of writing an integrated program for the Lisa Office System.

QuickPort provides the vanilla program with a window, which may be divided into a *Text Panel* and a *QuickDraw Panel* for displaying both textual and graphic information. Each of these optional panels is configurable in size and location, and may be independently scrolled horizontally or vertically. Text and Graphics windows may be overlaid, so the resulting window presents a composite of both types of output. The window may be resized, moved, covered, or uncovered without the vanilla program even being aware of such events. Textual and graphic information may be exchanged between a vanilla program's document and other documents, whether vanilla or integrated, by using the familiar Cut/Paste mechanism. Without any effort on the part of the vanilla program, the end user is given a large measure of control over the window's configuration and behavior, using mouse and menu actions supported by QuickPort.

The user may request printing of either the text panel or the graphics panel. In addition, vanilla programs may produce printed output under program control by writing to the -PRINTER logical device. Whereas, in the Workshop environment, printing is immediate (each line printing as soon as the program "writes" it), in the QuickPort/Desktop environment printing is all spooled. This means that the printed output of a vanilla program will be submitted to the Office system's PrintShop, which determines from the print queue when the document will be printed.

The *Text Panel* emulates a terminal display which corresponds to the Pascal built-in OUTPUT file, the built-in INPUT file, and the -CONSOLE and -KEYBOARD logical devices. Apple provides emulators for the *VT100* and *SORDC* terminals, and makes it possible for you to either customize them or create entirely new terminal emulators. These terminal emulators are actually *filters* which pre-process the character output stream destined for the *Standard Terminal Unit*, which provides the Text Panel display. Each emulator's job is to recognize the terminal-specific character sequences imbedded in the output stream which are commands to the terminal, and to call upon the Standard Terminal Unit to take the appropriate actions. A program may eliminate the filtering step, if desired, by calling directly upon the Standard Terminal Unit for display actions.

The *Graphics Panel* allows your program to display graphics on a bitmap which is a maximum of 720 pixels wide by 364 pixels high--the same size as Lisa's physical screen bitmap. This panel can be resized by the user or under program control, and can be scrolled horizontally and vertically to display different parts of the entire bitmap. The Graphics Panel supports every QuickDraw call, including those related to setting foreground and background colors for printed output. An application may write anywhere in the coordinate plane of its graphics panel ('grafPort', to use QuickDraw's terminology), without having to worry about where its window is placed on the screen or what other windows are in front of it. QuickDraw, with a little help from the Window Manager, keeps the application's output from getting out of the graphics panel or from clobbering other windows.

The ToolKit: The ToolKit is a set of libraries that provides standard behavior that follows the design principles characterizing Lisa applications:

- Extensive use of graphics, including windows and the mouse pointer.
- Use of pull-down menus for commands.
- Few or no operating modes.
- Data transfer between documents by simple cut and paste operations.

For example, all Lisa applications have windows that can be moved around the screen, and that can usually be resized and scrolled. The ToolKit takes care of all these functions. The ToolKit also displays a menu bar for the active application, and provides a number of standard menu functions, such as saving, printing, and setting aside.

However, the ToolKit is more than a set of libraries. Because the ToolKit is written using Clascal, the ToolKit is almost a complete program by itself. You can, in fact, write a five-line main program, compile it, link it with the ToolKit, and run it. What results is the Generic Application.

The Generic Application has many of the standard Lisa application characteristics. A piece of Generic Application stationary can be torn off, and, when the new document is opened, it presents the user with a window with scroll bars, split controls, size control, and a title bar. The mouse pointer is handled correctly when it is over the window. The window can be moved, resized, and split into multiple panes. There is a menu bar with a few standard functions, so that the generic document can be saved, printed, and set aside. The single Generic Application process can manage any number of documents. You cannot, however, do anything within the window, aside from creating panes. The space within the window, along with the additional menu fuctions, is the responsibility of the real application.

Therefore, when you write a Lisa application using the ToolKit, you essentially write extensions to the Generic Application. It is very easy to write extensions to any Clascal program. To insert your application's functions, you create a set of subclasses, including methods to perform the work of you application, and then you write a simple main program, and compile and link it with the ToolKit.

Whenever necessary, the ToolKit calls your application's routines. For example, if the user scrolls the document, the ToolKit tells your program to redraw the changed portions of the window. Your program does not need to be concerned with when redrawing is required.

One effect of Clascal is that you can write applications in steps. You can begin by doing the least amount possible, and get an application that does very little, but will run. You can then extend your application bit by bit, checking as you go. This characteristic of Clascal makes it easy to extend the capabilities of ToolKit programs, even years after the original program.

The ToolKit's debugger, KitBug, provides run-time debugging of ToolKit Clascal programs. It allows you to do performance measurements, set breakpoints and traces, single-step through your program one statement at a time, and do high-level examinations of data objects.

The Operating System: The Operating System provides an environment in which multiple processes can coexist, with the ability to communicate and share data. It provides a file system for I/O and information storage, and handles exceptions (software interrupts) and memory management.

The *File System* provides input and output. It accesses devices, volumes, and files. Each object, whether a printer, disk file, or any other type of object, is referenced by a pathname. Every I/O operation is performed as an uninterpreted byte stream. Using the File System, all I/O is device-independent. The File System also provides device-specific control operations.

A *process* consists of an executing program and the data associated with it. Several processes can exist at once, and will appear to run simultaneously because the processor is multiplexed among them. These processes can be broken into multiple segments which are automatically swapped into memory as needed. Communication between processes is accomplished through events and exceptions. An *event* is a message sent from one process to another, or from a process to itself, that is delivered to the receiving process only when the process asks for it. An *exception* is a special type of event that forces itself on the receiving process. In addition to a set of system-defined exceptions (errors), such as division by zero, you can use the system calls provided to define any other exceptions you want.

Memory management routines handle data segments and code segments. A *data segment* is a file that can be placed in memory and accessed directly. A *code segment* is a swapping unit that you can define. If a process uses more memory than the available RAM, the OS will swap code segments in and out of memory as they are needed.

PASLIB: PASLIB is the Pascal run-time support library. It provides the procedures and functions that are built into the Pascal language, acts as the run-time interface to the Operating System, and "completes" the 68000 instruction set by providing routines for the compiler-generated code to call upon in lieu of actual hardware instructions.

PASLIB routines are called with all parameters passed on the stack. There is an initialization routine to initialize necessary variables, libraries, and exception-handlers and set up global file buffer addresses, and a termination routine to kill processes. You can do four-byte integer arithmetic. Data can be moved, or scanned for a particular character. String manipulation routines include concatenating, copying, inserting or deleting a substring, determining the position of a substring, and comparing strings for equality. Set manipulation routines let you find set intersections or differences, adjust the size of a set, and compare sets for equality. There are range-checking and string range-checking routines. Heap routines let you allocate memory in the heap, mark or release the heap, check available memory in the heap, and check the heap result. I/O routines let you read and write lines, characters, strings, packed arrays of characters, booleans, and integers, as well as check for a keypress or an end-of-line, and send page marks. File I/O routines include rewriting, resetting or closing a file, detecting an end-of-file, reading and writing blocks, and get, put, and seek procedures.

Floating-Point Libraries: The Lisa provides arithmetic, elementary functions, and higher level mathematical algorithms in its intrinsic units **FPLib** and **MathLib**, which are contained in the file **IOSFPLIB**.

FPLib provides the same functionality as the SANE and Elems units on the Apple] [and *///*, including:

- Arithmetic for all floating-point and Comp types.
- Conversions between numerical types.
- Conversions between numerical types, ASCII strings, and intermediate forms.
- Control of rounding modes and numerical exception handling.
- Common elementary functions.

MathLib provides the extra procedures available only on the Lisa:

- Extra environments procedures.
- Extra elementary functions.
- Miscellaneous utility procedures.
- Sorting.
- Free-format conversion to ASCII.
- Correctly rounded conversion between binary and decimal.
- Financial analysis.
- Zeros of functions.
- Linear algebra.

QuickDraw: Virtually all of Lisa's graphics are performed by the QuickDraw unit. You can draw text, lines, and shapes, and you can draw pictures combining these elements. Drawing can be done to many distinct "ports" on the screen, each of which is a complete drawing environment. You can "clip" drawing to arbitrary areas, so that you only draw where you want. You can draw to an off-screen buffer without disturbing the screen, then quickly move your drawing to the screen.

Text characters are avilable in a number of proportionally-spaced fonts. Any font can be drawn in any size--if a font isn't available in a particular size, QuickDraw will scale it to the specified size. You can draw characters in any combination of boldface, italic, underlined, outlined, or shadowed styles. Text can be condensed or extended, and it can be justified (aligned with both a left and a right margin).

Straight *lines* can be drawn in any length and width, and can be solid-colored (black, white, or shades of gray) or patterned.

Shapes defined by QuickDraw are rectangles, rectangles with rounded corners, full circles or ovals, wedge-shaped sections of circles or ovals, and polygons. In addition, you can describe any arbitrary shape you want. All shapes can be drawn either hollow (just an outline, which has all the width and pattern characteristics of other lines) or solid (filled in with a color or pattern that you define).

QuickDraw lets you combine any of these elements into a *picture*, which can then be drawn--to any scale--with a single procedure call.

Three-dimensional graphics capabilities are also available, in a unit called Graf3D, which is layered on top of the QuickDraw routines. Graf3D lets you draw three-dimensional objects in true perspective, using real variables and world coordinates.

The Hardware Interface: The Hardware Interface unit lets you access Lisa hardware elements such as the mouse, the cursor, the display, the speaker, the keyboard, and the timers and clocks.

Mouse routines determine the location of the mouse, set the frequency with which software knowledge of the mouse location is updated, change the relationship between physical mouse movement and the movement of the cursor on the screen, and keep track of how far the mouse has moved since boot time.

Cursor routines let you define different cursors, track mouse movements, and display a busy cursor when an operation takes a long time.

Screen-control routines can set the size of the screen, and set contrast and automatic fading levels.

Speaker routines allow you to find out and set the speaker volume, and create sounds.

Routines are provided to handle the different *keyboards* available for the Lisa, as well as the mouse button and plug, the diskette buttons and insertion switches, and the power switch. You can find out which keyboard is attached, and set the system to believe that a different physical keyboard is connected. You can check to see what keys (including the mouse button) are currently being held down, look at or return the events in the keyboard queue, and read and set the repeat rates for repeatable keys.

Date and time routines let you access the microsecond and millisecond timers and check or set the date and time.

The Standard Unit: The Standard Unit (StdUnit) is an intrinsic unit providing a number of standard, generally-useful functions. The functions are divided into areas of functionality: character and string manipulation, file name manipulation, prompting, retrieval of error messages from disk files, Workshop support, and conversions.

The unit provides types for standard strings and for sets of characters, definitions for a number of standard characters (such as <CR> and <BS>), and procedures for case conversion on characters and strings, trimming blanks, and appending strings and characters.

File name manipulation functions let you determine if a pathname is a volume or device name only, add file name extensions (such as ".TEXT"), split a pathname into its three basic components (the device or volume, the file name, and the extension), put the components back together into a file name, and modify a file name given optional defaults for missing volume, file, or extension components.

Prompting procedures let you get characters, strings, file names, integers, yes or no responses, and so forth from the console, providing for default values where appropriate.

Special Workshop functions let you stop the execution of an EXEC file in progress, find out the name of the boot and current process volumes, and open system files, looking at the prefix, boot, and current process volumes when trying to access a file.

Conversion routines let you convert between INTEGERs (or LONGINTs) and strings.

The IOPrimitives Unit: The IOPrimitives unit provides you with fast, efficient text-file input and output routines with the functionality of the Pascal I/O routines. It includes routines for reading characters or lines, and for writing characters, lines, strings, and integers, plus the low-level routines on which the others are based.

The Program Communications Unit: The Program Communications unit (ProgComm) provides three mechanisms for communication between one program and another or between a program and the shell. The first two involve strings sent from a program to the shell; one tells the shell which program to run next, the other is a "return string" that can be read by the exec file processor to tell an exec file, for example, whether the program completed successfully. The third mechanism involves reading from and writing to a 1K byte communications buffer, global to the Workshop. Using the unit, a program can invoke another program and provide its input through the buffer, without user intervention.

LisaBug: LisaBug provides commands for displaying and setting memory locations and registers, for assembling and disassembling instructions, for setting breakpoints and traces to trace program execution, for manipulating the memory management hardware, and for measuring execution times using timing functions. Utility commands are also available to clear the screen, print either the main screen or the LisaBug screen, change between decimal and hexadecimal, change the setting of the NMI key, and display the values of symbols.

Where to Go from Here

The Lisa development software is not fully documented yet. The following is a list of what is available, some of it only internally, as of this publication. Note that the spring-release manuals will be organized differently from the current versions, and will incorporate much of the information that is now in the internals documentation or in separate documents.

Pascal Reference Manual for the Lisa
includes: QuickDraw
Hardware Interface
Floating-Point Library

Operating System Reference Manual for the Lisa

Workshop User's Guide for the Lisa

Lisa Development System Internals Documentation
includes: Pascal Run-Time Library
Standard Unit
LisaBug
Floating-Point Libraries

*QuickPort Applications User Guide**

*QuickPort Programmer's Guide**

An Introduction to Clascal

Clascal Self-Study

ToolKit Reference Manual

ToolKit Training Segments

Numerics Manual: A Guide to Using the Apple /// Pascal SANE and Elems Units
FPLib provides the same functionality as these units.

*MathLib Guide**

*These manuals currently in rough draft form.

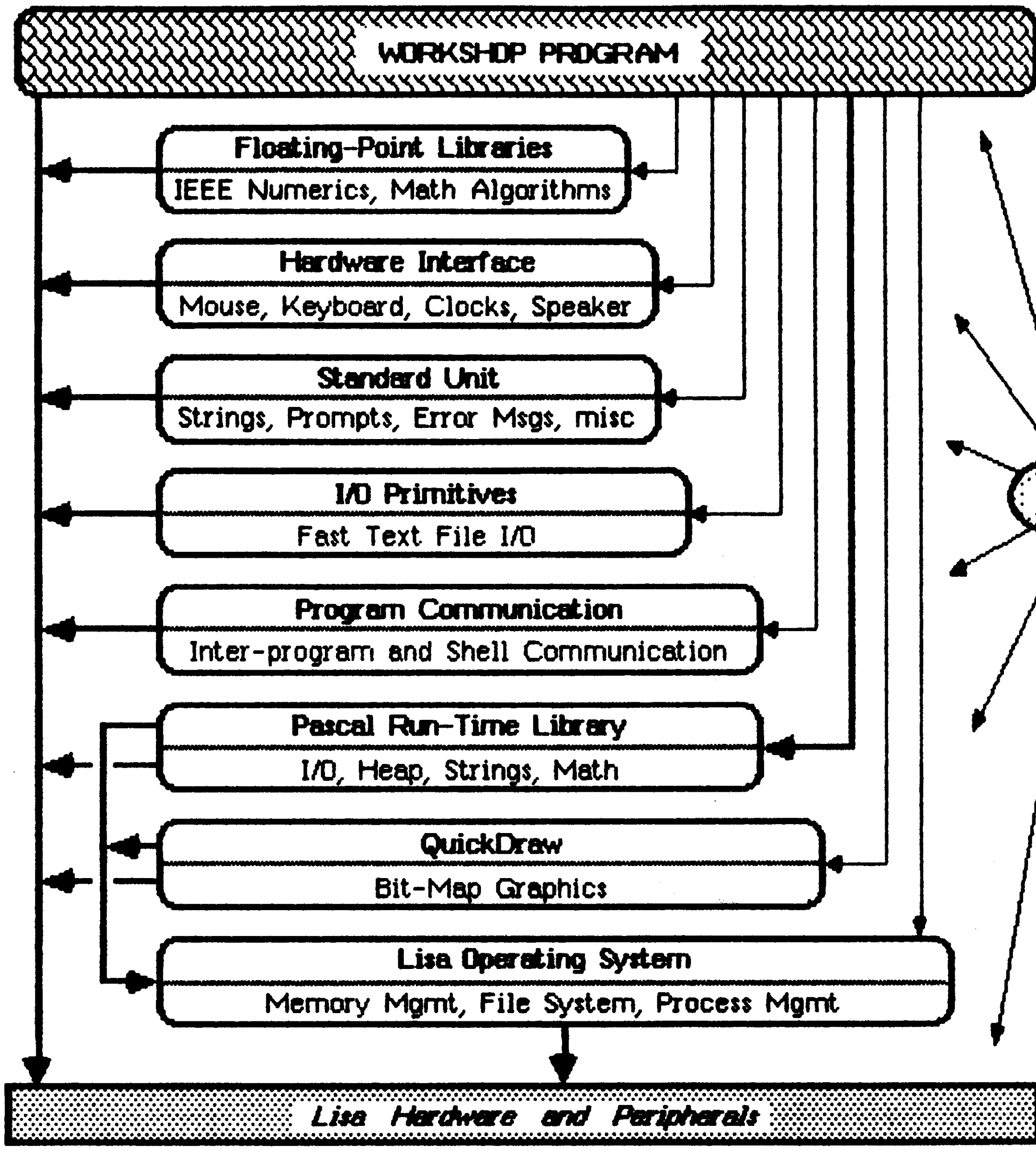
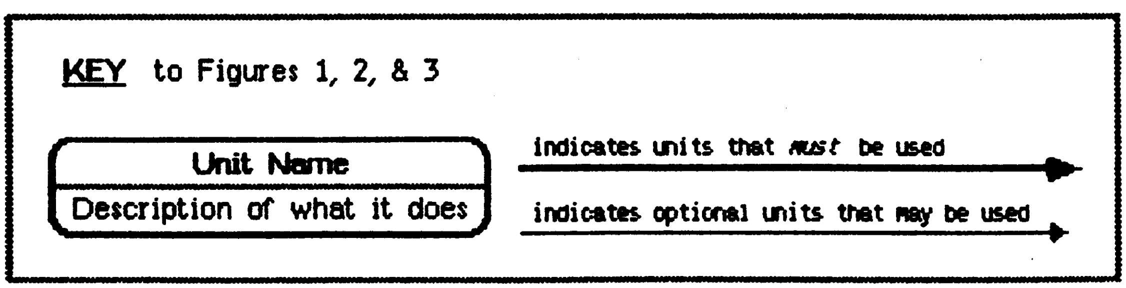


Figure 1
The Workshop Run-Time Environment



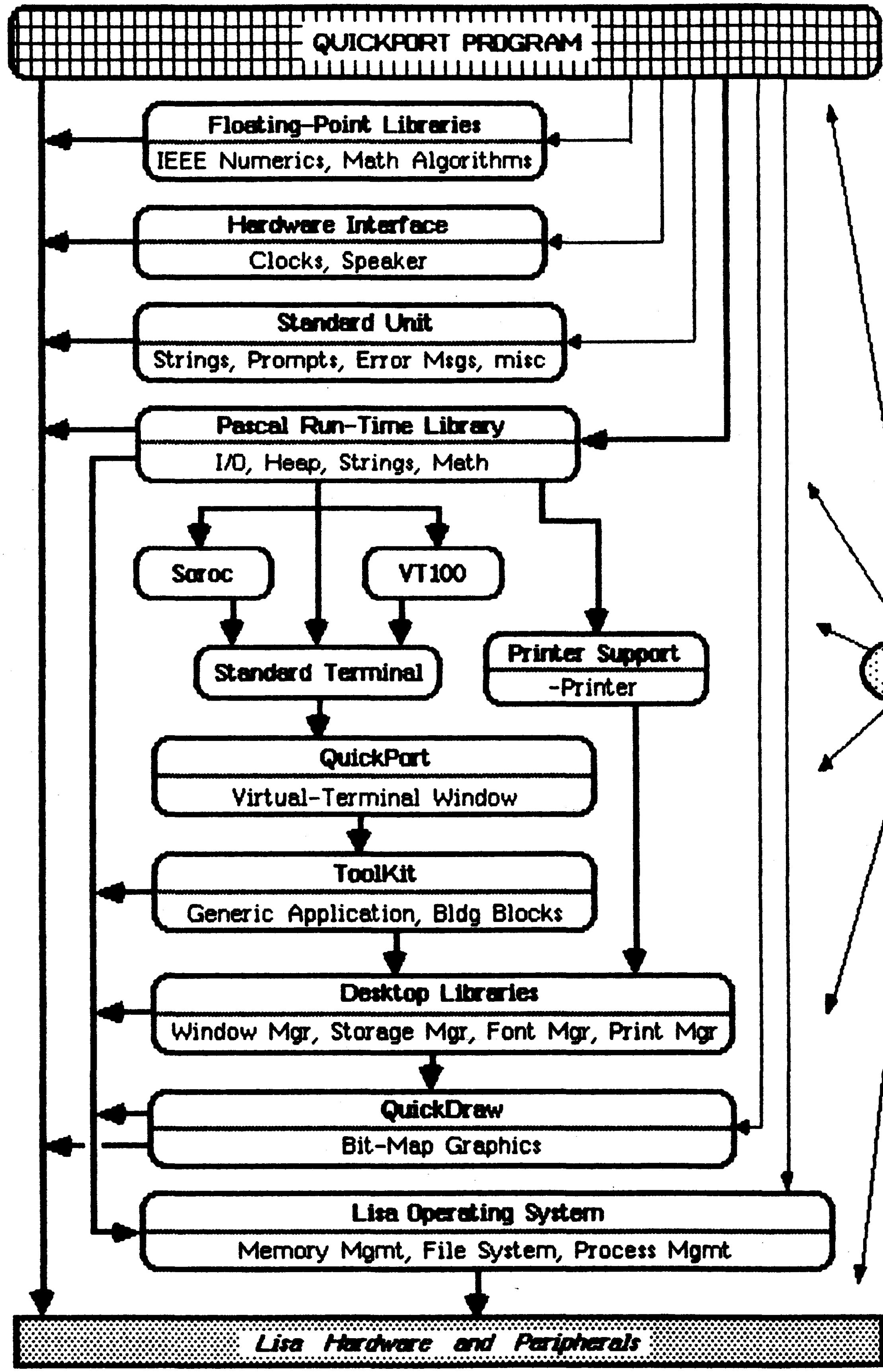


Figure 2
The QuickPort Run-Time Environment

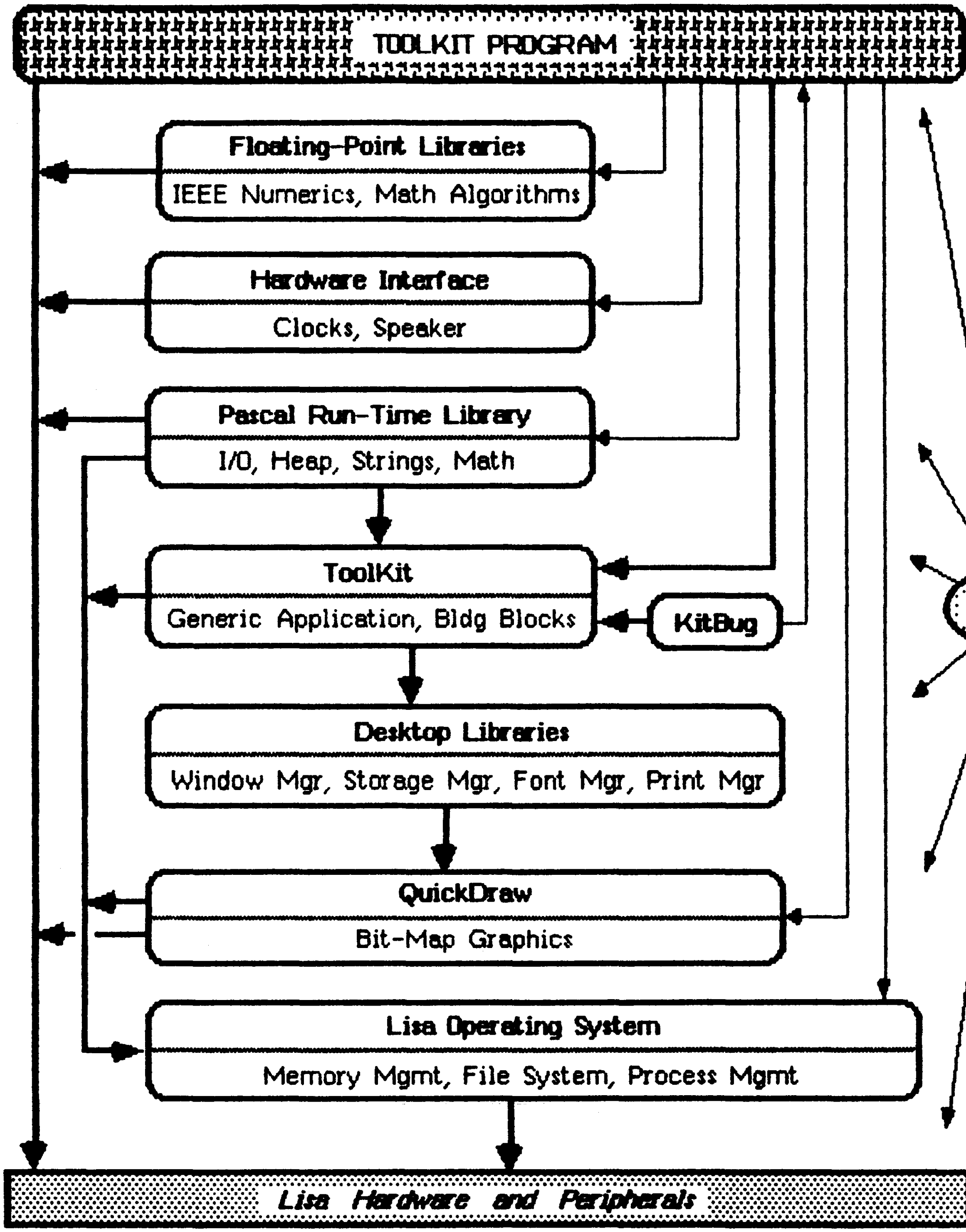


Figure 3
The ToolKit Run-Time Environment

Pascal Compiler Directives

The following compiler commands are available:

$%+** or **$%– Allow the % symbol in identifiers. The default is **$%–**.

$C+** or **$C– Turn code generation on (+) or off (–). This is done on a procedure-by-procedure basis. These commands should be written between procedures; results are unspecified if they are written inside procedures. The default is **$C+**.

$D+** or **$D– Turn the generation of procedure names in object code on (+) or off (–). These commands should be written between procedures; results are unspecified if they are written inside procedures. The default is **$D+**.

$E filename Start making a listing of compiler errors as they are encountered. Analogous to **$L filename** (see below). The default is no error listing.

$H+** or **$H– Disables handle checking so dereferenced handles (master pointers) may be used in **with** statements, on the left side of assignment statements, and in expressions involving procedure calls. The default is **$H+**.

$I filename Start taking source code from file **filename**. When the end of this file is reached, revert to the previous source file. If the filename begins with + or –, there must be a space between **$I** and the filename (the space is not necessary otherwise). Files may be **$I** included up to five layers deep.

$L filename Start listing the compilation on file **filename**. If a listing is being made already, that file is closed and saved prior to opening the new file. The default is no listing. If the filename begins with + or –, there must be a space between **$L** and the filename (the space is not necessary otherwise).

$L+** or **$L– The first + or – following the **$L** turns the source listing on (+) or off (–) without changing the list file. You must specify the listing file before using **$L+**. The default is **$L+**, but no listing is produced if no listing file has been specified.

$O+** or **$O– Suppress register opitimization (–). The default is **$O+**.

$OL Optimization limited--use the old (2.0 release) optimization mechanism, instead of the new one. The default is the new one.

$OV+** or **$OV– Turn integer overflow checking on (+) or off (–). Overflow checking is done after all integer add, subtract, 16-bit multiply, divide, negate, abs, and 16-bit square operations, and after 32 to 16 bit conversions. The default is **$OV–**.

$R+** or **$R– Turn range checking on (+) or off (–). At present, range checking is done in assignment statements and array indexes and for string value parameters. No range checking is done for type **longint**. The default is **$R+**.

$S segname Start putting code modules into segment **segname**. The default segment name is a string of blanks to designate the "blank segment," in which the main program and all built-in support code are always linked. All other code can be placed into any segment.

$U filename Search the file **filename** for any units subsequently specified in the uses-clause. Does not apply to intrinsic-units.

$U+** or **$U– Tell the system not to search **INTRINSIC.LIB** for units you **use** (–). The default is **$U+** — the system searches **INTRINSIC.LIB** first, then your own libraries.

$X+** or **$X– Turn automatic run-time stack expansion on (+) or off (–). Run-time stack expansion is the insertion of an extra 4-byte instruction per procedure to ensure that the Lisa's memory-management mechanism has mapped in enough stack space for the execution of the procedure. With **$X–**, excessive use of the stack by the procedure could cause a bus error. The default is **$X+**.

$SETC The **$SETC** command has the form:

{$SETC ID := EXPR}

or

{$SETC ID = EXPR}

where **ID** is the identifier of a compile-time variable and **EXPR** is a compile-time expression. **EXPR** is evaluated immediately. The value of **EXPR** is assigned to **ID**.

Compile-time variables are completely independent of program variables; even if a compile-time variable and a program variable have the same identifier, they can never be confused by the compiler.

Note the following points about compile-time variables:

- Compile-time variables have no types, although their values do. The only possible types are **integer** and **boolean**.
- At any point in the program, a compile-time variable can have a new value assigned to it by a **$SETC** command.

$IFC, $ENDC
$ELSEC

Conditional compilation is controlled by the **$IFC**, **$ELSEC**, and **$ENDC** commands, which are used to bracket sections of source text. Whether a particular bracketed section of a program is compiled depends on the **boolean** value of a *compile-time expression*, which can contain *compile-time variables*.

The **$ELSEC** and **$ENDC** commands take no arguments. The **$IFC** command has the form:

{$IFC EXPR}

where **EXPR** is a compile-time expression with a **boolean** value.

These three commands form constructions similar to the Pascal if-statement, except that the **$ENDC** command is always needed at the end of the **$IFC** construction. **$ELSEC** is optional.

$IFC constructions can be nested within each other to 10 levels. Every **$IFC** must have a matching **$ENDC**.

Compile-time expressions appear in the **$SETC** command and in the **$IFC** command. A compile-time expression is evaluated by the compiler as soon as it is encountered in the text.

The only operands allowed in a compile-time expression are:

- Compile-time variables
- Constants of the types **integer** and **boolean**. (These are also the only possible types for results of compile-time expressions.)

All Pascal operators are allowed except as follows:

- The **in** operator is not allowed.
- The **@** operator is not allowed.
- The **/** operator is automatically replaced by **div**.

Pascal Code Cruncher's Handbook

Fred Forsman

Revision 1.0
September 28, 1983

Remove unsightly, unwanted bytes
in the privacy of your own office.

No gimmicks, pills, fads or strenuous exercise.

PASCAL CODE CRUNCHER'S HANDBOOK

Fred Forsman

Introduction

This document explains how to reduce the size of Pascal code by changes at the Pascal source level. Thus what will be presented are source transformations which result in semantically equivalent, but smaller code.

While these transformations will produce smaller code, they are unlikely to produce code that is "better" in all senses. Sometimes you will be trading off clarity for efficiency since typically you will be changing what was the first and obvious way of writing your code. On the other hand, your code may benefit (and actually become clearer) just from having been thought about a second time. Nevertheless, if it is given that you must reduce your code size, you may find these source transformations more palatable (and more maintainable) than rewriting in assembly language.

Please note that this is a *living document,* that is, no claims are made that this is a complete or final list of source transformation techniques. New techniques will be added as I find out about them (so if you are aware of some transformations not mentioned here please let me know about them). Also, some of the techniques described will be removed from this document when future compiler optimizations obviate the need for them.

o o o

Thanks to Al Hoffman for his invaluable assistance in researching and documenting much of the material presented here. Thanks also to Ken Friedenbach and Rich Page.

How to find what code to crunch and how to measure your progress

Given a Pascal unit which you want to crunch, you need to identify the procedures which are most likely to benefit from crunching and you need a mechanism by which to measure the results of your efforts. The Pascal **code generator** writes information to the console on the size of the code generated for each procedure and the size of the code for the unit being compiled. With a compile exec file such as the one below you can redirect this information to a file, for use in later analysis.

```
$EXEC {perform a compile}
$ { the first parameter (%0) specifies what file to compile }
$ { if a second parameter is specified, it is used for the output obj
    file, otherwise we default to "%0.obj" }
$ { if a third parameter is specified, the code generator's console
    output is redirected to "%2.text", otherwise default to "g.text" }
$ { the intermediate file is put in a temp file on -paraport  }
P{Pascal}%0

-paraport-temp
$IF %2 <> '' THEN
  S{Sys-mgr}O{OutputRedirect}%2.text
$ELSE
  S{Sys-mgr}O{OutputRedirect}g.text
$ENDIF
Q{quit Sys-mgr}
G{generate}-paraport-temp
$IF %1 <> '' THEN
  %1
$ELSE
  %0
$ENDIF
S{Sys-mgr}O{OutputRedirect}-console
Q{quit}
$ENDEXEC
```

Once you have the code generator's console output, the first step is to identify the easy targets for crunching: most often these will be the larger routines (code size > 250 bytes, or some similar criterion). The above exec file can then be used to verify that any changes you make actually result in code size improvements.

If you are working on code that is not totally new, chances are that it has undergone a number of major and minor changes. As code is modified, "dead" code and variables are often left around inadvertently. These unused objects can be discovered and removed by checking the code with the various **cross reference utilities**. (While the Workshop linker will remove dead code automatically it will not remove dead variables.)

For those of you who want to know what the compiler is *really* doing, use the **DumpObj** utility to look at a disassembly of any of the procedures or functions you are interested in.

How to crunch code: techniques

Following are a number of techniques for Pascal source transformation. The fine print following the description of each technique attempts to estimate the potential space savings, the difficulty of implementation, and probability of introducing errors.

1. The first law of code crunching: don't use in-line code when a procedure to do the same thing exists. The in-line code may be faster, but space is more important in the vast majority of cases. In order to apply this law effectively you should **KNOW WHAT IS AVAILABLE IN THE LIBRARIES**. Similarly you should be familiar with what the language provides, particularly in the area of built-in procedures and functions.

   Using existing code is pure gain. The danger of doing so should be minimal since the compiler and libraries should be error free (or at least their bugs will be recognized and fixed sooner than your private code which is exercized less often).

2. An extension of the above law is the creation procedures which perform code sequences which are repeated often in your code (minor differences can be handled by parameterization). One name for this technique is "**factoring**". Use of parameters can degrade the optimization if the size of the code being factored is small. On the other hand, if introduction of a parameter will allow sharing of a long sequence of code the extra overhead should be well worth it. A word of warning: check to see whether your factoring really paid off — the code being factored out should not be smaller than the procedure call (and any parameter passing) that replaces it. A point to note is that factoring of even single statements can be fruitful, for example:

   A[F(X)] := A[F(X)] + 1;   *becomes*   INCA;

   Factoring can be a BIG win in many cases, often saving more than can be achieved by any other technique. So it often pays to look through your code for common code sequences. Difficulty and likelyhood of errors are low, but increase if parameters must be introduced.

3. Make procedures that are 50-100 lines long - around 300 bytes of code - to optimize allocation of variables to register. Shorter routines do not have enough occurrences of variables to make register allocation worthwhile, and longer routines create more opportunities for register optimization than there are registers available.

   The amount of improvement using this technique is highly variable. Difficulty is moderate; likelyhood of errors is low.

4. Avoid the use of global scalar (1 to 4 byte) variables whenever possible - global variables are never put into registers. Techniques applicable here include:

4a. Assign a frequently used global variable to a local variable, and change all references to be the local quantity. Caution! Beware of saving and restoring the global quantity around procedure calls that might access the global quantity.

The amount of improvement will be two to four bytes per reference, with the greatest gain appearing on assignments like A:=A+1. There is an overhead cost to assign the local and save registers (4 to 14 bytes). Improvement will not occur if the registers have already been assigned to locals that are used more frequently than the global is.

The amount of improvement using this technique is noted above. Difficulty is low; likelyhood of errors is high.

4b. Further leverage on (4a) can be obtained if the same local temporary variable is reused in different parts of the procedure for different global variables. In this way, less frequently used globals still have a chance for optimization into registers.

Improvement is two or more bytes per additional reference, less 4 bytes per new global assigned. Difficulty is moderate; likelyhood of errors is even higher than (4a).

4c. Another, more reliable way of converting a global to a local is to pass the global variable as a var parameter to the routine. Parameters are treated like local variables.

Improvement is two or more bytes per reference, less 8-10 bytes per additional parameter, subject to register competition as noted above. Difficulty and likelyhood of errors with var parameters is low.

4d. Move a large main program body into a main subroutine. Move all global variables that are only accessed by the main program into the subroutine.

Improvement is generally small, since the main program body is usually a small part of the total code. Difficulty and likelyhood of errors are low.

5. In a moderate to large procedure, the number of scalar (1 to 4 byte) local variables (and parameters) should be kept to a minimum, since there is competition for registers. Briefly used integer quantities and loop variables, for example, should all be stored in the same variable (which might be appropriately named "tempint" or some other generic name). Beware, of course, that the variables

usages are never simultaneous.

Improvement, for each additional local variable that overloads an existing register, is typically two bytes per reference. Difficulty is low; likelyhood of errors is moderate.

6. Avoid, at all costs, passing frequently used local variables as var parameters or using them in nested procedures. (Also for frequently accessed parameters.) These actions inhibit the value from being located in a register. Replace *passing as a var parameter* with assignment to a new local variable, passing the new local, then doing a reverse assignment. Replace *nested procedure usage* of the variable with passing the variable as a non-var parameter, use of the parameter inside the subroutine, then, if the nested procedure changes the value, copy the parameter into a new variable at the end of the subroutine copy it back into the main local variable after the call. The following example illustrates optimization of nested usage of A and B:

```
PROCEDURE UPPER;
VAR A,B:INTEGER;
    PROCEDURE LOWER;
    BEGIN
    A := B;

    END;
BEGIN
LOWER;
{other statements}
{frequent uses of A and B}
END;
```

converts to-->

```
PROCEDURE UPPER;
VAR A,B,TEMP:INTEGER;
     PROCEDURE LOWER(A,B:INTEGER);
     BEGIN
     A := B;
     TEMP := A;
     END;
BEGIN
LOWER;
A := TEMP;
{frequent uses of A and B}
END;
```

Note that, in the above case, if A is not frequently used in the subroutine, it could be eliminated as a parameter and the assignment could be made to TEMP directly:

```
    PROCEDURE LOWER(B:INTEGER);
    BEGIN
    TEMP := B;
    END;
```

A final added technique that can be used with procedure calls is to pass the local as a non-var parameter, change the procedure to a function, and assign the returned function result back to the local variable.

```
PROCEDURE PROC(VAR N:INTEGER);
PROCEDURE LOCAL;
...
```

becomes-->

```
FUNCTION PROC(N:INTEGER):INTEGER;
FUNCTION LOCAL(A,B:INTEGER):INTEGER
...
```

```
PROC(A)                                   A := PROC(A);
LOCAL;                                    A := LOCAL(A,B);
```

where A is a frequently used local variable used as a var parameter to PROC, and used in nested procedure LOCAL. This method, although limited in application, is elegant because no temporary-variable assignments have to be inserted.

Improvement is two or more bytes per reference of the frequently used variable in the main procedure, less 2-8 bytes per extra assignment statement, subject to register competition as noted above. Since this optimization can be applied to very frequently used variables that are abandoned by the compiler, large optimizations of up to 40 or more bytes are possible in large procedures. Difficulty and likelyhood of errors with var parameter substitution is low; difficulty and likelyhood of errors with nested procedures is moderate to high.

7. Don't use the set construct to check ranges; instead use comparisons against the upper and lower bounds.

   Getting rid of the set construct is a BIG savings (typically around 30 bytes for the usual double-ended range check). Difficulty is minimal, as are the chances of error.

8. Do not pass multi-word (more than 4 bytes) data structures as non-var parameters unless necessary. Change them to VAR parameters.

   Improvement is 12-18 bytes saved by not having code to copy the parameter into local storage in the called procedure. Difficulty is low; likelyhood of errors is moderately low.

9. Replace FOR loops with WHILEs and REPEATs. The equivalent REPEATs and WHILEs are typically 8 to 10 bytes shorter, even with the explicit loop variable initialization and increments. REPEATs are more efficient than WHILEs which are better than FORs. Sometimes the savings will be greater depending on the contents of the loops and the termination condition.

   Savings are typically 8 to 14 bytes per construct. Difficulty and chances of error are small (just take care to get your termination condition correct -- beware of off-by-one errors).

10. Convert array indexing in loops to pointer arithmetic, when the total number of indexing operations can be reduced. For example

```
FOR I := 1 TO 100 DO A[I] := 3          converts to

P := @A;   {A's origin is 1;  P is typed as ^A[I]}
FOR I := 1 TO 100 DO
    BEGIN
    P^ := 3;
    P := POINTER(ORD(P)+SIZEOF({A's element type}));
```

END;

Improvement is up to 18 bytes per index operation (more when the array origin is nonzero or the array element size is not byte; savings can be even higher on packed structures if the programmer is willing to add a few more contortions); difficulty is moderate; likelyhood of errors is moderate.

11. IFs without ELSE parts that have a conjunctive conditional (IF a AND b THEN ...) are more efficiently expressed as nested IFs (IF a THEN IF b THEN ...). In effect, this implements your own "short circuit" boolean evaluation.

    The savings is typically 4 bytes for each AND eliminated. Very easy to implement. Just don't try it on ORs.

12. Avoid packed structures whenever possible. Remember, packing is only useful when a large amount of data has to fit in a limited space -- it does not decrease the size of the code.

    Improvement is highly variable and can be vast. Difficulty is low; likelyhood of errors is low if tricks like (10) do not pervade the code.

13. Repetition of expressions in the code should be removed by pre-assigning a common expression value to a temporary variable.

    Improvement is highly variable. Difficulty is moderate; likelyhood of errors is low.

14. Convert procedure parameters to global or local variables when the same actual value is always passed to the subroutine, and when there is no recursion.

    Improvement is 2-4 bytes per parameter saved. Beware of creating uplevel addressing of 'hot' variables however (see (6)). Difficulty is moderate; likelyhood of errors is low.

15. When groups of local or global variables are commonly passed together as parameters, and are not 'hot' (assigned to registers), they could be combined into a single record, which would then be passed as a var parameter to the subroutine.

    Improvement is 4 bytes per parameter, with an overhead of 8 bytes (warning, the called procedure may grow in size if it already uses all registers). Difficulty is moderate; likelyhood of errors is low.

16. If you have several instances of the same string constant in your code declare it as a CONST, otherwise the compiler will store multiple versions of the same constant.

    The savings depends on the size of the string and the number of occurences. Easy to do.

17. Turn range checking off after a sufficient amount of testing has

occurred.

Improvement is 4-8 bytes per reference or assignment of a range-checked quantity; difficulty is too low; likelyhood of errors is fairly high since a sufficient amount of testing never occurs. Consider making this change on a procedure-by-procedure confidence level basis.

How to crunch code: some case studies

The following section presents some case studies demonstrating some of the techniques presented in the previous section. These examples are intended to demonstrate how some of the transformational techniques are typically used and how a whole series of transformations may be applied to a single body of code. The main purpose of the examples, however, is to give a sense of the thought processes involved in crunching code.

If you have any good "before" and "after" examples demonstrating how fat code was reduced please feel free to contribute them. Your efforts may provide ideas and inspiration to others.

CASE 1:

Following is the original form of the body of a routine (SUUpCh in the StdUnit) which converts lower case characters to upper case. The code size for the original routine was 94 bytes.

```
IF Ch IN ['a'..'z'] THEN
  SUUpCh := CHR (ORD (Ch) - 32)
ELSE
  SUUpCh := Ch;
```

The code above was replaced with the following, which replaced the set range test with two comparisons. The code for this version of the procedure was 66 bytes -- a savings of 28 bytes (about 30%, or actually more, since these sizes include the overhead for the procedure and the assignment statements). The moral here is that SET OPERATIONS ARE EXPENSIVE.

```
IF ('a' <= Ch) AND (Ch <= 'z') THEN
  SUUpCh := CHR (ORD (Ch) - 32)
ELSE
  SUUpCh := Ch;
```

The following change was then made which saved another 2 bytes (bringing the procedure size down to 64 bytes) by getting rid of the branch for the ELSE logic on the IF statement.

```
SUUpCh := Ch;
IF ('a' <= Ch) AND (Ch <= 'z') THEN
```

```
  SUUpCh := CHR (ORD (Ch) - 32);
```

A further change -- breaking the AND in the IF into nested IFs -- resulted in a 4 byte savings, leaving the procedure size at 60 bytes (an improvement of 36% over the original 94 bytes). In effect this is performing "short circuit" boolean evaluation at the source level. The source for this version is as follows:

```
SUUpCh := Ch;
IF 'a' <= Ch THEN
  IF Ch <= 'z' THEN
    SUUpCh := CHR (ORD (Ch) - 32);
```

Note that this last transformation would not have worthwhile if we had not already removed the ELSE part of the IF since the nested IFs would have required two ELSEs.

CASE 2:

Below is the body of the original version of SUUpStr which uppercases a string.

```
FOR I := 1 TO LENGTH (S^) DO
  S^[I] := SUUpCh (S^[I]);
```

The following version -- converting the FOR loop to a WHILE -- saved 8 bytes.

```
I := 1;
WHILE I <= LENGTH (S^) DO
  BEGIN
    S^[I] := SUUpCh (S^[I]);
    I := I + 1;
  END;
```

A further, time-oriented optimization would be to perform the upper-casing in reverse order with the call to LENGTH outside the loop, which also simplifies the termination condition to a test for zero.

An aside: **When appropriate (when the loop body will be executed at least once) a REPEAT will save another 2 bytes. I tested the three constructs with three test procedures (t1, t2, t3) as follows:**

```
procedure t1;
  var
    j : integer;
  begin
```

```
    for j := 1 to i do
      foo := bar;
  end;
procedure t2;
  var
    j : integer;
  begin
    j := 1;
    while j <= i do
      begin
        foo := bar;
        j := j + 1;
      end;
  end;
procedure t3;
  var
    j : integer;
  begin
    j := 1;
    repeat
      foo := bar;
      j := j + 1;
    until j > i;
  end;
```

T2 (WHILE) saved 8 bytes over T1 (FOR), and T3 (REPEAT) saved 10 bytes over T1 (FOR).

CASE 3:

A series of small transformations was applied to the following segment of TrimLeading (which trims leading blanks and tabs from a string).

```
FOR I := 1 TO ORD (S^[0]) DO
  IF (S^[I] = SUSpace) OR (S^[I] = SUTab) THEN
    { skip over leading spaces }
  ELSE
    BEGIN
      DELETE (S^, 1, I - 1);
      EXIT (TrimLeading);
    END;
{ we fell thru -- either '' or all blanks }
...
```

The first change was to change ORD (S^[0]) to LENGTH (S^), which saved 4 bytes. (I must have thought I was being clever in the original.) Calling the built-in function saves code by leaving the array access to the built-in.

The next change was to get rid of the ELSE in the FOR loop by reversing the sense of the condition (which resulted in the code below). This last change resulted in no code size change since a short branch was removed but another logical operator was added. But this prepared us for some subsequent changes.

```
FOR I := 1 TO LENGTH (S^) DO
  IF NOT ((S^[I] = SUSpace) OR (S^[I] = SUTab)) THEN
    BEGIN { delete leading as soon as we find a non-blank char }
      DELETE (S^, 1, I - 1);
      EXIT (TrimLeading);
    END;
{ we fell thru -- either '' or all blanks }
...
```

The next step was to apply de Morgan's law (remember your boolean algebra?) to simplify the conditional to the following form which saved 2 bytes by reducing the number of boolean operations.

```
FOR I := 1 TO LENGTH (S^) DO
  IF (S^[I] <> SUSpace) AND (S^[I] <> SUTab) THEN
    BEGIN { delete leading as soon as we find a non-blank char }
      DELETE (S^, 1, I - 1);
      EXIT (TrimLeading);
    END;
{ we fell thru -- either '' or all blanks }
...
```

Now we have converted the conditional into a form in which we can apply our short-circuit evaluation transformation by converting the AND into nested IFs, which saves another 4 bytes.

```
FOR I := 1 TO LENGTH (S^) DO
  IF (S^[I] <> SUSpace) THEN
    IF (S^[I] <> SUTab) THEN
      BEGIN { delete leading as soon as we find a non-blank char }
        DELETE (S^, 1, I - 1);
        EXIT (TrimLeading);
      END;
{ we fell thru -- either '' or all blanks }
...
```

Finally we convert the FOR construct to a WHILE which saved another 8 bytes.

```
I := 1;
WHILE I <= LENGTH (S^) DO
  BEGIN
    IF S^[I] <> SUSpace THEN
      IF S^[I] <> SUTab THEN
        BEGIN { delete leading as soon as we find a non-blank char }
```

```
          DELETE (S^, 1, I - 1);
          EXIT (TrimLeading);
        END;
    I := I + 1;
  END;
{ we fell thru -- either '' or all blanks }
...
```

CASE 4:

The following is applicable only to programs using WRITEs and WRITELNs, but the general technique of factoring can be applied anywhere. The section of code below prints out the defaults (volume, file name, and extension) for a file name prompt.

```
IF DefVol <> '' THEN
  WRITE ('[', DefVol, '] ');
IF DefFN <> '' THEN
  WRITE ('[', DefFN, '] ');
IF DefExt <> '' THEN
  WRITE ('[', DefExt, '] ');
```

The following factoring out of the expensive WRITE operations resulted in a savings of 168 bytes.

```
PROCEDURE WriteDefault (DefaultValue : SUStr);
  BEGIN
    IF DefaultValue <> '' THEN
      WRITE ('[', DefaultValue, '] ');
  END;
...
WriteDefault (DefVol);
WriteDefault (DefFN);
WriteDefault (DefExt);
```

CASE 5:

"Factoring" of common code does not always pay off. Following is an instance of how space was saved removing factoring. The SUStrToInt conversion routine had an internal procedure called BogusNumber which set the value of the CState parameter to the appropriate error return code and then exited from SUStrToInt:

```
PROCEDURE BogusNumber (CS : ConvNState);
  BEGIN
```

```
    CState := CS;
    EXIT (SUStrToInt);
  END;
...
```

BogusNumber was called 6 times in the original SUStrToInt. By replacing the calls to BogusNumber with BEGIN CState := ErrCode; EXIT(SUStrToInt) END we got rid of the 50 byte BogusNumber routine and the size of SUStrToInt when down from 500 bytes to 380 bytes, a total saving of 170 bytes. The moral here is to CHECK YOUR FACTORING TO SEE THAT IT REALLY PAYS OFF.

The Last Whole Earth Text File Format

Fred Forsman

This is the latest proposal for the definition of text files. In creating this definition I had three (not always convergent) goals in mind.

1) Text files should support Pascal's model of files of type TEXT as well as possible -- that is, if a file was written by Pascal WRITEs and WRITELNs it should be a valid text file with as few exceptions as possible.

   The intent here is to give reasonable support to Pascal's TEXT mechanism as it is defined in the language -- while the language makes no statement about the form of TEXT files, one would expect that files written without errors will result in valid text files of some sort. This is not to say that all tools should support every perverse file that can be generated via Pascal text I/O. At a minimum, however, the Pascal run-time system should be as accomodating as possible in its support of Pascal TEXT I/O, and the editor should should make similar efforts since it is the device most often used to inspect text files (whether normal or aberrant).

2) To make the processing of text files as straightforward and efficient as possible.

3) To be compatible with the UCSD text file formats in the Pascal systems on the Apple II and Apple ///.

   The following definition follows the UCSD text file format fairly closely. The one or two deviations don't pose a very serious threat to compatibility since they involve abnormal cases which are not likely to be encountered or generated in normal practice.

The following definition involves compromises to all of the above goals. The determination of which goal has been most violated I leave as an exercise to the reader.

The definition of a text file:

- A text file is a sequence of 1024-byte pages.
- One 1024-byte header page is present at the beginning of the file. This is not considered to be part of the actual contents of the text file, but is used by the editor to store formatting information, etc. Anyone creating a header page should do so with nulls in all 1024 bytes, unless there is a good reason to do otherwise. (The format and interpretation of the header page will be described in a forthcoming document.)

- Each text page (i.e., those following the header page) contains some number of *complete* lines of text and is filled with null characters (ASCII 0) after the last line.

  The Pascal run-time system should ensure that all text files end with a CR when CLOSEd, in particular, dealing with the case where the last action before the CLOSE was a WRITE instead of a WRITELN. Similarly, the run-time system should also ensure that pages terminate with CRs even if inordinately long lines are written by a series of WRITEs without any WRITELNs (however determining when to insert a CR can be a tricky issue). (For more on related issues, see the following two points.)

- The end of a text page must terminate with at least one null. For simplicity, the first instance of a CR-null sequence will signal the end of the page.

  As a consequence of this simplifying assumption, a WRITELN followed by a WRITE (CHR (0)) will inadvertently terminate the current page, but anyone writing nulls to a text file is living in a state of sin and deserves what they get.

  To be on the safe side, code dealing with text files at the BLOCKREAD level should not assume that a final CR-null always exists, making sure not to run off the end of page buffers. Our tools should not blow up on invalid input.

- A line is a sequence of zero or more characters followed by a *CR*. A line may be "arbitrarily long" (1023 bytes long, counting the CR, with room for a terminating null at the end of the page) but programs (such as development system tools) may choose to consider as significant only the first *N* characters (where *N* is a reasonable and well documented number, i.e., either 132 or 255).

  The Pascal run-time system should allow the reading and writing of arbitrarily long lines. The contents of a long line should be obtainable via a series of READs. The action of READLN should be to read past the next CR, returning an IORESULT warning value if characters are skipped in the process.

  Support of "arbitrarily long" lines should not be viewed as a threat to tool implementors. Tools may have reasonable restrictions on what text files they choose to accept, as long as they don't blow up on other text files. Tools may choose to ignore the excess on unreasonably long lines, give a warning, or signal an error and abort processing.

- A sequence of spaces at the beginning of a line *may* be compressed into a two-byte code, namely a *DLE character* (ASCII 16) followed by a byte containing 32 plus the number of spaces represented.

- A null text file (i.e., one which has no contents -- as might be created by opening a file and then closing it before anything is written to it) consists of only the 1024-byte header page.

Pascal's Packing Algorithm

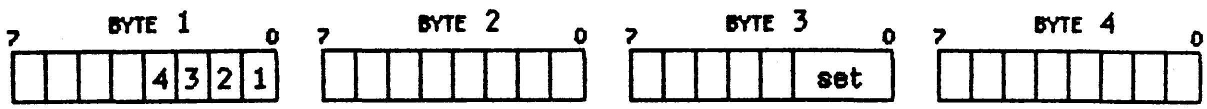
Packed Records

Packed records are very expensive in terms of the number of bytes of code generated by the compiler to reference a particular field. In general, you should avoid packing records unless there will be many more instances of the record than there are references to it. Packed records are packed in the following bizarre way:

1. Fields are packed as tightly as possible without crossing word boundries, starting at the low-ordered bit of the first byte. (Note that in a packed record, a character or 0..255 fits into a byte.) Records will always occupy either one byte or an even number of bytes.

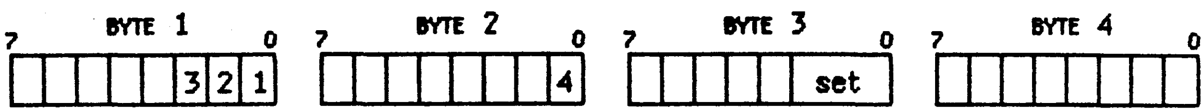
   Note that only scalar values and subranges are considered packable; everything else must go on a word boundry.

   For example, 4 booleans and a set are packed as follows:



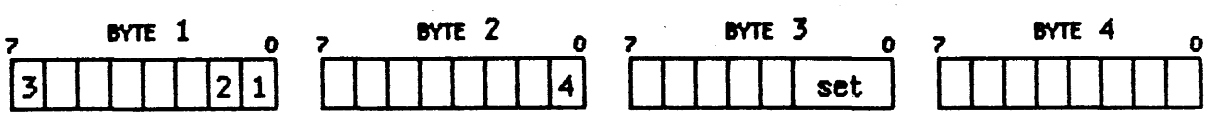
2. Any empty bytes are filled by moving the previous field into the empty byte if:

   - The field fits into a byte.
   - The field was not previously on a byte boundry.



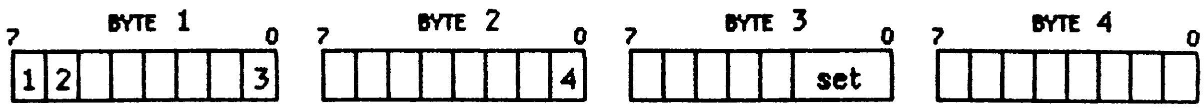
3. Any field that fits in a byte or word and does not share that space with other fields is now designated "unpacked".

   Any field that is still considered "packed," and is closest to the high end of a byte or word, is moved to the high end of that space.



4. The last field is treated after steps 2 & 3 have been completed on the other fields.

5. Finally, bytes containing packed fields are flipped (bits reordered).



The following is a (slightly) simpler description of what appears to happen when packed records are packed, if you don't need to know the actual process.

1. Fields are packed as tightly as possible without crossing word boundries, starting at the high-ordered bit of the first byte.

   All packed records take up either one byte or an even number of bytes.

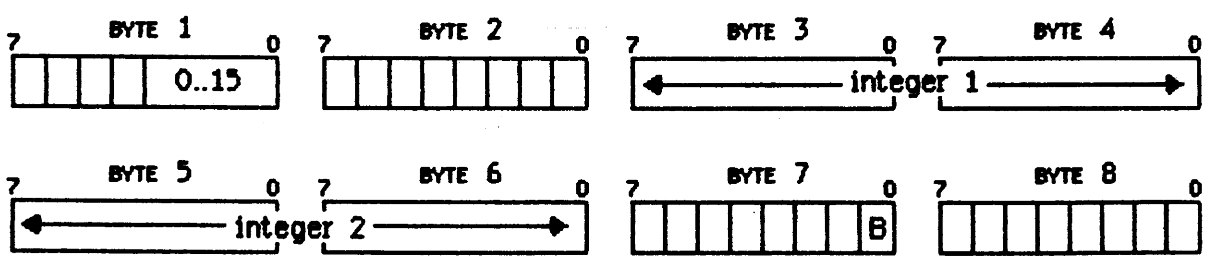
   Only boolean or subrange types can be packed; all other types start on word boundries, so steps 2 and 3 only apply to these types.
2. If a byte would be left empty (so the next field can start on a word boundry), and there is more than one field in the previous byte, the last (low-ordered) field is moved into the empty byte.
3. The last (low-ordered) field in any byte with unused space is moved to the low end of the byte. (This happens even if it's the only field in the byte.)

Unpacked Records

Fields of unpacked records are packed in order, starting on word boundries, except for booleans and subranges that can fit in a byte. Values that don't take up a full byte or word will be packed at the low-ordered end of that space.

The whole record will take up either one byte or an even number of bytes.

For example, a record containing a subrange of 0..15, two integers, and a boolean would be packed as follows:



Packed Arrays

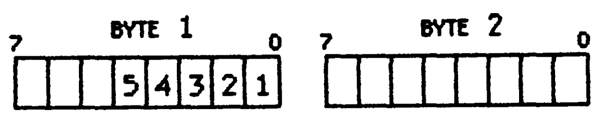
Packed arrays are also code-expensive, except for packed arrays of char. (These are treated as a special case, and the code associated with them is compact.)

The number of bits per element in a packed array is the smallest of 1,2,4,8 or 16 bits that will accommodate the element. For example, a subrange of column A requires the number of bits per element in column B:

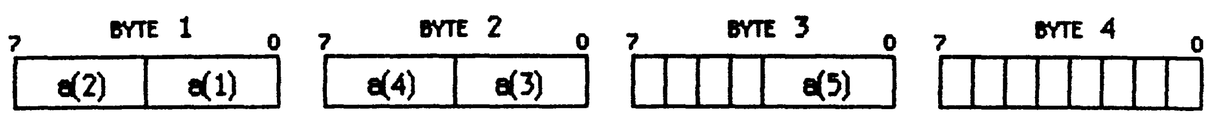
A	B
0..1	1
0..2	2
0..3	2
0..4	4
0..10	4
0..20	8
0..255	8
0..395	16

Booleans are packed one boolean per bit. The packed array as a whole must occupy an even number of bytes.

A packed array[1..5] of boolean would be packed as follows:



A packed array[1..5] of [0..6] would be packed as follows:



You can use the @ operator to poke around inside any packed value and thereby discover what the packing algorithm (probably) is.

Signed Subranges

Signed subranges (e.g. -5..14) are packed in packed types (unlike UCSD Pascal, which won't pack them). The minimum field size for a signed subrange is the minimum number of bits needed to represent any number of the subrange in two's complement form.

The minimum field size is then subject to the rules for a particular packed type. For example, though -1..2 only needs three bits, if it's in a packed array, it will take up four (see above table). If it's in a packed record, on the other hand, it might take up only three bits, or it might use a whole byte, depending on what's packed around it.

NOTE

A variable of type **-127..128** takes up a *byte*.
A variable of type **0..255** takes up a *word*.
A variable of type **char** takes up a *word*.

PASLIB Procedure Interface

(Workshop Release 1.0)

PASLIB is the Pascal run-time support library. It provides the procedures and functions that are built into the Pascal language, acts as the run-time interface to the Operating System, and "completes" the 68000 instruction set by providing routines for the compiler-generated code to call upon in lieu of actual hardware instructions.

The interface to PASLIB is very tightly coupled with the Pascal compiler, and is very likely to be changed to improve performance and reduce code size. For this reason, only call these routines from assembly language if you absolutely and positively have to; stay in Pascal as much as possible when dealing with PASLIB. Most of these routines support the Pascal built-in procedures, which are described in detail in the *Pascal Reference Manual*.

There are a few conventions for using these routines, which must be followed to ensure correct results and successful execution. All the routines are called with parameters passed on the stack. The parameters are pushed onto the stack in the order of the parameter list shown in each routine. 'ST.L' indicates a four-byte parameter, 'ST.W' two-byte, 'ST.B' one-byte (stored in the upper byte of a word), and 'ST.S' a set. The parameters passed will be popped by these routines before return. The function results, if any, will be returned on the stack after the parameters are popped out. Note that the function-type routines do not expect room for the function result to be reserved on the stack before the call. Also note that these routines do not check for room on the stack; the caller must guarantee enough room on the stack for saved registers. The caller should follow the Pascal procedure preamble code for expanding the stack before calling these routines. Standard register preservation conventions are followed except in the routines indicated. Refer to the *Workshop User's Guide* for the usage of the special registers and the stack frame allocation.

Contents

1. Initialization and Termination Routines: %_BEGIN, %_END, %_INIT, %_TERM

None of these routines have parameters, return values, or destroy any registers.

Every main program must have the following beginning and ending sequences calling these routines:

```
JSR      %_BEGIN       ; beginning sequence
LINK     A6,#$0000     ; no-op for LisaBug, to look like standard module
                         head
MOVE.L   (A7)+,A6
LINK     A5,#$0000     ; set up global frame for main program
SUBA.L   $0010(A5),A7  ; variables for units, etc. passed by loader
JSR      %_INIT
.
.                      ; main program code goes here
.
JSR      %_TERM        ; ending sequence
UNLK     A5
JSR      %_END
RTS
UNLK     A6            ; no-op for LisaBug, to look like standard module
                         tail
RTS
```

Note that the size of the program global variables allocated to the loader is offset +16 from register A5.

%_BEGIN - Beginning routine. Currently a no-op; reserved for future extensions.

%_END - Ending routine. Currently a no-op; reserved for future extensions.

%_INIT - Initializes PASLIB internal global data for each process:

1. Sets up an f-line trap routine, which signals a "sys_terminate" exception if an f-line trap is encountered in the user code, terminating the program.
2. Sets up global input and output file buffer addresses. These buffers are used for screen, keyboard, exec files and output redirection. The address locations are fixed on the stack: the input buffer address is offset +8 from register A5; the output buffer address is offset +12. They are set up to point to global

file buffers in the shared data area of PASLIB.

3. Initializes the OS exception handlers.

4. Initializes the Pascal heap local variables.

NOTE: The %_INIT routine will restart at step 5 if the calling process is a resident process.

5. Initializes the PASLIB local variables.

6. If the floating-point library IOSFPLIB is linked, it is initialized.

%_TERM - Terminate. If the process is resident, it jumps to step 5 of %_INIT (see above), if not, it calls the OS routine "Hit_End" to terminate the process. Control does not return after this call.

2. Integer Arithmetic Routines: %I_MUL4, %I_DIV4, %I_MOD4

%I_MUL4 - Multiply two 4-byte integers

Parameters: ST.L - Argument 1
ST.L - Argument 2

Returns: ST.L - Product

Registers used: All registers are preserved.

The multiplication algorithm is as follows:

-argument 1's upper word is multiplied by argument 2's lower word.
-argument 2's upper word is multiplied by argument 1's lower word.
-these two products are added, and the sum is put in the result's upper word.
-the two arguments' lower words are multiplied, and this value is put in the result's lower word.

%I_DIV4 - Divide two 4-byte integers

Parameters: ST.L - Dividend
ST.L - Divisor

Returns: ST.L - Quotient

Registers used: All registers are preserved.

The division is performed by subtracting the dividend from the divisor 31 times (for each of the 32 bits except the sign bit).

%I_MOD4 - Remainder from the division of two 4-byte integers

Parameters: ST.L - Dividend
ST.L - Divisor

Returns: ST.L - Remainder

Registers used: All registers are preserved.

The division is performed in the same way as %I_DIV4, above.

3. Data Move and Scan Routines: %_MOVEL, %_MOVER, %_FILLC, %_SCANE, %_SCANN

%_MOVEL - Moveleft

Parameters: ST.L - From Address
ST.L - To Address
ST.W - Number of bytes to move

Returns: ---

Registers used: D0, D1, D2, A0, A1, A2

If the number of bytes to move is 7 or less, they are moved a byte at a time. If the source address + 2 is the destination address, the data is moved one word at a time. If there are more than 7 bytes to be moved, then data is moved a long word at a time. If the ending address is a byte address, the trailing byte is moved.

%_MOVER - Moveright

Parameters: ST.L - From Address
ST.L - To Address
ST.W - Number of bytes to move

Returns: ---

Registers used: D0, A0, A1, A2

Data is moved one byte at a time.

%_FILLC - Fillchar

Parameters: ST.L - Address to fill
ST.W - Number of bytes to fill
ST.W - Fill character

Returns: ---

Registers used: D0, D1, A0, A2

Fills the address with the given character one byte at a time.

%_SCANE - Scan equal

Parameters: ST.W - Length to scan
ST.W - Character to scan for
ST.L - Address to scan

Returns: ST.W - The position of the character (0 being the first)

Registers used: All registers are preserved.

Scans the string for the given character, one byte at a time.

Note that "Length to scan" can be negative, and the scan will go in the lower address direction.

%_SCANN - Scan not equal

```
Parameters: ST.W - Length to scan
            ST.W - Character to scan for
            ST.L - Address to scan

Returns:    ST.W - The first character position that is not equal
                   to the character to scan for (0 being the
                   first)
```

Registers used: All registers are preserved.

Scans the string for the first character not equal to the given character, one byte at a time.

Note that "Length to scan" can be negative, and the scan will go in the lower address direction.

4. String Manipulation Routines: %_CAT, %_POS, %_COPY, %_DEL, %_INS

All the string manipulation routines are performed one byte at a time.

%_CAT - Concatenate

```
Parameters: ST.L - Address of 1st string
            ST.L - Address of 2nd string
            ...
            ST.L - Address of Nth string
            ST.L - Address to put result
            ST.W - N

Returns:     ---
```

Registers used: All registers are preserved.

Copies all the given strings to the result string.

%_POS - Position of one string in another

Parameters: ST.L - Address of substring
ST.L - Address of main string

Returns: ST.W - Position

Registers used: All registers are preserved.

Compares the substring with the main string until a match is found. If no match is found, 0 is returned.

%_COPY - Copy a substring

Parameters: ST.L - Source string address
ST.W - Starting index
ST.W - Size to copy
ST.L - Address of result

Returns: ---

Registers used: All registers are preserved.

If the number of bytes to copy is 0, or if the source string is longer than the number of bytes to copy, the result string has 0 lenth.

%_DEL - Delete a substring from a string

Parameters: ST.L - Address of string
ST.W - Position to start deleting
ST.W - Number bytes to delete

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%_INS - Insert one string in another

Parameters: ST.L - Address of string to insert
ST.L - Address of main string
ST.W - Position in main string to insert

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

5. String Comparison Routines: %S_EQ, %S_NE, %S_LE, %S_GE, %S_LT, %S_GT

All the string comparison routines are performed one byte at a time.

%S_EQ - String equal
%S_NE - String not equal
%S_LE - String less than or equal
%S_GE - String greater than or equal
%S_LT - String less than
%S_GT - String greater than

Parameters: ST.L - Address of first string
ST.L - Address of second string

Returns: ST.B - Boolean result

Registers used: All registers are preserved.

6. Set Manipulation Routines: %_INTER, %_SING, %_UNION, %_DIFF, %_RDIFF, %_RANGE, %_ADJ, %_SETGE, %_SETLE, %_SETEQ, %_SETNE

The format of a set on the stack is:

```
+----------+  high address
| 15 - 0   |
+----------+
| 31 - 16  |
+----------+
|   ...    |
+----------+
|last word |
+----------+
| # Bytes  |
+----------+  low address
```

```
%_INTER - Set intersection:  set1 AND set2
%_UNION - Set union:  set1 OR set2
%_DIFF  - Set difference:  set1 AND (NOT set2)
%_RDIFF - Reverse set difference:  (NOT set1) AND set2
```

Parameters: ST.S - First set
ST.S - Second set

Returns: ST.S - Result set

Registers used: All registers are preserved.

%_SING - Singleton set

Parameters: ST.W - Singleton value

Returns: ST.S - Result set

Registers used: All registers are preserved.

%_RANGE - Set range

Parameters: ST.W - Minimum value
ST.W - Maximum value

Returns: ST.S - Result set

Registers used: All registers are preserved.

Returns the set representation of the values from minimum to maximum. If minimum is greater than maximum, a null set is returned.

%_ADJ - Set adjust

Parameters: ST.S - Set
ST.W - Desired size in bytes

Returns: ST.S' - Adjusted set without size word

Registers used: All registers are preserved.

Changes the size of a set to the given size. If the set is larger than the desired size, the extra values are thrown out; if the set is smaller than the desired size, extra fields are added and initialized to 0.

%_SETNE - Set inequality test
%_SETEQ - Set equality test
%_SETGE - Set inclusion test (returns **true** if set2 is the same as or included in set1)
%_SETLE - Set inclusion test (returns **true** if set1 is the same as or included in set2)

Parameters: ST.S - First set
ST.S - Second set

Returns: ST.W - Boolean Result

Registers used: All registers are preserved.

7. Miscellaneous Routines: %_GOTOXY, %_GOTO, %_HALT

%_GOTOXY - Move the cursor to a specified location

Parameters: ST.W - X coordinate
ST.W - Y coordinate

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%_GOTOXY sends the following escape sequence to the screen to move the cursor position: ESC
=
Y+32
X+32

Y values are between 0 and 31; X values between 0 and 79. If the coordinate given is outside these bounds, it is set equal to the boundry value.

%_GOTO - Global GOTO code segment remover

Parameters: ST.L - Pointer to the desired last-segment jump table

Returns: ---

Registers used: A0

Jumps from a nested routine to the first-level process.

%_HALT - Halt

If the process is resident, it goes to step 5 of the %_INIT routine. If not, it calls "terminate_process" with the value of event_ptr as **nil**. Control does not return after this call.

8. Range Check Routines: %_RCHCK, %_SRCHK

%_RCHCK - Range check, to check the bounds of subrange type variables

Parameters: ST.W - Value to check
ST.W - Lower bound
ST.W - Upper bound

Returns: ---

Registers used: All registers are preserved.

Note that if the check fails, this routine causes the system exception 'SYS_VALUE_OOB' to be signalled and the message 'VALUE RANGE ERROR' to be displayed before the process is forced to enter the debugger. If the process has not declared an exception handler for this exception, the system default handler is entered after the debugger returns control. The system default handler terminates the process.

%_SRCHK - String range check, to check a string index against its length

Parameters: ST.B - Value to check: 0..255
ST.W - Upper bound

Returns: ---

Registers used: All registers are preserved.

Note that if the check fails, this routine causes the system exception 'SYS_VALUE_OOB' to be signalled and the message 'ILLEGAL STRING INDEX' to be displayed before the process is forced to enter the debugger. If the process has not declared an exception handler for this exception, the system default handler is entered after the debugger returns control. The system default handler terminates the process.

9. Heap Routines: %_NEW, %_MARK, %_RELSE, %_MEMAV, %_HEAPRES

%_NEW - The New procedure. Allocate memory in the Pascal heap.

Parameters: ST.L - Address of pointer
ST.W - Number of bytes needed

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%_NEW sets the address of the pointer to **nil**.

%_NEW checks whether the heap has been initialized (whether a data segment has been allocated) via the boolean HeapInited. If HeapInited is **false**, a call is made to the GrowHeap function to create and initialize a 'new heap'. If GrowHeap is unsuccessful (returns **false**) then %_NEW is exited with the pointer set to **nil**.

The GrowHeap function initializes a 'new heap' by calling the PLInitHeap procedure. Growheap passes PLInitHeap the size of the Pascal heap data segement, the memory size (HeapDelta) and the logical data segment number (LDSN = 5). PLInitHeap then creates a private data segment with the pathname PascalLHeap, and assigns the segment pointer address to the pointers HeapStart and HeapPtr. PLInitHeap sets the pointer HeapEnd to point to the end of the segment (HeapStart + segment size - 256).

Before assigning an address to the pointer, %_NEW determines whether there is enough room on the heap (i.e. in the data segment) for the variable. %_NEW makes a second call to the GrowHeap function. If GrowHeap is unsuccessful, then %_NEW is exited with the pointer set to **nil**.

The GrowHeap function calls the GetSafeAmmount procedure to determine the maximum number of bytes by which the heap can be increased (the amount of system memory available to the calling process). If this amount is greater than the current size of the heap, then GrowHeap will double the size of the heap, otherwise GrowHeap will increase the heap to the maximum amount available. The pointer HeapEnd is incremented by the amount of increase.

%_NEW then sets the address of the pointer to the address of HeapPtr, which points to the next free area on the heap. The address of HeapPtr is increased by the size of the variable that was placed on the heap.

%_MARK - The Mark procedure. Mark the Pascal heap.

Parameters: ST.L - Address of pointer to be marked
ST.W - Number of bytes needed

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%_MARK checks whether the heap has been initialized via the boolean HeapInited. If HeapInited is **false**, a call is made to the GrowHeap function to create and initialize a 'new heap'. If the function is unsuccessful (returns **false**) then %_MARK is exited.

The GrowHeap function is described under %_NEW, above.

%_MARK sets the address of the pointer to the address of HeapPtr, which points to the next free area on the heap.

%_RELSE - The Release procedure. Release the Pascal heap.

Parameters: ST.L - Address of pointer to release to.

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%_RELSE checks whether the heap has been initialized via the boolean HeapInited. If HeapInited is **false**, a call is made to the GrowHeap function to create and initialize a 'new heap'. If GrowHeap is unsuccessful (returns **false**) then %_RELSE is exited.

The GrowHeap function is described under %_NEW, above.

If the pointer does not point within the heap (i.e., address memory between HeapStart and HeapEnd), an error will result and the procedure will be exited.

If the pointer is less than HeapEnd minus HeapDelta, (where HeapDelta is the original size of the heap) the heap is reduced in size by HeapDelta.

%_RELSE sets HeapPtr (which points to the next free area on the heap) to the address of the pointer.

%_MEMAV - The Memavail function. Memory Available in the Pascal heap.

Parameters: None.

Returns: ---

Registers used: All registers are preserved.

%_MEMAV generates a call to the %_PHWordsAvail function, which determines the amount of words available.

%_PHWordsAvail checks whether the heap has been initialized via the boolean HeapInited. If HeapInited is **false**, a call is made to the GrowHeap function to create and initialize a 'new heap'. If GrowHeap is unsuccessful (returns **false**) then %_PHWordsAvail is exited.

The GrowHeap function is described under %_NEW, above.

%_PHWordsAvail determines the maximum number of words available (the amount left in the heap data segment minus the maximum amount of system memory available) and the current number of LDSN words available (the maximum number of words you can get by the chosen LDSN minus the number of words already used). If the maximum number of words available is greater than the current number of LDSN words available, then the current number of LDSN words available is returned, otherwise the maximum number of words available is returned.

%_HEAPRES - The HeapResult function.

Parameters: ST.W - Heap result

Returns: ---

Registers used: All registers are preserved.

Refer to the *Workshop User's Guide* for the values of the heap result.

%_HEAPRES generates a call to the %_HHeapRes function. %_HHeapRes is assigned the integer value of HErrResult.

10. Read and Write Routines: %_KEYPRESS, %W_LN, %W_C, %W_STR, %W_PAOC, %W_I, %W_B, %_PAGE, %R_C, %R_I, %R_STR, %R_PAOC, %R_LN, %_EOLN

All the read and write routines take 'file address' as a parameter, which is the address of the file variable. The address of the Pascal standard **input** is in offset 8 from register A5; the address of **output** is in offset 12 from A5.

%_KEYPRESS - The Keypress function.

Parameters: ST.L - File address

Returns: ST.B - Boolean Result

Registers used: All registers are preserved.

Note that the file address is not used in the current implementation.

%_KEYPRESS generates a call to the %_PKeyPress function and returns the result of %_PKeyPress as its result.

The %_PKeyPress function determines whether any keys have been pressed. It returns **true** if the look-ahead buffer is full, otherwise it returns **false**.

%W_LN - WriteLn

Parameters: ST.L - Address of output file

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%W_LN calls the FWriteln procedure, passing it the address of the file. FWriteln calls the FWriteChar procedure, passing it an ASCII <CR> (end-of-line) to be appended to the string.

%W_C - WriteChar. Display a character on the console.

Parameters: ST.L - Address of output file
ST.B - Character to be output
ST.W - Size of field to print

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%W_C calls the FWriteChar and OutCh procedures to write a character to the file. %W_C passes OutCh the character to be written and the address of the output file. OutCh then calls FWriteChar to write the character to the file.

The default field size is 1. If the field size is greater than 1, %W_C calls FWriteChar to write out the appropriate number of spaces, then calls OutCh, which calls FWriteChar to write the character.

%W_STR - Write string

Parameters: ST.L - Address of output file
ST.L - Address of string
ST.W - Size of field to print

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

If the string size is greater than 255 characters, then %W_STR truncates it to 255.

%W_STR then compares the field size (MinWidth) to the specified string size. If the field size is less than or equal to zero, it's set to the string size. If the field size is less than the string size (but greater than zero), then the string size is set to the field size. If the field size is greater than the string size, then a call is made to the FWriteChar procedure to write out [MinWidth minus string size] spaces.

%W_STR then calls FWriteChar to write out the string with the specified string size.

%W_PAOC - Write a packed array of characters

Parameters: ST.L - Address of output file
ST.L - Address of string
ST.W - Actual length
ST.W - Size of field to print

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

The effect of %W_PAOC is the same as calling %W_STR with the specifed field size equal to the number of elements in the array.

%W_I - Write an integer

Parameters: ST.L - Address of output file
ST.L - Value to print
ST.W - Size of field

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%W_I compares the field size (MinWidth) to the size of the integer. If the field size is greater than the size of the integer, then %W_I calls the FWriteChar procedure to write out [MinWidth minus integer size] spaces.

%W_I then calls FWriteChar to write out the integer with the specified integer size.

%W_B - Write a boolean

Parameters: ST.L - Address of output file
ST.B - Value to print
ST.W - Size of field

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%W_B calls the %W_STR procedure, passing it the string to be written, the size of the string, and the address of the output file.

If 'value to print' is zero, %W_B passes the string 'FALSE' to %W_STR, with a string size of 5.

If 'value to print' is 1, %W_B passes the string 'TRUE' to %W_STR, with a string size of 4.

%W_STR then writes the string to the output file.

%_PAGE - Page procedure

Parameters: ST.L - Address of output file

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%_PAGE writes the ASCII character 'FF' to the output file by calling the OutChar procedure. OutChar is passed the character to be written (e.g. 'FF') and the address of the output file.

%R_C - ReadChar

Parameters: ST.L - File Address

Returns: ST.B - the character read

Registers used: D0, D1, D2, D3, A0, A1, A2

%R_C reads a character from the specified file by calling the InCh function, then returns the character on the stack.

InCh calls the FReadChar function, passing it the file address.

FReadChar verifies that the file has been opened, calls the FGet procedure, reads the character that is placed in the window buffer area by FGet, and passes the character back to InCh.

%R_LN - ReadLn

Parameters: ST.L - Address of input file

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%R_LN reads a line from the specified file by calling the FReadLn procedure, passing it the file address.

FReadLn verifies that the file has been opened and then calls the FGet procdure to read each character on the line until EOLN is **true**. When EOLN is **true**, FReadLn resets EOLN to **false** and returns to %R_LN.

%R_PAOC - Read Packed Array of Character

Parameters: ST.L - File Address
ST.L - Array Address
ST.W - Size of array in bytes

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

The effect is the same as calling %R_STR whose specified field is the number of elements in the array.

%R_STR - Read String

Parameters: ST.L - File Address
ST.L - String Address
ST.W - Max size of string

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%R_STR first verifies that EOLN is **false**, otherwise %R_STR returns to the calling routine.

%R_STR then generates a loop which reads a character from the file by calling the InCh procedure (described under %R_C, above), then checks whether EOLN is **true**. If EOLN is **true**, %R_STR returns to the calling routine. If EOLN is **false**, %R_STR reads the character and returns to the beginning of the loop to read the next character.

After InCh returns a character, %R_STR checks whether the character is a RUBOUT (ASCII 'DLE') or BACKSPACE (ASCII 'BS'). If the character is either of the two, %R_STR processes the character accordingly and then reads the next character. If the character is not RUBOUT or BACKSPACE, the character is read and %R_STR returns to the beginning of the loop to read the next character.

%R_I - Read Integer

Parameters: ST.L - File Address

Returns: ST.B - The integer read

Registers used: D0, D1, D2, D3, A0, A1, A2

%R_I consists of two main loops which reads characters from the file to form a valid representation of an integer value.

The first loop reads a character from the file by calling the InCh procedure (described under %R_C, above). If this character is <CR> or space, %R_I returns to the beginning of the loop to read the next character. If the character is not <CR> or space, %R_I exits the first loop.

Next, %R_I determines whether the character read is a sign character ('+' or '-'). If it is, %R_I enters the second loop and calls InCh to read the next character. If the character is not a sign character, %R_I enters the second loop bypassing the call to InCh.

The character is then checked to see if it's a RUBOUT or BACKSPACE character; if it is, the character is processed accordingly and %R_I returns to the beginning of the first loop.

The character is checked once more to determine if it is a valid integer value (0 ≤ character ≤ 9). If it is, %R_I returns to the beginning of the second loop and calls InCh to read the next character.

If the character is not a valid integer, then %R_I checks to see if any characters read previously have been valid integers (by checking register D6). If no characters have been valid integers (D6 = 0), then %R_I generates an IOResult error. If the characters read previously have been valid integers (D6 =1), then %R_I returns to the calling routine with an integer result.

%_EOLN - End of line predicate

Parameters: ST.L - File address

Returns: ST.B - Boolean Result

Registers used: All registers are preserved.

%_EOLN returns **true** if the end of a line has been reached in the specified file.

11. File I/O Routines: %_REWRT, %_RESET, %_CLOSE, %_EOF, %_BLKRD, %_BLKWR, %_IORES, %_GET, %_PUT, %_UPARR, %_SEEK

%_REWRT - Rewrite a file

Parameters: ST.L - File Address
ST.L - Address of Name String
ST.W - Kind: -2=text, -1=file, >0=number of words per record

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

Creates and opens a new file.

%_REWRT first initializes the file's FIB (file identification block) by making a call to FInit and passing it the file type via the parameter recBytes. Once the file type is determined, the value of FRecSize is initialized. The values of recBytes and FRecSize and the file types are:

recBytes	file type	FRecSize
-2	text	-1
-1	untyped	0
0	interactive	-1
>0	typed	value in recBytes

Other important FIB entries are initialized as follows:

```
FIsOpen   := false  .. The file is marked as not open
FNewFile  := false  .. The file is marked as not new
                       (i.e. no creation of new files)
FEOF      := true   .. End Of File is set to true
FEOLN     := true   .. End Of Line is set to true
FModified:= false   .. The file is marked as not modified
FIsOS     := true   .. The file is marked as an OS File
```

%_REWRT then calls FOpen. Within FOpen:

A check is made to determine whether the file has been opened by referencing the boolean FIsOpen. If FIsOpen is **true**, an IOResult error will occur; if not, it is set to **true**.

FOpen then determines whether the filename is one of the character devices CONSOLE, KEYBOARD, or PRINTER. If it is, FOpen opens the file. If the filename is PRINTER, a check is made to determine if the printer is connected. If the printer is not connected, an IOResult error will be generated. The FIB variable FUnit is also set accordingly: 1=CONSOLE, 2=KEYBOARD, 3=PRINTER, 10=other devices (not pseudo devices).

The FIB variable FNewFile is set to **true** to indicate that a new file is being created with a rewrite, otherwise its value would remain **false** indicating a reset operation.

FOpen creates and opens a new temporary file if the filename does not exist (i.e. if FNewFile is **true**), otherwise it opens the existing file. If the temporary file is of type TEXT, FOpen writes two header blocks of null to the file. FOpen also kills the temporary file so that it may be unkilled during the close.

%_RESET - Reset a file

Parameters: ST.L - File Address
ST.L - Address of Name String
ST.W - Kind: -2=text, -1=file, >0=number of words per record

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

Opens an existing File.

%_RESET behaves in the same manner as %_REWRT, by making calls to procedures FInit and FOpen. However, %_RESET does not create a temporary file (FNewFile is **false**). It attempts to open the existing file and if it is unsuccessful will issue an IOResult error.

Before exiting FOpen, %_RESET makes a call to the FReset procedure which in turn calls the FGet procedure. This has the effect of advancing the file position to the first record of the file.

%_CLOSE - Close a file

Parameters: ST.L - File Address
ST.W - Mode: 0=NORMAL, 1=LOCK, 2=PURGE, 3=CRUNCH

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

If the file is a character device (e.g. console, keyboard) or if the file is not open (FIsOpen is **false**), the close procedure has no effect.

CRUNCH and LOCK Options:

If the close option is either CRUNCH or LOCK, and the file is a text file that had been opened by RESET (FNewFile is **false**), a check will be made to determine if the number of blocks is odd. If it is, a null block will be written to the end of the file.

If a previously existing file was opened by REWRITE (FNewFile is **true**), it will be killed (i.e. deleted). Its temporary file, which was killed by FOpen, is unkilled using the original file name as the new file name.

PURGE Option:

If the file was created by REWRITE, the temporary file will have already been killed in FOpen.

The PURGE option will kill the original file provided it was opened by RESET (FNewFile is **false**).

NORMAL Option:

If the file was created by REWRITE, the temporary file will have already been killed in FOpen.

The original file is left untouched.

%_EOF - End of file predicate

Parameters: ST.L - File address

Returns: ST.B - Boolean Result

Registers used: All registers are preserved.

Detects the end of a file by referencing the FIB boolean entry, FEOF.

%_BLKRD - Blockread

```
Parameters: ST.L - File Address
            ST.L - Buffer address
            ST.W - Number of blocks to read
            ST.W - Block Number, -1 = Sequential
   ******   ST.W - DoRead, 0 = write, 1 = read     *****
```

Returns: ST.W - Number of blocks actually read

Registers used: D0, D1, D2, D3, A0, A1, A2

%_BLKRD generates a call to the FBlockIO function, passing the parameters listed above. The boolean variable DoRead is set to **true** for Blockread and **false** for Blockwrite.

Within FBlockIO:

If the file is not open (FIsOpen=**false**) and the number of blocks to transfer is less than zero, FBlockIO will generate an IOResult error and the file will not be processed.

If the file is the character device CONSOLE or KEYBOARD, an IOResult error will be generated and the file will not be processed.

If the file is the character device PRINTER, the block number to start the transfer (RBLOCK) is set to -1.

If the boolean DoRead is **true**, FBlockIO reads blocks from the file via a READ_DATA call, otherwise FBlockIO writes blocks to the file via a WRITE_DATA call.

Before these OS calls can be made, the mode and offset must be determined.

If the block number to start the transfer (RBLOCK) is less than zero, the mode is SEQUENTIAL and the offset is zero, otherwise the mode is ABSOLUTE and the offset is calculated as:

```
ord4(rblock) * FBlkSize
```

where FBlkSize is the Standard Disk Block Length (512)

The number of blocks actually read or transferred is calculated as:

```
FBlockIO := actual div FBlkSize
```

where 'actual' is the number of bytes transferred by the READ_DATA or WRITE_DATA OS calls.

EOF (FEOF) is set to **true** when the last block is read.

%_BLKWR - Blockwrite

```
Parameters: ST.L - File Address
            ST.L - Buffer address
            ST.W - Number of blocks to write
            ST.W - Block Number, -1 = Sequential
   ******   ST.W - DoRead, 0 = write, 1 = read    *****

Returns:    ST.W - Number of Blocks actually written
```

%_BLKWR behaves in the same manner as %_BLKRD, except it passes the boolean variable DoRead with a value of **false** when calling FBlockIO.

%_IORES - IOResult

Parameters: None

Returns: ST.W - IOResult

Registers used: All registers are preserved.

Refer to the *Workshop User's Guide* for the values of IOResult.

Returns an integer value that reflects the status of the last completed I/O operation. Note that the code 0 indicates successful completions, positive codes indicate errors, and negative codes are warnings.

%_IORES makes a call to function FIOResult, which in turns references the variable IORslt. The variable IORslt is assigned values by the procedure %_SETIORSLT. This procedure is called by FPLib and appastext only.

%_GET - Read the next record in a file

Parameters: ST.L - File Address

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

%_PUT - Write the current record in a file

Parameters: ST.L - File Address

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

If %_PUT is called immediately after a file is opened with %_RESET, the PUT will write the second record of the file (since the %_RESET sets the current position to the first record and %_PUT advances the position before writing).

%_UPARR - Compute the address of F^

Parameters: ST.L - Address of file

Returns: ST.L - Address of F^

Registers used: All registers are preserved.

%_SEEK - Allows access to an arbitrary record in a file.

Parameters: ST.L - Address of file
ST.W - Record number to seek

Returns: ---

Registers used: D0, D1, D2, D3, A0, A1, A2

If the record number specified does not exists,

1) %_SEEK causes the next GET to access the last record in the last block of the file.

2) %_SEEK causes the next PUT to append the record to the end of the file.

PaslibCall Unit

(from the *Workshop User's Guide*)

The unit PASLIBCALL provides you with several system functions. In order to access the PASLIBCALL routines, you must use the units SYSCALL and PASLIBCALL:

```
USES
   {$U SysCall} SYSCALL,
   {$U PaslibCall} PASLIBCALL;
```

This gives you access to the routines listed below. These routines are contained in IOSPASLIB.OBJ, so programs using them require no additional inputs to the Linker.

function PAbortFlag : boolean

This function tells whether or not the ⌘-period key combination has been pressed. It enables programs to exit out of long operations. The flag is cleared when **PAbortFlag** is called. If you want your program to stop when you press ⌘-period, you must use this function in the program to detect that the key combination has been pressed. For example:

```
{This program fragment hangs in an infinite loop until ⌘-period is pressed}
aborted :=false
Repeat {Wait for ⌘-period.  You might want to do other things here}
   aborted :=PAbortFlag;
until aborted.
```

procedure ScreenCtr (contrfun : integer);

This procedure provides standard screen control functions, and enables programs to perform screen control without having to to use escape sequences. (Escape sequences are explained in Appendix C of the *Workshop User's Guide.*) The parameter specifies the screen control function. It is defined in the constants as follows, in the PASLIBCALL unit:

Function	Constant	Value Decimal	Hex
clear screen	CclearScreen	1	1
clear to the end of screen	CclearEScreen	2	2
clear to end of line	CclearELine	3	3
move cursor to home position	CgoHome	11	B
cursor left one position	CleftArrow	12	C
cursor right one position	CrightArrow	13	D
cursor up one line position	CupArrow	14	E
cursor down one line position	CdownArrow	15	F

Screen control example:

```
{This program fragment clears the screen, and positions the cursor on the
third line}

ScreenCtr (CgoHome);
ScreenCtr (CclearScreen);
ScreenCtr (CdownArrow);
ScreenCtr (CdownArrow);
```

procedure GetGPrefix (var prefix : pathname);

This procedure provides your program with the first level prefix setting in the File-Mgr in the Workshop.

procedure GetPrDevice (var PrDevice : e_name);

This procedure returns the corresponding default printer device name so that you can perform additional device control functions using DEVICE_CONTROL. (The *Operating System Reference Manual* explains the device control call.) The default printer device name is the one corresponding to the logical device **-PRINTER**. Note that the device name returned contains a leading '-'.

```
procedure PLINITHEAP  (var errnum,refnum:integer;
                       size,delta:longint
                       ldsn:integer;
                       swappable:boolean);
```

where:

errnum is the error number returned if the procedure has any problems making a data segment having a mem_size of size bytes. (See Appendix A of the *Workshop User's Guide* for an explanation of the error codes.)

size is the number of bytes in the heap.

refnum is the refnum of the heap.

delta is the amount you want the data segment to increase when the current space is used up. If you use a large heap, use a large number for delta.

ldsn is the logical data segment number used for the heap. The default is 5. For more information see the *Operating System Reference Manual for the Lisa.*

swappable is the boolean that determines if the system can swap the heap data segment out to disk if it needs to.

This procedure can be used when you have special needs; for example, when you want to specify your own **ldsn** or heap size. When you use **PLINITHEAP**, you must call it before calling other heap routines. For more information on the heap, see the *Workshop User's Guide*.

PPaslibC Unit: Privileged PASLIB Calls

The unit PPaslibC provides you with several useful low-level system functions. However, *they are not for everyone!* They are tricky, in some cases have global effects on the entire system, and should be used with caution.

In order to use these routines, you must use the units SYSCALL and PPaslibC:

```
USES
   {$U SysCall} SYSCALL,
   {$U PPaslibC} PASLIBCALL;
```

This gives you access to the routines listed below. These routines are contained in IOSPASLIB.OBJ, so programs using them require no additional inputs to the Linker.

procedure BlockIOinit;

Initializes all shared PASLIB data. Opens **inputfile** and **outputfile**, associating them with the filename **-CONSOLE**.

BlockIOinit must be called by every shell before performing any I/O; it will only be executed by the first shell that calls it.

It is called by the **system.shell** at boot time, once for the entire system.

procedure BlockIOdisinit;

PASLIB cleanup. **BlockIOdisinit** closes the console only for the first shell that called the **BlockIOinit** procedure.

procedure LockPaslib (var errnum: integer);

where:

errnum is the error number returned if the procedure has any problems. (See Appendix A of the *Workshop User's Guide* for an explanation of the error codes.)

Locks the PASLIB1 segment in memory so it won't be swapped out. Used by the filer for unmounting the boot device.

procedure LockPasIOlib (var errnum: integer);

where:

errnum is the error number returned if the procedure has any problems.

Locks the PASIOLIB segment in memory so it won't be swapped out. Used by the filer for unmounting the boot device.

procedure MoveConsole (var errnum: integer; applconsole: consoledest);

where:

errnum is the error number returned if the procedure has any problems.

applconsole tells where to move the console. (**Consoledest** is an enumerated type of: alscreen, mainscreen, xsorocA, xsorocB, folder, spare1, spare2, spare3.)

Moves the console to the main screen, an alternate screen, or an external terminal connected through RS232A or RS232B. The file names are:

Alternate Screen	**-ALTCONSOLE-X**
Main Screen	**-MAINCONSOLE-X**
External RS232A Terminal	**RS232A-X**
External RS232B Terminal	**RS232B-X**

procedure ExecReset (var errnum: integer; execfile: pathname; stopexec: boolean);

where:

errnum is the error number returned if the procedure has any problems.

execfile is the exec file name.

stopexec tells whether to open or stop the exec file. TRUE = stop; FALSE = open.

If **stopexec** is TRUE, **ExecReset** closes the input file and reopens it, associating it with the temporary exec file. It then generates two calls to the **FReadchar** function to read and save the temporary file's first character into the variable **gfirstchar**, and the next character into **greadahead**. **ExecReset** then sets the boolean **gexecflag** to TRUE.

If **stopexec** is FALSE, **ExecReset** calls the **Resetinput** procedure, which closes and reopens the input file, associating it with **-CONSOLE**. **ExecReset** then sets the boolean **gexecflag** to FALSE.

Opens or stops an exec file.

ExecReset is called once by the Exec Command Interpreter, to open and read from the exec temporary file and reopen the input file to the console.

function ExecFlag: boolean;

Tells whether an exec file is open. TRUE = open; FALSE = closed.

ExecFlag references the input file FIB boolean entry **FSOFTBUF**.

procedure OutputRedirect (var errnum: integer; outfile: pathname; stopoutput: boolean);

where:

errnum is the error number returned if the procedure has any problems.

outfile is the file name.

stopoutput tells whether to close the file or leave it open. TRUE = close; FALSE = leave open.

If **stopoutput** is TRUE, **OutputRedirect** calls the **Resetoutput** procedure, which closes and reopens the output file, associating it with **-CONSOLE**.

If **stopoutput** is FALSE, **OutputRedirect** closes the output file and reopens it, associating it with the filename **outfile**.

Redirects output to a file.

function OutputRFlag: boolean;

Tells whether output has been redirected to a file. TRUE = output file open (output redirected); FALSE = closed (output not redirected).

OutputRFlag references the output file FIB boolean entry **FSOFTBUF**.

procedure DSPaslibCall (var ProcParam: dsProcParam);

where:

```
dsProcParam = record
                case ProcCode : dsProcCode of
                  dsResProg : (RProcessId : longint); {must be called
                              before the process starts running.}
                  dsSoftPwbtn  : (SPButton : boolean); {result}
                  dsPrintDev   : (PrDevice : e_name);
                  dsSetGPrefix : (errnum : INTEGER;  {result}
                                  prefix : pathname);
                  dsEnbDisk    : (DiskEvent : boolean);
                  dsCiTranLisaCar : (toTranslate : boolean);
                       {to turn on or off translation for C. Itoh}
                end;

dsProcCode = (dsResProg, dsSoftPwbtn, dsPrintDev, dsSetGPrefix,
              dsEnbDisk, dsCiTranLisaCar);
```

dsResProg passes the process ID of a process that is going to be resident to PASLIB.

dsSoftPwbtn returns the soft power button setting. If the button is pressed, it returns TRUE; if not, it returns FALSE.

dsPrintdev passes the physical device name of the corresponding logical device **-PRINTER** to PASLIB.

dsSetGPrefix passes the global prefix volume name to PASLIB. If an error has occurred, it is returned in **errnum**.

dsEnbDisk tells PASLIB to enable (if **DiskEvent** is TRUE) or disable (if **DiskEvent** is FALSE) the automatic mounting and ejecting of a diskette.

dsCoTranLisaCar tells PASLIB to turn on (if **toTranslate** is TRUE) or off (if **toTranslate** is FALSE) the Lisa character translation for a C. Itoh printer for the calling process. The default setting is on.

DSPaslibCall is a new call in the PPaslibC unit that communicates to and from PASLIB about the run-time support for the system or the calling process. It has a variant-record parameter for indicating various functions. Note that most of these functions dictate system behavior; they are not safe for any process to call except the Lisa character translation function.

Floating-Point Libraries

Introduction

The Lisa provides arithmetic, elementary functions, and higher level mathematical algorithms in its intrinsic units **FPLib** and **MathLib**, which are contained in the file **IOSFPLIB**.

The contents of **FPLib** are described in the manuals for the Standard Apple Numeric Environment. The best currently available description of the Standard Apple Numeric Environment is the *Numerics Manual: A Guide to Using the Apple /// Pascal SANE and Elems Units* (part #030-0660-A), which will eventually be superseded by a manual applicable to all Apple products. **FPLib** provides the same functionality as the SANE and Elems units on the Apple] [and ///, including:

- Arithmetic for all floating-point and Comp types.
- Conversions between numerical types.
- Conversions between numerical types, ASCII strings, and intermediate forms.
- Control of rounding modes and numerical exception handling.
- Common elementary functions.

The **MathLib** guide (currently in draft form) describes the extra procedures available only on the Lisa. **MathLib** provides:

- Extra environments procedures.
- Extra elementary functions.
- Miscellaneous utility procedures.
- Sorting.
- Free format conversion to ASCII.
- Correctly rounded conversion between binary and decimal.
- Financial analysis.
- Zeros of functions.
- Linear algebra.

How to Use FPLib

FPLib is available as an intrinsic unit to Pascal programmers. If your only use of floating point is as Pascal REAL variables used within the limits of standard Pascal, then it is not necessary to include a USES statement for **FPLib**. But if you explicitly require any of the types or procedures defined in the **FPLib** interface, be sure to include a USES statement such as

USES FPLib;

after the program statement in a main program or after the interface statement in a unit. If you are also using other units, include **FPLib** in the list of units in your one

USES statement. **FPLib** may be listed before or after other Apple-supplied units that you are using.

When linking, be sure to include **IOSFPLIB** in your list of files to be linked along wiht **IOSPASLIB** and your own files, even if your only use of floating point is as Pascal REAL variables.

How to Use MathLib

MathLib is available as an intrinsic unit to Pascal programmers. When writing your Pascal source code, be sure to include a USES statement such as

USES FPLib, MathLib;

after the program statement in a main program or after the interface statement in a unit. If you are also using other units, include **FPLib** and **MathLib** in the list of units in your one USES statement. They may be listed before or after other Apple-supplied units that you are using, but **FPLib** muct appear in the list before **MathLib**.

When linking, be sure to include **IOSFPLIB** in your list of files to be linked along wiht **IOSPASLIB** and your own files.

TO: Development System Group, Mark Neubieser, Lee Nolan, Steve Flournoy, Wendell Henry

FROM: Fred Forsman

DATE: March 31, 1983

SUBJECT: Intrinsic unit providing standard functions -- the "StdUnit"

================================ INTRODUCTION ==================================

An intrinsic unit has been developed which provides a number of standard, generally-useful functions (particularly for the development system). The "StdUnit" unit has groups of functions dealing with (1) character and string manipulation, (2) file name manipulation, (3) prompting, (4) retrieval of error messages from disk files, (5) special WorkShop-oriented features, and (6) conversions.

The StdUnit is now available in the WorkShop A5 intrinsic library. A non-intrinsic Monitor-based version of the unit is available in my office.

Development system tools should be converted to use the unit where possible, especially in the area of prompting and OS error reporting since this will help make the WorkShop interface more consistent.

The rest of this memo explains the standard unit and its use. The material is organized into three sections:

(1) FUNCTIONAL AREAS -- a description of the areas of functionality
(2) SOME EXAMPLES -- some examples of how to use the functions
(3) THE INTERFACE -- the unit's interface

============================== FUNCTIONAL AREAS =================================

The five basic areas of functionality provided are:

(0) Initialization of unit

This is not really an area of functionality but it should not be overlooked. The unit needs to be initialized before it can be used. (Using the unit without initializing it will often result in an address or bus error.)

(1) String and character manipulation

The unit provides a standard string type "SUStr", a type for sets of characters, definitions for a number of standard characters (such as CR and BS), and procedures for case conversion on characters and strings, trimming blanks, and appending strings and characters.

NOTE: The names of EVERYTHING in StdUnit begin with the letters "SU". This may seem somewhat unnatural, but it practically insures that you will have no name conflicts when incorporating the standard unit into your code. It has the additional benefit of identifying where everything comes from.

(2) File name manipulation

A number of functions dealing with file names are provided -- determining if a pathname is a volume or device name only, adding extensions (such as ".text") to file names (the procedure is cognizant of our various conventions about when extensions should and should not be added), splitting a pathname into its three basic components (the device or volume component, the file name component, and the extension component), putting the components back together into a file name, and modifying a file name given optional defaults for missing volume, file or extension components.

NOTE: several of the procedures return overflow flags for identifying when a file name component has exceeded its character limit. You may chose to ignore the overflow condition, particularly you think it likely to occur only in perverse circumstances.

NOTE: you will notice that the string parameters to these procedures are often typed differently, sometimes SUStr's, or VAR SUStr's, or SUStrP's (ie, pointers to SUStr's). The apparent inconsistency of types is deliberate; the goal was to avoid awkward problems with Pascal string typing when using the procedures with strings which are not SUStr's (PathName's for example). It might have been best to use only SUStrP's, but the compiler does not allow @ of a string constant, so this would have been inappropriate when passing defaults such as '.text'. Please let me know if you can think of a way to make these procedures easier to use.

(3) Prompting

The unit provides a number of procedures which get characters, strings, file names, integers, yes/no responses, etc. from the console, providing for default values where appropriate. An attempt was made to do a cosmetically nice job of echoing responses, displaying defaults, etc. (I am open to further suggestions.)

Most of the prompting procedures return a PromptState which indicate such things as whether an escape (CLEAR) was typed, whether the default was taken, or whether there was a request for options with a '?'. The states returned are given for each procedure. The strings and prompt states returned have been designed to allow you to ignore the prompt states you are not interested in. For example, if you are not interested in treating '?' as a request for options, you may ignore the SUOptions state altogether and treat the '?' returned as a file name or whatever.

(4) Error Text Retrieval

The unit provides a mechanism which retrieves single-line error messages from specially formatted error files. Error messages can be looked up by number in one or more error files.

The original motivation for this was the aggravation of constantly looking up OS error numbers. A error file for OS errors is provided in the WorkShop release -- 'OSErrs.Err'. This makes it simple to return a real message when an OS error occurs, as is demonstrated in one of the examples in the following section. (Note that OS errors are also returned via Pascal's IORESULT.)

Whether the tool is useful for storing your program's error messages will depend primarily on whether you think your error messages are taking up too much space in memory. A program (described below) is available to make your own message files. One benefit of using this error mechanism is that you may add and modify messages without recompiling your program.

The "ErrTool" program is provided to construct your own compacted error message files. The tool produces an error file with an ordered directory of error numbers at the beginning of the file, along with pointers to the corresponding message text. The input to ErrTool consists of text lines of the form:
    <number><space><message>
The error numbers may be sparse, and the messages may be up to 255 chars long.

A call to retrieve a message will open the error file, search the directory for the error number, seek to the location of the message, and return the text. This may result in several file system accesses but the response seems reasonable (even with a large number of errors with a directory spanning several blocks as in the OS error message file).

A program may use the unit to access any number of different error files simultaneously. You may, for example, access different files for OS and your own error messages.

(5) WorkShop Support

Special WorkShop-oriented functions supported are: the ability to stop the execution of an EXEC file in progress, the ability to find out the name of the boot and current process volumes (SysVols), and a super-RESET which will try to open a file first on the prefix volume, then on the boot volume, and, if all else fails, on the current process volume.

(6) Conversions

Routines to convert from INTEGERs (and LONGINTs) to strings and from strings to INTEGERs (and LONGINTs) are provided.

IMPORTANT NOTE: The standard unit and its interface have been written so as to work on either the Monitor or the OS depending on the setting of a compilation flag "ForOS" which you should set before you use the unit (you will get a compile error if you don't). Note that the Monitor version of the interface provides a definition of "PathName" which would normally come from SysCall when on the OS.

```
============================== SOME EXAMPLES ===================================

{ EXAMPLE 1
  Assume we are going to prompt for an output file name (OutFName) and that we
  already have the input file name (InFName).  We will use SUSplitFN to split
  the input file name into its various components.  Then we will prompt for the
  output file name (with SUGetFN) using the volume and file name components of the
  input file  name as defaults but with a '.ERR' extension.  We then do a CASE on
  the prompt state (PState) returned by SUGetFN.  Our program will terminate if the
  file specification was an escape (CLEAR on the keyboard); say that no options
  are available if '?' is typed as an option request; prompt again if no file is
  specified, since we want to require an output file; and fall through if the
  default is accepted or some other file is specified.  Note that we only have to
  check for the prompt states we are interested in for special handling. }

9999:
      WRITE ('Name of Error Output File ');
      SUSplitFN (@InFName, @VolN, @FN, @Ext);
      SUGetFN (@OutFName, PState, VolN, FN, '.ERR');
      CASE PState OF
        SUEscape:  EXIT (ErrFileP);  { exit from program }
        SUOptions: BEGIN
                     WRITELN ('No options available.');
                     GOTO 9999;
                   END;
        SUNone:    GOTO 9999;
      END {CASE};


{ EXAMPLE 2
  Suppose we have just made a Pascal IO call and want to report an error (along
  with the OS message text) if we receive a non-zero IORESULT.  Note that we copy
  IORESULT into our IOStatus variable so that the subsequent WRITELN will not
  reset the value of IORESULT before we get a chance to use it.  (EMsg should be
  an SUStr.) }

      IF IORESULT <> 0 THEN
        BEGIN
          IOStatus := IORESULT;
          WRITELN ('Error openning input file.');
          SUErrText ('OsErrs.Err', IOStatus, @EMsg);
          WRITELN (EMsg);
        END;

================================= INTERFACE ====================================

{-------------------------------- SU: StdUnit ----------------------------------}
{ Copyright 1983, Apple Computer, Inc.                                          }
```

```
{                                                                             }
{ This unit provides a number of standard type definitions and a collection of }
{ procedures which perform a variety of common functions.  The areas covered are: }
{    (1) String and Character manipulation                                    }
{    (2) File Name Manipulation                                               }
{    (3) Prompting                                                            }
{    (4) Retrieval of messages from disk                                      }
{    (5) Development System Support                                           }
{    (6) Conversions                                                          }
{                                                                             }
{ Fred Forsman 3-28-83                                                        }
{-----------------------------------------------------------------------------}

  INTERFACE

{$IFC ForOS}
    USES
      {$U SysCall.obj    } SysCall,  { for definition of PathName, etc. }
      {$U PasLibCall.obj } PasLibCall,
      {$U PPasLibC.obj   } PPasLibC;
{$ENDC}

    CONST
      SUMaxStrLeng = 255;
      SUNullStr    = '';
      SUSpace      = ' ';
      SUOrdCR      = 13;
{$IFC ForOS}
      SUMaxPNLeng  = 66;    { max length of path name }
      SUMaxVNLeng  = 33;    { max length of volume name, includes leading '-' }
      SUMaxFNLeng  = 32;    { maximum length of file name }
      SUVolSuffix  = '-';   { suffix or end of device or volume name }
{$ELSEC}
      SUMaxPNLeng  = 39;    { max length of path name }
      SUMaxVNLeng  = 24;    { max length of volume name, includes trailing ':' }
      SUMaxFNLeng  = 15;    { maximum length of file name }
      SUVolSuffix  = ':';   { suffix or end of device or volume name }
{$ENDC}

    TYPE
      SUSetOfChar  = SET OF CHAR;
      SUStrP       = ^SUStr;
      SUStr        = STRING[255];
      SUVolName    = STRING[SUMaxVNLeng];
{$IFC NOT ForOS}
      PathName     = STRING [255];  { supply definition of PathName for Monitor }
{$ENDC}
      SUFile       = FILE;
      SUFileP      = ^SUFile;
      PromptState = (SUDefault,     { the default (if any) was chosen) }
                     SUEscape,      { the "Clear" key was pressed }
                     SUNone,        { nothing specified in response to prompt }
                     SUOptions,     { "?" was entered -- ie, an option query }
                     SUValid,       { valid reponse }
                     SUInvalid      { invalid reponse -- eg, non-number to SUGetInt }
                     );
      ErrTextRet  = (SUOk,          { successful }
                     SUBadEFOpen,   { could not open error file }
                     SUBadEFRead,   { error reading error file }
                     SUErrNNotFound { error number not found }
                     );
      ConvNState  = (SUValidN,      { valid number }
                     SUNoN,         { no number -- nothing specified }
                     SUBadN,        { invalid number }
                     SUNOverFlow    { overflow -- number too big }
                     );

    VAR
```

```
{$IFC ForOS}
      SUOsBootV : SUVolName;  { The volume the OS was booted from }
      SUMyProcV : SUVolName;  { The volume MyProcess was started from }
{$ENDC}
      SUBell, SUBs, SUCr, SUTab, SUEsc, SUDle, SUNul : CHAR; { predefined ch vars }
      SUNullS   : SUStr;                                     { predefined str var }

{============================== INIT AND DONE ================================}

    PROCEDURE SUInit;
      { Should be called before using rest of unit.  On the OS this opens
        "-KeyBoard".  It also initializes the standard character variables.
        Note that SUInit sets SUOsBootV and SUMyProcV to null strings, and
        that SUInitSysVols should be called to set them to the correct values. }

    PROCEDURE SUDone;
      { Can be called when done using unit (although this is not strictly
        necessary).  On the OS this closes "-KeyBoard". }

{============================ STRINGS AND CHARS ==============================}

    FUNCTION SUUpCh (Ch : CHAR) : CHAR;
      { SUUpCh returns the ch that was passed, uppercased if it was lower case. }

    FUNCTION SULowCh (Ch : CHAR) : CHAR;
      { SULowCh returns the ch that was passed, lowercased if it was upper case. }

    PROCEDURE SUUpStr (S: SUStrP);
      { SULowStr uppercases the string that is passed. }

    PROCEDURE SULowStr (S: SUStrP);
      { SULowStr lowercases the string that is passed. }

    PROCEDURE SUTrimBlanks (S: SUStrP);
      { SUTrimBlanks removes leading and trailing blanks and tabs in the passed
        string. }

    PROCEDURE SUAddCh (S: SUStrP; Ch : CHAR; MaxStrLeng : INTEGER;
                       VAR OverFlow : BOOLEAN);
      { SUAddCh appends the passed ch to the end of the passed string.
        OverFlow is set to TRUE if adding the ch will cause the string to be
        longer than MaxStrLeng. }

    PROCEDURE SUConcat (S1: SUStrP; S2: SUStrP);
      { SUConcat appends the second passed str to the end of the first passed
        string.  It is assumed that the target string is of sufficient size to
        accomodate the new value. }

    PROCEDURE SUAddStr (S1: SUStrP; S2: SUStrP; MaxStrLeng : INTEGER;
                        VAR OverFlow : BOOLEAN);
      { SUAddStr appends the second passed str to the end of the first passed
        string.  OverFlow is set to TRUE if adding the second string will cause
        the resulting string to be longer than MaxStrLeng. }

    PROCEDURE SUSetStr (Dest: SUStrP; Src: SUStrP);
      { SUSetStr sets the target string (Dest) to the given value (Src) by
        copying the value onto the target.  It is assumed that the target string
        is of sufficient size to accomodate the new value. }

    PROCEDURE SUCopyStr (Dest: SUStrP; Src: SUStrP; Start, Count: INTEGER);
      { SUCopyStr sets the destination string (Dest) to the specified substring of
        the source string (Src) by copying the appropriate part of the source to
        the destination.  It is assumed that the destination string is of
        sufficient size to accomodate the new value, and that the Start and Count
        values are reasonable. }

{================================ FILE NAMES =================================}
```

```
FUNCTION SUIsVolName (FN: SUStrP): BOOLEAN;
  { SUIsVolName returns a boolean indicating whether the passed file name, FN,
    is a volume or device name (i.e., not a full file name) }

PROCEDURE SUAddExtension (FN: SUStrP; DefExt: SUStr;
                          MaxStrLeng: INTEGER; VAR OverFlow: BOOLEAN);
  { SUAddExtension will add the default extension, DefExt, to the end of the
    file name, S, if the extension is not already present.  If the file name
    ends with a dot, the dot will be removed and no extension will be added.
    If the pathname is a device or volume name only no extension will be
    added.  OverFlow is set true if adding the extension will overflow the
    string (determined using MaxStrLeng). }

PROCEDURE SUSplitFN (PathN: SUStrP; VolN: SUStrP; FN: SUStrP; Ext: SUStrP);
  { SUSplitFN splits a PathName into its volume (device), file name, and file
    name extension components. }

PROCEDURE SUMakeFN (PathN: SUStrP; VolN: SUStrP; FN: SUStrP; Ext: SUStr;
                    VAR OverFlow: BOOLEAN);
  { SUMakeFN constructs a PathName from its volume (device), file name, and
    file name extension components.  The OS VolN's are assumed to have a
    leading "-", while monitor VolN's are assumed to have a trailing ":".
    OverFlow is set if any of the file name components are too long. This
    procedure will not create a file name over SUMaxPNLeng chars long.}

PROCEDURE SUChkFN (FN: SUStrP; VAR PState: PromptState; DefVol: SUStr;
                   DefFN: SUStr; DefExt: SUStr);
  { SUChkFN checks a file name specification, putting result type in PState.
    If no file name is given, then DefFN is used.  If FN does not have DefExt
    in it, then the extension is appended.  If no volume is specifed then
    the DefVol is used.  PState is set appropriately:
      PState = SUOptions  if '?' is hit to ask for options
      PState = SUDefault  if nothing specified when a default is present
      PState = SUNone     if default overriden with '\' or if CR with no default
      PState = SUInvalid  if one or more of the file name components overflowed
      PState = SUValid    otherwise }

{================================ PROMPTING ==================================}

PROCEDURE SUGetCh (VAR Ch: CHAR);
  { SUGetCh reads a character from the console without echoing it and }
  { without interpreting <cr> as <sp>, as Read (Ch) does. }

PROCEDURE SUGetLine (S: SUStrP; VAR PState: PromptState);
  { SUGetLine reads a line from the console a character at a time, performing
    its own line editing.  PState is set appropriately:
      PState = SUEscape  if <clear> was hit.
      PState = SUValid   otherwise. }

PROCEDURE SUGetStr (S: SUStrP; VAR PState: PromptState; DefVal: SUStr);
  { SUGetStr reads a string from the console; it is like SUGetLine with the
    addition of defaults.  PState is set appropriately:
      PState = SUDefault if <cr> only was hit; S is set to DefVal.
      PState = SUEscape  if <clear> was the first character hit.
      PState = SUValid   otherwise. }

PROCEDURE SUGetFN (FN: SUStrP; VAR PState: PromptState; DefVol: SUStr;
                   DefFN: SUStr; DefExt: SUStr);
  { SUGetFN reads a file name from the console, with result type in PState.
    SUGetFN will print out any defaults in brackets (such as [FOO] [.TEXT])
    before prompting for for the file name.  If no file name is given, then
    DefFN is used.  If FN does not have DefExt in it, then the extension is
    appended.  If no volume is specifed then the DefVol is used.  If only a
    volume name is specified then no default file name or extension will be
    added.  PState is set appropriately:
      PState = SUEscape   if <clear> hit
      PState = SUOptions  if '?' is hit to ask for options
      PState = SUDefault  if nothing specified when a default is present
```

```
        PState = SUNone     if default overriden with '\' or if CR with no default
        PState = SUInvalid  if one or more of the file name components overflowed
        PState = SUValid    otherwise }

    PROCEDURE SUGetInt (VAR I: INTEGER; VAR PState: PromptState; DefVal: INTEGER);
      { SUGetInt reads an INTEGER from the console, with PState set as in
        SUGetStr, except that PState = SUInvalid when a non-numeric is input. }

    PROCEDURE SUWaitEscOrSp (VAR PState: PromptState);
      { SUWaitEscOrSp prints a message 'Type <space> to continue, <clear> to exit.'
        & waits for the user to hit a <sp> or <clear>, setting PState appropriately:
          PState = SUEscape if <clear> was hit
          PState = SUValid  if <sp> was hit }

    PROCEDURE SUWaitSp;
      { SUWaitSp prints a message ('Type <space> to continue.') and waits for the
        user to hit a <sp>. }

    PROCEDURE SUGetChInSet (VAR Ch: CHAR; Chars: SUSetOfChar);
      { SUGetChInSet reads characters from the console (without echoing) until
        a character from the given set is typed.  The accepted character is echoed
        and an end-of-line is written. }

    FUNCTION SUGetYesNo : BOOLEAN;
      { SUGetYesNo prints the message "(Y or N)" and reads characters from the
        console (without echoing) until a 'y', 'Y', 'n', or 'N' is typed.  If a
        'y' is typed "Yes" will be printed followed by an end-of-line; if 'n' is
        typed "No" will be printed.  The appropriate boolean value is returned. }

    FUNCTION SUGetBool (Default: BOOLEAN): BOOLEAN;
      { SUGetBool prints the message "(Y or N)  [<default>]" and reads characters
        from the  console (without echoing) until a 'y', 'Y', 'n', 'N', space or
        return is typed.  If a 'y' is typed "Yes" will be printed in the place
        of the default.  If 'n' is typed "No" will be printed.  If a space or
        return is typed the default is used.  The appropriate boolean value is
        returned. }

{============================ ERROR TEXT RETRIEVAL ==============================}

    PROCEDURE SUGetErrText (ErrFN: SUStr; ErrN: INTEGER; ErrMsg: SUStrP;
                            VAR ErrRet: ErrTextRet);
      { SUGetErrText retrieves error message text, given an error number and
        and error file to look the error up in.  The error file should have
        been generated by the error file processor.  SUGetErrText use SUSysReset
        to open the error file. }

    PROCEDURE SUErrText (ErrFN: SUStr; ErrN: INTEGER; ErrMsg: SUStrP);
      { SUErrText retrieves error message text, just as does SUGetErrText;
        however, if the text is not obtainable due to a non-SUOk ErrRet value
        from SUErrText, SUErrText will return the string
        "Error message text not available." }

{============================== DEV. SYS. SUPPORT ===============================}

    PROCEDURE SUStopExec (VAR ErrNum: INTEGER);
      { Kills and exec file on the OS, returns any error conditions in errnum }

{$IFC ForOS}
    PROCEDURE SUInitSysVols;
      { Initializes "SUMyProcV" and "SUOsBootV", the name of the volume on which
        my process was created and the name of the volume which the OS was booted
        off of.  A message may be printed if there is trouble getting this
        information from the OS.  This can be called more than once; it will only
        make the OS calls if SUMyProcV and SUOsBootV are both null strings (as
        they will be after a call to SUInit. }
{$ENDC}

    PROCEDURE SUSysReset (F : SUFileP; FN : SUStr; VAR IOStatus : INTEGER);
```

```
    { SUSysReset is for opening system files, and will try the prefix, boot,
      and current process volumes (in that order) when trying to access a file.
      SUSysReset assumes that the file name FN does not have a volume name.
      SUSysReset may sometimes have to call SUInitSysVols. }

{================================= CONVERSIONS =================================}

  PROCEDURE SUIntToStr (N : INTEGER; S : SUStrP);
    { SUIntToStr converts an integer into its string form; The string which S
      points to should be of length >= 6 (5 digits + sign). }

  PROCEDURE SULIntToStr (N : LONGINT; S : SUStrP);
    { SULIntToStr converts an longint into its string form; The string which S
      points to should be of length >= 11 (10 digits + sign). }

  PROCEDURE SUStrToInt (NS : SUStrP; VAR N : INTEGER; VAR CState : ConvNState);
    { SUStrToInt converts a string to an INTEGER. Leading and trailing blanks
      and tabs are permitted. A leading sign ['-', '+'] is permitted. The
      CState variable (conversion state) will be set to indicate if the number
      was valid, if no number was present, if an invalid number was specified,
      or if the number overflowed. }

  PROCEDURE SUStrToLInt (NS : SUStrP; VAR N : LONGINT; VAR CState : ConvNState);
    { SUStrToLInt converts a string to a LONGINT. It behaves just like
      SUStrToInt otherwise. }
```

Rich Page
Apple Computer, Inc.
May 4, 1983

Execution Environment of the Lisa Pascal Compiler

Registers:

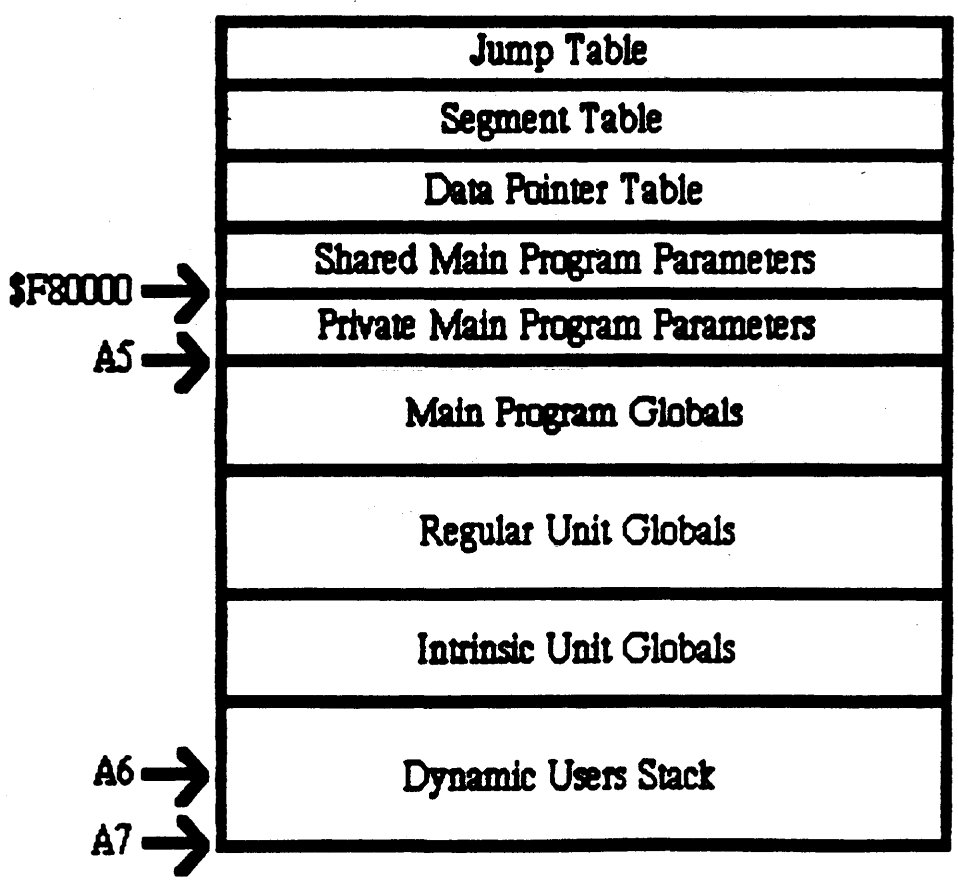
D0-D2/A0-A1	User temporaries
D0-D3/A0-A2	Compiler temporaries
D4-D7/A3-A4	Compiler uses for locals & pointers
A5	Pointer to global frame
A6	Pointer to local frame
A7	Pointer to top of stack

Global Frame:

The global frame consists of two segments:

1) The Jump Table Segment
2) The Stack Segment (first of N segments)

The global frame is layed out as follows:



The Jump Table is a an array of 6 byte JMPs used to transfer control between segments of the program and the regular units used by the program. This is built by the Linker from Entry points and Externals reference lists.

The Segment Table is a structure which defines each of the segments of the program and the regular units. This is used by the Loader to swap in segments. For each of the segments, the Segment Table provides a file address, size of code (packed & unpacked sizes) and the logical address (ie. segment number).

The Data Pointer Table is an array of 4 byte pointers which is used to reference global data for intrinsic units. This structure is built by the Loader and referenced by compiled code.

The Shared Main Program Parameters is an area reserved for use by the Loader to store information about the main program. Currently this area is $100 bytes.

The Private Main Program Parameters is an area initialized by the loader and referenced by compiled code. This area contains pointers to INPUT and OUTPUT file buffers and other information such as the size of the regular unit globals. Currently this area is $100 bytes.

The Main Program Globals is the global data allocated by the compiler for the program.

The Regular Unit Globals is the combination of all global data required by the regular units used by the program.

The Intrinsic Unit Globals is the private global data which is required by the intrinsic units used by the program.

The Users Dynamic Stack is that area which is used by the program for local frames, temporary data and procedure linkages (both pascal and assembly language).

Initially the Loader allocates enough space to cover these areas and the user min stack requirements. The system also enforces a upper limit (ie. max stack).

Local Frame:

The local frame consists of the following:

1) Function result and parameters
2) Static and dynamic links
3) Locals and compiler temporaries
4) Dynamic stack area

The local frame is layed out as follows:

	Function Result *
	Parameters **
	Static Link ***
	Return Address
	Dynamic Link
A6 →	Locals
	Compiler Temps
	Dynamic Stack Area

* Two or four byte function result, present only for functions.
** N bytes depending on the parameter list.
*** Present only for non level 1 procedures and parameters.

The local frame is allocated by the compiler and allows the compiled code to reference locals, paramters, static links, ...

The dynamic link (ie. OldA6) is pushed by the LINK A6 instruction which allocates space for locals and compiler temps.

The static link is pushed by the caller as part of the parameter list. The static link is a copy the parents A6 (ie. local frame).

Compiler temporaries are used to implement constructs such as non local gotos and expressions computed by the compiler which happen to not be in registers. These expressions may include for loop limits or with expressions.

Parametric procedures and functions appear as follows:

Address of proc/func body
Zero or static link

Note zero is used for level 1 procedures.

Automatic Stack Expansion:

The compiler communicates the space requirements for each procedure by preceding each LINK A6,#-size with one of the following sequences:

```
        TST.W     e(A7)
or
        MOVE.L    A7,A0
        SUB.L     #size,A0
        TST.W     (A0)
```

The offset used in the first case or the size in the second reflect the sum of the procedures static and dynamic requirements. This sum is inflated by at least $100 bytes to allow assembly language procedures to use a small amount of stack space at low cost (ie. they need not check). Note the code for automatic stack expansion can be controlled with a compile option.

JSRs, JMPs, LEAs and PEAs:

These instructions are used to transfer control and obtain the address of a procedure or function. These instructions exist in three forms all of which occupy 4 bytes each:

1) Within a segment: PC relative
2) References to regular segments: Offsets from A5
3) References to intrinsic segments: IU Trap instructions

The first form is simply a reference to a procedure from within the same segment which uses the PC relative addressing mode.

The second form is a reference to a procedure which is not in the same segment but is contained in a segment of the program or a regular unit. This is implemented by using an offset from A5 to reference the procedure through the Jump Table.

The third form is a reference to a procedure which is contained in an intrinsic segment (ie. in an intrinsic unit). This form is implemented by using Line 1010 trap mechanism to compress the opcode and 24 bit logical address into a 4 byte instruction.

In each of the above cases the compiler emits references the desired procedure or function and the linker constrcuts the appropiate addressing mode for JSRs, JMPs, LEAs and PEAs.

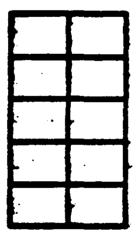
Structure of Code for a Pascal Procedure or Function:

The code emitted by the compiler contains three constructs which can be controlled via compile time options. These are as follows:

1) Automatic stack expansion.
2) Range checking for values, indexes and strings.
3) Debuggung info (ie. the procedure name).

The code for a typical procedure will look as follows:

```
TST.W    e(A7)         Tests for sufficient stack space
LINK     A6,#-size     Allocates space for locals

body of the procedure or function

UNLK     A6            Restores previous local frame
RTS                    Exit sequence
```



Eight byte procedure name and two byte data size. This is the optional debugging information.

constant data area for strings & sets

The exit sequence emitted by the compiler is dependent on the number of bytes of parameters. If there are no parameters then the RTS is used as shown above. The compiler emits one of the following sequences when parameters must be deleted:

Case #1: 2, 6 or 8 bytes of paramters

```
MOVEL    (A7)+,A0      2
ADDQ.W   #size,A7      2
JMP      (A0)          2
                       6 bytes total
```

Case #2: 4 bytes of parameters

```
MOVEL    (A7)+,(A7)    2
RTS                    2
                       4 bytes total
```

Case #3: more than 8 bytes of parameters

```
MOVEL    (A7)+,A0      2
ADD.W    #size,A7      4
JMP      (A0)          2
                       8 bytes total
```

Segmentation & Large constants passed by value:

Since the 68000 is not restartable, (ie. use a 68010 instead) the data (ie. stack and heaps) for a given program must be present while the program is executing. Since code segments must be swapped into memory as needed and set and string constants are stored with the code, large constants passed by value pose a problem. Currently, we solve this problem by having the compiler use the instruction TST.B (Ai) to check to see if the the actual value parameter is in memory. If the TST.B (Ai) causes a fault then the system loads the segment containing the address in Ai.

When copying strings the compiler emits code which depends only on the size of the destination. This may cause the code to read beyond the end of a segment. The system allows for this by mapping code segments to cover size + 256 bytes. The heap segments also have an additional 256 bytes.

Intrinsic Units

NOTE: The information in this document will be in the Units section of the Pascal Manual in the spring release.

Intrinsic units provide a mechanism for Pascal programs to share common code. A single copy of the code is kept on disk, and when loaded into memory this code can be executed by any program that declares the intrinsic unit (via a uses clause, just as for regular units) and has been linked against the library file. In addition, a *shared* intrinsic unit provides for the sharing of common data (i.e., one copy of the data on the system).

The code of the entire unit, or of blocks within the unit, must be placed in one or more named segments. Segmentation is controlled by the **$S** compiler command (described in the *Pascal Reference Manual*), the **ChangeSeg** utility, and the **+M** linker option (both described in the *Workshop User's Guide*). Code from an intrinsic unit cannot be placed in the same segment with code from a program or a regular unit.

Writing Intrinsic Units

An intrinsic unit has the same syntax as a regular unit, except that it has an intrinsic clause in the heading.

> NOTE: For syntactic compatibility with UCSD Pascal, the keywords **code** and **data** may appear in the unit heading of an intrinsic unit, together with integer constants. These keywords and constants are accepted but are ignored.

If the keyword **shared** appears in the intrinsic clause, the system will contain only a single data area for the unit; the data is shared among all programs that use this unit. If **shared** does not appear in the intrinsic clause, each program that uses the unit has its own data area for the unit.

If an intrinsic unit contains a uses clause, it can only use other intrinsic units; an intrinsic unit cannot use a regular unit.

Each unit used by a program (or by another unit) must be compiled, and its object file must be accessible to the compiler, before the program (or unit) can be compiled.

A single copy of the code of an intrinsic unit is available to all programs in the system; therefore, intrinsic units must be coordinated as part of system generation and system maintenance activities. Specifically, all intrinsic units that have code in the same run-time code segment file must be linked together into an intrinsic segment file, and the intrinsic segment file must be referenced in the system intrinsics library, INTRINSIC.LIB.

Building Library Files

To create intrinsic units and link them into a library file, you must perform the following steps in order, as shown in Figure 1:

STEP 1A Compile and Generate the intrinsic units.
STEP 1B Define the intrinsic units, code segments, and file names, using the IUManager. (Steps 1A and 1B can be done in either order.)
STEP 2 Link the intrinsic libraries.
STEP 3 Install the library files, using the IUManager.
STEP 4 Develop the main programs (not shown in detail).
STEP 5 Run main programs which use the library files. (The system must be rebooted before this step.)

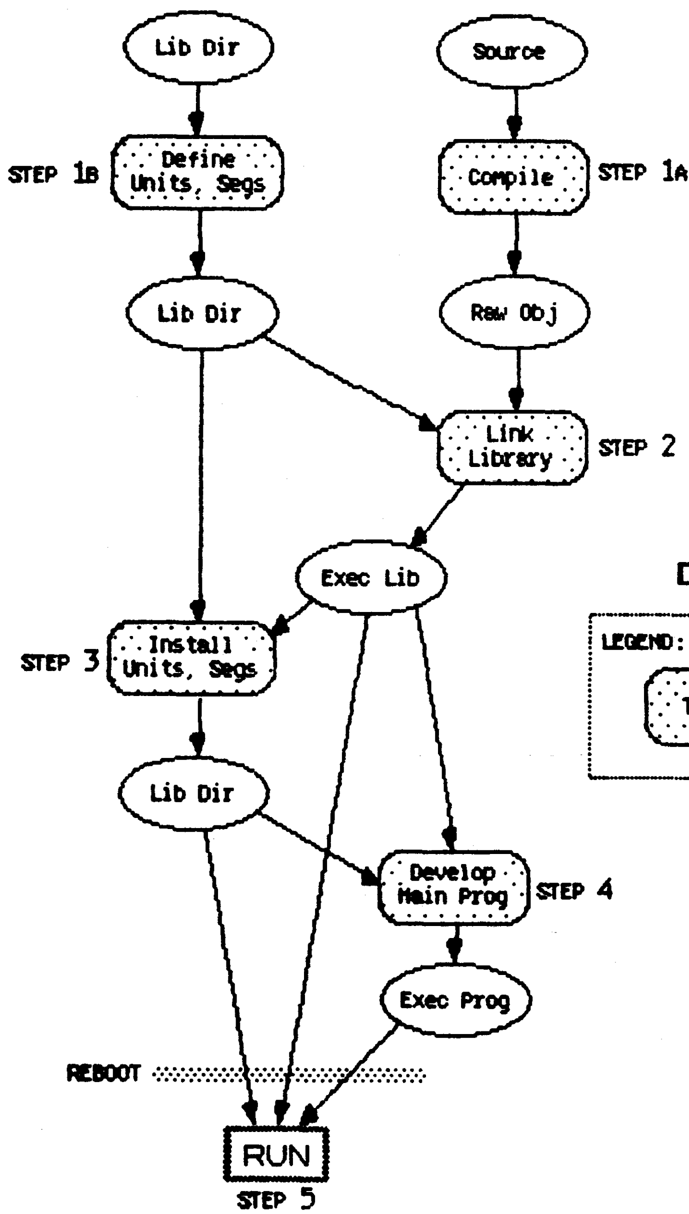


Figure 1
Developing Intrinsic Libraries

LEGEND: Tool File info flow

The IUManager

(For versions 1.x and 2.x software)

The IUManager program is used to manage the directory of library files. You can add, delete, or change intrinsic units, segments, and files in the directory. To use the IUManager, you should be familiar with the way that units and segments are handled in Pascal on the Lisa. (Information on intrinsic units is in the Intrinsic Unit ERS by Ken Friedenbach from September 16, 1981.) This document describes the version of the IUManager in software prior to the "spring release".

The IUManager has three modes, which do the following:

Units: Add, delete, or change intrinsic units. An intrinsic unit is a unit of Pascal code that can be accessed by different processes. There are two kind of intrinsic units--regular and shared. A regular intrinsic unit has a private global data area associated with it; shared intrinsic units share data as well as code.

Segments: Add, delete, or change segments. Units can be broken up into segments, so that interdependant parts of different units will be swapped in and out of memory at the same time. You can segment your code with either the $S Compiler option or the ChangeSeg utility.

Files: Add, delete, or change library files. Units and segments are arranged in library files.

When you run the IUManager, you are asked the input and output names of the library directory that you want to edit. The default name for both is INTRINSIC.LIB, the directory that the system looks for at boot time. (Don't play with the INTRINSIC.LIB unless you know what you're doing, or your system may not boot!)

When you first enter the IUManager, you're in the segments mode. The IUManager has only one command line, so if you don't know which mode you're in, either L(ist the current table or type S(egs, U(nits, or F(iles to get to the mode you want. The commands available in the IUManager are:

Q(uit Quit the IUManager and rewrite the directory.

S(egs — Select the segments mode and list the segment table. Entries in the segment table have the following information:

SEGMENT	The segment name.
NUM	The segment number (17-128).
F-NUM	The number of the file that the segment is in.
F-LOC	The byte location of the segment in the file.
PACKED/ UNPACKED	The number of packed or unpacked bytes in the segment.
FILE-NAME	The name of the file that the segment is in.

U(nits — Select the units mode and list the unit table. Entries in the unit table have the following information:

UNIT	The unit name.
NUM	The unit number (1-256).
F-NUM	The number of the file that the unit is in.
TYPE	The type of unit: Intrinsic or Shared Intrinsic.
DATA-SIZE	The number of bytes of global data (Shared Intrinsic units only).
FILE-NAME	The name of the file that the unit is in.

F(iles — Select the files mode and list the file table. Entries in the file table have the following information:

NUM	The file number (1-64).
FILE	The file name.

I(ns — Install a library in the directory. This stores the segment and unit tables from the linked object file. The Install command puts you in the files mode if you're not in it already, lists the file table, and prompts you for the file number to install.

L(ist — List the entries in the currently selected table. Use -S to stop the output for tables of more than 32 entries.

P(rt — Print all three tables. This command doesn't work. If you accept the default [PRINTER:], the tables are not printed, but are sent to a file named PRINTER:. To print the tables, send them to a .TEXT file (or change the PRINTER: file to a .TEXT file), and print them from the Editor.

R(em — Remove an entry from the currently selected table. You are prompted for the segment, unit, or file number. If you try to remove a file that is used by the segment table, you will get a warning, and the file will not be removed.

C(hng — Change an entry in the currently selected table. You will be asked for the segment, unit, or file number, and prompted for changes in each field. If you enter an unused number, the Change command works just like the New command. If, in changing a unit or segment, you specify a file name that has not been used, a new file will be created with the next available file number.

N(ew — Create a new entry in the currently selected table. You will be asked for the segment, unit, or file number, and prompted for each field. If you enter a number already associated with an entry, the New command works just like the Change command. The default entry number is the first unused number in the table. Valid ranges for entry numbers are:

Segments	17 - 128
Units	1 - 256
Files	1 - 64

NOTE: Segment numbers 1-16 are used by the OS, but the IUManager doesn't know this, and prompts you for them. DO NOT USE THEM, or unspecified evil things will happen.

If you add a unit or segment and specify a file name that has not been used, a new file will be created with the next available file number.

V(erify — Verify that the information in the linked object file is consistent with the directory.

The IUManager

(For Apple pre-release version 3.x software)

The IUManager utility is used to manage the directory of library files. You can add, delete, or change intrinsic units, segments, and files in the directory. To use the IUManager, you should be familiar with the way that units and segments are handled in Pascal on the Lisa. (Information on intrinsic units is in the Intrinsic Unit ERS by Ken Friedenbach from September 16, 1981.) This document describes the internal pre-release version of the IUManager in the "spring release", which is liable to change without notice (though not significantly).

The IUManager has three modes, which do the following:

UNITS: Add, delete, or change intrinsic units. An intrinsic unit is a unit of Pascal code that can be accessed by different processes. There are two kind of intrinsic units--regular and shared. A regular intrinsic unit has a private global data area associated with it; shared intrinsic units share data as well as code.

SEGMENTS: Add, delete, or change segments. Units can be broken up into segments, so that interdependant parts of different units will be swapped in and out of memory at the same time. You can segment your code with either the $S Compiler option or the ChangeSeg utility.

FILES: Add, delete, or change library files. Units and segments are arranged in library files.

When you run the IUManager, you are asked the input and output names of the library directory that you want to edit. The default name for both is INTRINSIC.LIB, the directory that the system looks for at boot time. (Don't play with the INTRINSIC.LIB unless you know what you're doing, or your system may not boot!)

When you first enter the IUManager, you're in the FILES mode. To switch between modes, the following commands are available:

S(egments	Enter the SEGMENTS mode and display the segment table. Entries in the segment table have the following information:	
	SegName	The segment name.
	Seg#	The segment number.
	File#	The number of the file that the segment is in.
	FileLoc	The byte location of the segment in the file.
	Packed/ UnPacked	The number of packed or unpacked bytes in the segment.
	FileName	The name of the file that the segment is in.
U(nits	Enter the UNITS mode and display the unit table. Entries in the unit table have the following information:	
	UnitName	The unit name.
	Unit#	The unit number.
	File#	The number of the file that the unit is in.
	Type	The type of unit: Intrinsic or Shared Intrinsic.
	DataSize	The number of bytes of global data (Shared Intrinsic units only).
F(iles	Enter the FILES mode and display the file table. Entries in the file table have the following information:	
	File	The file number.
	FileName	The file name.

Other than the S(egments, U(nits, and F(iles commands, the commands available in all three modes are the same:

C(hange — Change an entry in the currently selected table. You will be asked for the file, unit, or segment number, and prompted for changes in each field. If you enter an unused number, the Change command works just like the Add command.

A(dd — Add a new entry in the currently selected table. You will be asked for the file, unit, or segment number, and prompted for each field. If you enter a number already associated with an entry, the Add command works just like the Change command. The default entry number is the first unused number in the table. If you add a unit or segment and specify a file name that has not been used, a new file will be created with the next available file number.

D(elete — Delete an entry from the currently selected table. You are prompted for the file, unit, or segment name or number. If you try to delete a file that is used by the segment table or unit table, you will get a warning, and the file will not be removed. If you try to delete a segment that is used by the system table as a Public Interface segment, the segment will not be removed.

L(ist — List the entries in the currently selected table.

Q(uit — Quit the IUManager and rewrite the directory.

? — Typing ? from the main command line displays the alternate command line, with the following commands:

I(nstall — Install a library in the directory. This stores the segment and unit tables from the linked object file. The Install command puts you in the FILES mode if you're not already, displays the file table, and prompts you for the file name or number to install.

V(erify — Verify that the information in the linked object file is consistent with the directory. You are prompted for the name of the file to verify.

P(rint — Print all three tables. (You can send the tables to a .TEXT file instead of -PRINTER if you want to look at them in the Editor.)

? — Typing ? from the alternate command line returns you to the main command line.

CONFIDENTIAL

Subject: Lisa Object File Formats

Date: August 14, 1982 (O.S. 5.2, Monitor 10)

From: Ken Friedenbach

CONTENTS

1.0 Introduction

This document provides a detailed reference manual for the object file formats and system conventions which define the software run-time environment for Lisa Applications. This information is of use to developers of compilers which emit object code to be linked with the IULinker. Object code which is in these formats can be executed under the Monitor or the O.S. Loaders or be debugged with LisaBug. Fred Forsman is currently working on a Symbolic Debugger which will assume these formats. Some of this information will be of use to third-party software developers who develop libraries of Intrinsic Units to support specialized applications. This information may be of use to programmers who develop and debug programs at the machine or assembly language level.

This document describes a set of Intrinsic Units used by programs in the Pascal Development System which create and access object files. These units are useful in building utility programs which can be maintained across changes in object file formats. The units are distributed in the library file named ObjIOLib.OBJ. The ObjIOLib units are used by the Pascal Compiler, the Code Generator, the Assembler, the Monitor Loader, the IULinker, the IUManager, and a variety of utility programs including DumpObj, ChangeSeg, GXRef, SegMap, CodeSize, PackSeg, and ReUse. The units will be used by the Symbolic Debugger. Information on the functions and use of the above programs is contained in the Pascal Development System Manual.

Developers of Code Generators are strongly urged to use the ObjIOLib units for writing object files and developing object file utilities. This will reduce maintainence difficulties caused by object file format changes.

This document describes the object files in their present form (Monitor Release 10, O.S. Release 5.2). Except for additions in the area of Symbolic Debugging, this form should be the formats for First Release of the Lisa Office System. In some places in the document, future changes or extensions are mentioned. This information is tentative and is primarily intended to aid in long range planning for maintenance.

1.1 Related Documents

The reader is assumed to be familiar with the following documents:

PASCAL DEVELOPMENT SYSTEM MANUAL, Bill Schottstaedt, February 16, 1982. Sections of relevance are: The Linker, Segmentation and Intrinsic Unit Management, and Object File Debugging.

PASCAL DEVELOPMENT SYSTEM INTERNAL DOCUMENTATION, Bill Schottstaedt, February 16, 1982. This document is an expansion of the sections: Linker File Layout and Jump Table Formats.

LISA PASCAL: LANGUAGE SPECIFICATION, Rich Page and David Casseres, February 19, 1982. Background material is contained in Section 14: UNITS.

LISA HARDWARE REFERENCE MANUAL. Especially the sections on Memory Mapping and address translation.

LISA OPERATING SYSTEM REFERENCE MANUAL. Especially the description of the loader (task initialization) and the flushing of INTRINSIC.LIB.

1.2 Overview of the Lisa Hardware/Software System

The Lisa Hardware supports the mapping of a 16 M-byte logical address space into a smaller physical address space at run-time. The 16 M-byte logical address space is divided into 128 (logical) segments of 128 K-bytes each.

The IULinker supports Intrinsic Units (shared code) by linking main programs and intrinsic units into absolute locations in the 16 M-byte logical address space. The system Loaders support the execution of programs which use Intrinsic units by swapping code into memory, setting up a Memory Management Unit (MMU) to translate logical adrreses into physical addresses, and handling the sharing of code between different programs (processes).

Uniform addressability of code is achieved by assigning an MMU number (128 K-bytes of logical address space) to each Intrinsic Unit segment. Code segments for a Main Program are assigned MMU numbers which are not among those assigned to Units used by the program.

Uniform addressability of data areas for Intrinsic Segments is achieved via pointers which are at a fixed location relative to the Global Frame pointer (register A5). This allows a "compact" allocation of global variables for Intrinsic Units without "holes" for Units which are not used.

Unlike UCSD Pascal the assignment of numbers to Segments and Units is done at Link time, not at compile time. Only Symbolic names are assigned at Compile Time. Also, the control of Segmentation is much more flexible than in UCSD Pascal. Procedures from different Units can be combined into the same segment.

Short Jumps (4 bytes rather than 6 bytes) to Intrinsic Unit Segments are achieved via emulated instructions. These instructions are editted by the IULinker. They make use of the "Axxx" class of emulation instructions supported by the hardware. See the section on the Intrinsic Unit Trap Handler in the PASCAL DEVELOPMENT SYSTEM INTERNAL DOCUMENTATION for more details. Current instructions emulated include:

IUJSR -- JSR to an IU Segment procedure or function.
IUJMP -- JMP to an IU Segment procedure or function.
IULEA -- LEA of an IU Segment procedure or function (except into A7).
IUPEA -- PEA of an IU Segment procedure or function.

The major advantages of this architecture are the following:

One copy of code (on the disk and in memory) can be part of several different programs.

Code can be swapped into memory in a state that is "ready to execute". No patching or load-time linking is needed.

Since code segments are "read only" code never needs to be swapped out. (However, debuggers must be aware of swapping to reinstall breakpoints.)

Some of the disadvantages are:

The size of the Intrinsic Unit library is limited by the number of MMU's supported by the hardware. (This could be expanded by treating the library as a tree structure or by swapping related segments and mapping them with a single MMU.)

The size of the largest program using Intrinsic Units is limited by the number of MMU's supported by the hardware.

There is a slight performance penalty in accessing global variables in Intrinsic Units indirectly via the table of pointers. (The Pascal Compiler puts such references into the pool of computations to optimize by saving results in registers.)

There is a penalty in speed in emulating the instructions IUJSR, IUJMP, IULEA, and IUPEA. For the most common instruction (JSR) the penalty is about 8:1 for the emulated version. This causes an overall 2:1 increase in the average procedure overhead (including LINK, UNLINK, return, argument passing and scrubbing, saving optimization registers, automatic stack expansion, etc.)

1.3 Basic Definitions

Segment

This term is used in two different senses which are related but distinct. From the hardware point of view, a segment is a portion of the logical address space which is mapped by an MMU and can include from 0 to 256 blocks of 512 bytes (zero to 128 K bytes). From the software point of view, a segment is a swappable piece of code of up to 32 K bytes. (The 32 K limitation is related to using signed words for PC relative branches.) There are also special segments, such as the stack segment and the jump-table segment. Where the distinction is important, the terms "logical segment" and "code segment" will be used.

Module (Block)

A module or block is a contiguous piece of memory. In unlinked files produced by the Pascal Compiler, a module is a procedure or function including string constants, set constants and embedded debug information. In unlinked files produced by the Assembler, a module is a single .PROC or .FUNC section of code. The IULinker also defines several other blocks of memory which are referenced and defined implicitly by the languages and run-time environment: the "global data" area (or initial stack), the "data pointer" area for accessing the global data of intrinsic units, and the jump table of a Main Program. The heap is the only part of the run-time environment which the Linker does not define as a block.

In a linked file, a block is a code segment (i.e. the smaller blocks of memory have been bound together into a larger contiguous piece of memory). Code modules (whether linked or unlinked) are represented in an object file by a set of object file records, beginning with a ModuleName block and ending with a EndBlock.

Note: the use of the term "module" to mean a "block" is due to historical roots. At some time in the future it would be nice to switch to the following terms, although this will involve massive edits to existing programs:

Block -- a contiguous piece of memory.
Module -- a block of data and one or more blocks of code.
Class -- a Module which can be instantiated with several data blocks.
Segment -- code blocks of one or more modules linked together.
ObjRecord -- a file format.

1.4 Types of Object Files

A object file contains one or more records of information relating to the execution of machine code. There are several types of object file:

Intrinsic Unit Directory (IUDirectory)
Intrinsic library and Main Program
Unlinked Units and Code

The general function of each type of object file is discussed below. The detailed specification of which blocks are present is given in the Section 7.0. Detailed formats of each block are given in Appendix A.

Intrinsic Unit Directory (IUDirectory)

Intrinsic Unit Dircectories are read and written by the IUManager. The "current" or "active" directory is found by convention in the file INTRINSIC.LIB on the O.S. boot volume or the Monitor root volume on the working device. Loaders read INTRINSIC.LIB to locate Intrinsic Segments. The IULinker uses INTRINSIC.LIB to compute the transitive closure of Intrinsic Units referenced and to assign absolute logical addresses. The Compiler reads INTRINSIC.LIB to locate the interfaces of Intrinsic Units.

Intrinsic Library and Main Program Files

Intrinsic library and main program files are written by the IULinker and loaded for execution by loaders on the Monitor and the O.S. Intrinsic library files contain linked intrinsic unit code which is ready to be loaded and executed as part of a main program. In addition, Intrinsic library files may contain linker information and unit interfaces used in the compilation and linking of other units and main programs. Intrinsic library and Main Program files can be stripped and packed by the PackSeg Utility in order to minimize disk space in a production system.

In the present development environment, the IUManager must be used to define Intrinsic Segments and Intrinsic Units before the IULinker links them. After the IULinker has linked an Intrinsic library file, the Intrinsic library file must be "Installed" using the IUManager. The installation operation places file relative location information in INTRINSIC.LIB so that the loaders can efficiently locate and load segments.

2.0 Grammatical Definition of Object Files

The grammar used is a form of Extended BNF similiar to that used by Wirth in describing Modula-2 [1]. The major differences are the adoption of a "list-of" construct suggested by DeRemer [2] and the interpretation of {E} as one or more occurances of E. The Extended BNF is capable of describing itself concisely:

2.1 GRAMMAR-GRAMMAR

Syntax:

```
syntax     = {production}.
production = NTSym "=" expr ".".
expr       = <term "|" >.
term       = {factor}.          -- one or more factors
factor     = TSym | NTSym   | "(" expr ")"
           | "[" expr "]"   | "<" expr TSym ">"
           | "{" expr "}".
```

Semantics:

```
E1 | E2 denotes either E1 or E2
        that is, one of two alternatives.
{E} denotes E, EE, EEE, etc.
        that is, one or more E's.
[E] denotes the empty string or E
        that is, an optional E.
<E P> denotes E, EPE, EPEPE, etc.
        that is, a list of E's separated by P's.
( ) are used for grouping.

NOTE: [{E}] denotes the empty string, E, EE, EEE, etc.
```

Scanning:

```
Comments are delimited by "--" and the end-of-line.
Special character terminals are in quotes.
The string """" is a quoted ". For example:
    The sentence:       "It's hot today!", he said.
    would be quoted:    """It's hot today!"", he said."
```

Conventions:

Syntactic class names begin with a lower case letter.
Terminal class names begin with an upper case letter.

Object file formats are descibed in the form used by the Development System during the development and testing of software. The "stripped and packed" formats produced by the PackSeg utility are documented in the comments. The PreLink file formats are also mentioned a few places in the comments, but have not been completely specified.

2.2 SOFTWARE CONFIGURATION FILES

```
objFile =
    iuDirectory     | sysPackTable
  | iuLibrary       | mainProg
  | unlinkedUnit    | unlinkedModule.

iuDirectory =
    VersionControl UnitLocation  SegLocation  FilesBlock
    [CodeBlock] EOFMark.

sysPackTable = VersionControl PackTable EOFMark.
```

The iuDirectory defines the intrinsic units and intrinsic segments which are available for use by main programs. By convention the name of the active iuDirectory is INTRINSIC.LIB. The optional CodeBlock contains the IU Trap Handler for the O.S. without LisaBug. This file cannot be packed.

The sysPackTable file contains the PackTable record used in packing any intrinsic library or main program files on the O.S. By convention the active PackTable is in PACKTABLE.LIB. This file cannot be packed.

2.3 LINKED FILES

```
iuLibrary =
    VersionControl SegLocation
    [InterfLocation]        -- stripped.  Present if Interfaces in file.
    UnitTable SegmentTable -- stripped, only used by Linker
    {UnitBlock}             -- interfaces are stripped
    {iuLibModule} EOFMark.

iuLibModule =
    ModuleName
    [{EntryPoint}]    -- stripped
    [{CommonReloc | ShortExternal}]  -- later, to support PreLink
    CodeBlock EndBlock.

mainProg =
    VersionControl [UnitTable] Executable [SegmentTable]
    {module} EOFMark.

module = ModuleName {otherModBlock} EndBlock.

otherModBlock =
      EntryPoint    | StartAddress | CommonReloc   | ShortExternal
    | CodeBlock     | Relocation   | External.
```

The SegLocation block in iuLibrary files is for future Loader support of slightly different versions of files on a system, i.e. packed Lisa Office System files and Development System versions with interfaces and linker information. Presently, one set of numbers is installed in INTRINSIC.LIB and is assumed valid for any file of the indicated name.

The InterfLocation block is used by the Compiler to quickly access interfaces in the UnitBlock(s). The UnitTable and the SegmentTable contain the transitive closure of intrinsic units used and intrinsic segments from code within a file. The UnitTable and SegmentTable are only present if intrinsic units are referenced.

The UnitBlock contains the size of the global data area for a particular intrinsic unit and optionally the interface or interface location information.

Presently iuLibrary modules do not contain relocation records. However this is planned for the PreLink and InstallLink programs which will support third party software development and distribution.

The Executable block contains the segment table and the jump table for the main program and regular unit segments.

2.4 UNLINKED FILES

```
unlinkedUnit =
                        -- later: VersionControl
    UnitBlock
    [{module}]          -- units can be definitions only
    EOFMark
    TextBlocks.         -- note: TextBlocks after EOFMark.

unlinkedModule =
            -- later: Version Control
    {module} EOFMark.
```

An unlinked unit file is the output of a compiler which is intended for "use" or "import" by another compilation. The kludge of having text blocks tacked on the end of the file is scheduled for replacement by compilation to an intermediate form which includes definitions.

A unlinked file is formed by a compiler or an assembler. Version control blocks are not presently placed on unlinked files but are sheduled to be shortly added (11.0).

3.0 Future Directions

3.1 Version Control

Version control will be needed for two purposes:

To prevent the execution from inconsistent library and main program files.

For consistency checking of a software configuration, i.e to support the Make facility and the Software Management Utility.

Version control for execution is scheduled for implementation after the second product build (internal use).

3.2 The Software Management Utility

A Software Management Utility is being developed which will facilitate the management of system dependencies and the automatic regeneration of a system based on consistency checking. This facility represents an extension of the UNIX "make" facility to include:

Distinction between interface and implementation editing changes.
Distinction between linking with regular units (code is copied) and intrinsic units (code is referenced).
Support for the concept of "reuseable" intrinsic library files.
Support for the concept of a "run-time" library directory.

The Software Management will provide for management of four levels of system implementation and configuration:

Run-time Systems
Intrinsic Library Files
Unlinked or Raw Object Files
Source Code Files

3.3 Symbolic Debugging

A new attempt at defining and implementing a Symbolic Debugger is being made. In the previous effort, the emphasis was on dumping symbolic information from the compiler into an independent .DBG file. In the current effort we are examining the possibility of passing more information through the .I-code file to the code generator. Some forms of debugging information will be embedded in the CodeBlock. Other forms of debugging will be introduced as new block types.

3.4 Other Languages

Currently we are planning for COBOL to generate object files which can be linked with the IULinker. We are also investigating the feasibility of bringing Modula-2 up on Lisa. Over the course of the next year Lisa will begin to support multi-language development projects.

4.0 Object Record Details

Object file records consist of a Header, an Invariant part and a Variant part.

The Header consists of a byte which indicates the BlockType followed by a three byte length field. The GetObjInvar and PutObjInvar procedures in ObjIO manage the details of the BlockType encoding and translate the particular encoding into an enumerated type.

The Invariant part is always a fixed length (possibly zero) for a given BlockType. The Invariant part characterizes the record. The following BlockTypes are currently supported:

```
BlockType=
   (ModuleName, EndBlock, EntryPoint,
    External, StartAddress, CodeBlock,
    Relocation, CommonReloc,
    ShortExternal,
    UnitBlock, InterfLoc,
    Executable, VersionCtrl,
    SegmentTable, UnitTable, SegLocation, UnitLocation, FilesBlock,
    PackedCode, PackTable,
    EOFMark);
```

Note: the current ObjIO Unit includes some additional BlockTypes that are supported for compatibility reasons, but are not intended for future support.

For each of the above BlockTypes there is a corresponding invariant record definition in ObjIO. For instance, the BlockType "ModuleName" has an invariant record definition "iModuleName". These are shown in detail below.

The Variant part may be missing, optional or of varying length depending on the BlockType. When present, the Variant part of an object file record usually consists of a varying number of fixed size entries. There are exceptions, however, such as the Executable block which has a complex variant structure (segment table, jump table and a few miscellaneous entries). The following VariantTypes are currently supported:

```
VariantType=
   (NoVariant,
    RefVariant, ShortRef, ModVariant, Comments,
    SegVariant, UnitVariant, IntfLocVariant,
    SegLocVariant, UnitLocVariant, FilesVariant,
    JumpTVariant, JTSegVariant, ObjectCode);
```

The association of a VariantType with each BlockType is expressed in two ways. In the invariant record definition a comment at the end documents the corresponding VarinatType. In addition, there is an array of information in ObjIO which contains the mapping. GetObjInvar and PutObjInvar manage the communication of this information to programs accessing object files.

In the definition of object file records, there are some standard types used in addition to Integer, LongInt, Boolean, Char, etc. The following types are introduced in the indicated Units:

```
     (* from Unit PasDefs: *)

  const NameStrLen = 8;          (* Length of Identifier Names *)
        MaxLStringLen = 80;      (* Reasonably long: error messages etc. *)

  type NameString = packed array [1..NameStrLen] of char;
       LString = String [MaxLStringLen];

       MemPtr = ^integer;        (* "untyped" pointer to memory *)
       ProcPtr = ^integer;       (* in place of Procedure variables *)

     (* from Unit ObjIO: *)

  type FileAddr = longint;       (* 0 based, byte address within a file *)
       MemAddr = longint;        (* 24-bit virtual address *)
       SegAddr = longint;        (* 0 based, byte address within a segment *)
```

Note: the name of the type NameString may need to change in the future due to a conflict with a different type in the O.S. and the lack of support for qualified names in Pascal.

4.1 VERSION CONTROL

VersionCtrl:

```
iVersionCtrl = record
  SysNum, MinSys,
  MaxSys, Reservl,
  Reserv2, Reserv3: longint;
end;
```

```
+---+--+--+--+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 99 |  size  | SysNum| MinSys| MaxSys|Reservl|Reserv2|Reserv3|
+---+--+--+--+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
1    2        5       9       13      17      21      25  28
```

99	-	Hexadecimal 99
size	-	Number of bytes in this block
SysNum	-	(reserved)
MinSys	-	(reserved)
MaxSys	-	(reserved)
Reservl	-	(reserved)
Reserv2	-	(reserved)
Reserv3	-	(reserved)

Note:

Contents are currently ignored by loaders and system programs for all fields.

Future plans:

See the VERSION CONTROL - SPECIFICATION document for detailed plans for releases 11.0 and 12.0.

4.2 MODULE BLOCKS

ModuleName:

```
             iModuleName=record
                ModuleName,
                SegmentName: NameString;
                CSize: LongInt;
                (* Comments *)
              end;

+---+---+---+---+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+---+
| 80 |  size  |  ModuleName    |  SegmentName  | CSize | Comments ...  |
+---+---+---+---+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+---+
  1   2     4  5            12 13           20 21   24               size
```

```
   80          - Hexadecimal 80
   size        - Number of bytes in this block
   ModuleName  - Blank padded ASCII name of this module
   SegmentName - ASCII name of segment in which this module will reside
```

Notes:

CSize is always zero. The actual CSize is in the EndBlock.
Comments are not currently generated.

Future plans:

CSize will be dropped.
Comments will be replaced with stack frame descriptor for debugging.
Linker will do language checking and size checking of args and locals.

EndBlock:

```
             iEndBlock=record
               CSize: LongInt;
             end;

+---+---+---+---+-+-+-+-+
| 81 |  size  | CSize |
+---+---+---+---+-+-+-+-+
 1    2     4  5     8
```

```
    81         - Hexadecimal 81
    size       - Number of bytes in this block (always 000008)
    CSize      - Numer of bytes in the code block for this module
```

Note:

CSize is the actual number of bytes of code in the CodeBlock, i.e. CSize is equal to the number of Variant bytes in the CodeBlock. By convention, the Monitor and O.S. loaders load the CodeBlock header and invariant part as well. So other records such as SegLocation blocks and the segment table in the Executable block generally indicate a code block size which is larger.

EntryPoint:

```
iEntryPoint=record
  LinkName,
  UserName: NameString;
  Loc: SegAddr;
  (* Comments *)
end;
```

```
+---+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+---+
| 82 |  size  |      LinkName      |      UserName      |  Loc  | Comments ...  |
+---+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+---+
 1   2     4 5              12 13              20 21    24 25            size
```

82	- Hexadecimal 82
size	- Number of bytes in this block
LinkName	- Blank padded ASCII linker name of entry point
UserName	- Blank padded ASCII user name of entry point
Loc	- Location of entry point relative to this module

Note:

Comments are not currently generated.

In Pascal files each module has only one EntryPoint and Loc is zero.

In Assembly language files there is an EntryPoint record for the .PROC or .FUNC and one for each .DEF

In Intrinsic library files with Linker information there is an EntryPoint record for each procedure or function in an Interface section.

For languages with nested scopes (such as Pascal) LinkName has a special format ("$nnnnnnn") for nested names or names in Implementation sections which do not need to be unique globally. LinkNames must be unique within a file. The Linker will remap the LinkNames to preserve uniqueness when reading the file. See Appendix C on the IULinker functions for more details.

Future plans:

Addition of UnitName to support qualified name references.

Switch from eight character case-insensitive names to longer case-sensitive names. The length will probably be either a fixed 16 characters or a varying 31 characters (i.e. an index in a NameTable).

External:

```
            iExternal=record
              LinkName,
              UserName: NameString;
              (* RefVariant *)
            end;

+---+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+----+--+--+--+
| 83 |  size  |     LinkName     |    UserName     | ref 1 | ... | ref n |
+---+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+----+--+--+--+
 1   2     4 5              12 13             20 21   24              size
```

83	- Hexadecimal 83
size	- Number of bytes in this block
LinkName	- Blank padded ASCII linker name of external reference
UserName	- Blank padded ASCII user name of external reference
ref 1	- Location of first reference relative to this block
...	- Other references
ref n	- Location of last reference

Note:

See the notes and futures plans for names under EntryPoint.

StartAddress:

```
            iStartAddress=record
              Start: SegAddr;
              GSize: LongInt;
              (* Comments *)
            end;

+---+--+--+--+--+--+--+--+--+--+--+--+--+-------+
| 84 |  size  | Start | GSize | Comments ...  |
+---+--+--+--+--+--+--+--+--+--+--+--+--+-------+
 1    2     4  5    8 9    12 13           size
```

84	- Hexadecimal 84
size	- Number of bytes in this block
Start	- Starting address relative to this block
GSize	- Number of bytes in the global data area
Comments	- Arbitrary information. Ignored by the Linker.

Note:

Comments are not currently generated.

CodeBlock:

```
            iCodeBlock=record
              Addr: SegAddr;
              (* ObjectCode *)
            end;

+---+--+--+--+-+-+-+-+-+-+-+-+-+------+
| 85 |  size  | Addr   | ObjectCode ... |
+---+--+--+--+-+-+-+-+-+-+-+-+-+------+
 1    2    4  5     8 9            size
```

```
    85         - Hexadecimal 85
    size       - Number of bytes in this block
    Addr       - Address of first byte of code
    ObjectCode - The object code.  Always an even number of bytes.
```

Note:

For raw object files (unlinked) the address is always 0.

For linked files the address is an absolute address in the logical address space. (MMU # times 128 K + const).

Relocation:

```
            iRelocation=record
              (* RefVariant *)
            end;

            iRefVariant=SegAddr;

+---+--+--+--+-+-+-+-+------+-+-+-+-+
| 86 | size   | ref 1 | ...  | ref n |
+---+--+--+--+-+-+-+-+------+-+-+-+-+
 1    2    4  5     8           size
```

```
     86        - Hexadecimal 86
     size      - Number of bytes in this block
     ref 1     - Location of first address to relocate
     ...       - Other addresses
     ref n     - Location of last address to relocate
```

Note:

Relocation records are generated by the old Linker (partial links) and by the old Library program. They are not supported by the current Linker.

Future plans:

Reloction records will be used by the PreLink and InstallLink versions of the Linker.

CommonReloc:

```
              iCommonRelocation=record
                CommonName: NameString;
                (* RefVariant *)
              end;

              iRefVariant=SegAddr;
```

```
+----+--+--+--+-+-+-+-+-+-+-+-+-+-+-+-+-------+-+-+-+
| 87 |  size  |  CommonName     | ref 1 | ...  | ref n |
+----+--+--+--+-+-+-+-+-+-+-+-+-+-+-+-+-------+-+-+-+
 1    2     4 5             12 13    16          size
```

```
   87          - Hexadecimal 87
   size        - Number of bytes in this block
   CommonName  - Blank padded ASCII name of common block
   ref 1       - Location of first reference relative to this module
   ...         - Other references
   ref n       - Location of last reference
```

Note:

Common relocation references in the code are zero based relative to the beginning of the named regular unit.

ShortExternal:

```
              iShortExternal=record
                LinkName,
                UserName: NameString;
                (* ShortRef *)
              end;

              iShortRef=Integer;
```

```
+----+--+--+--+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+--+--+-----+--+--+
| 89 |  size  |     LinkName     |   UserName      | ref1| ... | refn|
+----+--+--+--+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+--+--+-----+--+--+
  1   2     4 5             12 13            20 21 22          size
```

```
    89         - Hexadecimal 89
    size       - Number of bytes in this block (always 000016)
    LinkName   - Blank padded ASCII linker name of external reference
    UserName   - Blank padded ASCII user name of external reference
    ref1       - Location of first address to relocate
     ...       - Other addresses
    refn       - Location of last address to relocate
```

4.3 UNIT BLOCKS

UnitBlock:

```
        iUnitBlock=record
          UnitName: NameString;
          CodeAddr,
          TextAddr: FileAddr;
          TextSize,
          GlobalSize: LongInt;
          UnitType: integer;      (* 0=Reg, 1=Intrin, 2=Shared *)
          (* comments = interface section of Unit (compressed) *)
        end;
```

```
+---+---+---+---+-+-+-+-+-+-+-+---+---+---+---+---+---+---+---+---+---+---+---
| 92 | size     |   UnitName        | CodeAddr  | TextAddr  | TextSize  |...
+---+---+---+---+-+-+-+-+-+-+-+---+---+---+---+---+---+---+---+---+---+---+---
 1    2      4 5               12 13        16 17        20 21        24

         ---+---+---+---+---+------+------+---+---+---+--------+
         ...| GlobalSize| UnitType| Comments ...    |
         ---+---+---+---+---+------+------+---+---+---+--------+
            25       28 29     30 31            size
```

92	- Hexadecimal 92
size	- Number of bytes in this block (always 00001E)
UnitName	- Name of this unit
CodeAddr	- Disk address of module
TextAddr	- Disk address of text block
TextSize	- Size of text block
GlobalSize	- Number of bytes of globals in this unit
UnitType	- 0=Regular, 1=Intrinsic, 2=Shared
Comments	- Compressed ASCII text of Interface

Note:

In an unlinked (raw) file:

CodeAddr is the address of the first Module Name Block (i.e. the first byte after this UnitBlock).

TextAddr is the (block aligned) File Address of the Interface (past the EOFMark).

TextSize is the size of the interface (= n*1024) where n is the number of text pages.

Comments is missing.

The Interface is found in standard .TEXT file blocks.

In a linked (intrinsic library) file:

CodeAddr is 0.

TextAddr is 0.

TextSize is 0.

Comments is either empty (no interfaces in the library) or contains the compressed interface (blanks and meaningless comments removed).

For Pascal the Interface is defined to begin with the character after the semicolon in the "Unit Foo;" statement and extends through the word "implementation".

Future Plans:
The kludge of having Text blocks at the end of the file may not be supported forever. Compilers should be designed to get interfaces from the variant part of the UnitBlock record, whether they are stored in text form or are represented as intermediate code.

InterfLoc:
```
             iInterfLoc= record
               (* IntfLocVariant *)
             end;

+---+--+--+--+-+-+-+------+-+-+-+
| 86 | size   | loc 1 | ...  | loc n |
+---+--+--+--+-+-+-+------+-+-+-+
 1    2     4  5     8          size
```

```
   92          - Hexadecimal 92
   size        - Number of bytes in this block
   loc 1       - Location record for first unit interface
   ...         - Other location records
   loc n       - Location record for last unit interface
```

Note:
The interface location block is only present if the +I option has been specified to the Linker to include interfaces when linking an intrinsic library file.

IntfLocVariant:
```
                  iIntfLocVariant = record
                    UnitName: NameString;
                    IfLoc: FileAddr;
                  end;

             +-+-+-+-+-+-+-+-+-+-+-+-+
             |  UnitName     | IfLoc |
             +-+-+-+-+-+-+-+-+-+-+-+-+
              1            8 9    12
```

```
              UnitName      - Blank padded ASCII Unit Name
              IfLoc         - File Address of first byte of Interface
```

4.4 MAIN PROGRAM

Executable:

```
iExecutable=record
  JTLaddr: MemAddr;
  JTSize,
  DataSize, MainSize,
  JTSegDelta, StkSegDelta,
  DynStack, MaxStack,
  MinHeap, MaxHeap: LongInt;
  (* Unknown = numSegs + JTSegVariants +
       numDescriptors + JumpTVariants + other stuff *)
end;
```

```
+---+-+-+-+-+-+-+-+-+-+-+-+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
| 98| size|JTLaddr| JTSize|   DataSize |  MainSize |
JTSegDelta|StkSegDelta|..
+---+-+-+-+-+-+-+-+-+-+-+-+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
1     2   4 5     8 9     12 13        16 17        20 21        24 25        28

   ---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+-------+
...|  DynStack |  MaxStack |  MinHeap  |  MaxHeap  | jump table ...     |
   ---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+-------+
     29        32 33        36 37        40 41        44 45                 size
```

98	- Hexadecimal 98
size	- Number of bytes in this block
JTLaddr	- Absolute load address of jump table
JTsize	- Number of bytes in jump table
DataSize	- Total number of bytes in regular units global data areas
MainSize	- Size of main program global data area
JTSegDelta	- Distance from base of segment to beginning of data pointers
StkSegDelta	- Distance from JTSegDelta to A5 at runtime
DynStack	- Initial dynamic stack size
MaxStack	- Maximum total stack size
MinHeap	- Initial heap size
MaxHeap	- Maximum total heap size
jump table	- The jump table itself.

The format of the jump table is:

```
+--------------------+
| Number of segments |  2 bytes
+--------------------+
| Main Segment Table | 12 bytes
+--------------------+
| Segment Table #2   | 12 bytes
+--------------------+
|       . . .        |
+--------------------+
| Segment Table #n   | 12 bytes
+--------------------+
|Number of Descriptors|  2 bytes
+--------------------+
| Start Descriptor   | 6 bytes
+--------------------+
| S#1 P#2 Descriptor |
+--------------------+
|      . . .         |
+--------------------+
| S#1 P#n1 Descriptor |
+--------------------+
| S#2 P#1 Descriptor |
+--------------------+
|      . . .         |
+--------------------+
| S#2 P#n2 Descriptor |
+--------------------+
| S#3 P#1 Descriptor |
+--------------------+
|      . . .         |
+--------------------+
| S#m P#nN Descriptor | 6 bytes
+--------------------+
|      Old Stuff     |
+--------------------+
```

Note:

By convention, the main segment has a blank name, and is the first segment in the jump table. Also, the first descriptor in the first segment is the entry point for the main program.

Segment Table Entry:

```
          iJTSegVariant = record
              SegmentAddr: FileAddr;
              SizePacked: integer;
              SizeUnpacked: integer;
              MemLoc: MemAddr;
          end;

  +---+---+---+---+-----+-----+------+------+---+---+---+
  |  SegmentAddr  | SizePacked| SizeUnpacked|    MemLoc   |
  +---+---+---+---+-----+-----+------+------+---+---+---+
   1             4 5         6 7           8 9          12
```

```
SegmentAddr     - File address of either CodeBlock or PackedCode block
SizePacked      - Number of bytes in PackedCode record
SizeUnpacked    - Number of bytes in (unpacked) Code record
MemLoc          - Absolute logical address of segment
```

Note:

If SizePacked = 0 then segment is not packed.

SizePacked and SizeUnpacked include the invariant part of the record.

Future plans:

Both SizePacked and SizeUnpacked will become LongInts at the next non-compatible object code release.

Jump Table Descriptor:

```
          iJumpTVariant = record
              JumpL: integer;
              AbsAddr: MemAddr;
          end;

          +---+---+--+--+--+--+
          | JumpL |  AbsAddr  |
          +---+---+--+--+--+--+
           1     2 3         6
```

```
JumpL    - JMP.L $xxxxxxx instruction
AbsAddr  - Absolute address of procedure in logical address space
```

4.5 INTRINSIC UNITS

SegmentTable:

```
              iSegmentTable = record
                nSegments: integer;
                (* SegVariant *)
              end;
```

```
+---+--+--+--+-----+-----+----------+-   -+----------+
| 9A | size    | nSegments | segInfo1 | ... | segInfoN |
+---+--+--+--+-----+-----+----------+-   -+----------+
 1    2      4 5     7          25              size
```

```
     9A         - Hexadecimal 9A
     size       - Number of bytes in this block
     nSegments  - Number of segment descriptors in table
     segInfo1   - First SegVariant record
       ...
     segInfoN   - Last SegVariant record
```

Note:

The Segment Table contains the transitive closure of the intrinsic segments referenced by segments in this file. The transitive closure is currently computed fairly loosely: inclusion of a file in the Linker input list is taken as a reference to all the segments in the file. This is consistent with the notion of "reuseable" and the notion of "changes in implementation" not affecting reuseability, i.e. references can be added to other parts of a lower level library without affecting the transitive closure computation.

```
              iSegVariant = record
                SegName: NameString;
                SegNumber: integer;
                Version1: longint;
                Version2: longint;
              end;
```

```
+-+-+-+-+-+-+-+-+----+----+-+-+-+-+-+-+-+-+
|   SegName      |SegNumber| Version1 | Version2 |
+-+-+-+-+-+-+-+-+----+----+-+-+-+-+-+-+-+-+
 1               9        11         15       18
```

```
 SegName     - Segment Name
 SegNumber   - Segment (MMU) number
 Version1    - (reserved)
 Version2    - (reserved)
```

UnitTable:

```
            iUnitTable = record
              nUnits,
              maxunit: integer;
              (* UnitVariant *)
            end;
```

```
+----+--+--+--+--+--+----+----+----------+-   -+----------+
| 9B | size     | nUnits| maxunit |UnitInfo1 | ... |UnitInfoN |
+----+--+--+--+--+--+----+----+----------+-   -+----------+
 1    2        4 5      7        9          21                size
```

```
9B        - Hexadecimal 9B
size      - Number of bytes in unit table block
nUnits    - Number of unit descriptors in table.
maxunit   - maximum unit number found in the table.
UnitInfo1 - First UnitVariant record
  ...
UnitInfoN - Last UnitVariant record
```

Example:

If units number 1, 7, and 11 are present then nUnits=3 and maxunit=11.

```
iUnitVariant = record
  UnitName: NameString;
  UnitNumber: integer;
  UnitType: integer;
end;
```

```
+-+-+-+-+-+-+-+-+-----+-----+----+----+
|     UnitName     | UnitNumber| UnitType|
+-+-+-+-+-+-+-+-+-----+-----+----+----+
1                9         10 11      12
```

```
UnitName     - Unit Name
UnitNumber   - Index into data pointer table
UnitType     - 0=Regular, 1=Intrinsic, 2=Shared
```

Note:

UnitType = 0 would be an error.

SegLocation:

```
iSegLocation = record
  nSegments: integer;
  (* SegLocVariant *)
end;
```

```
+----+--+--+--+----+----+----------+-   -+----------+
| 9C | size   |nSegments| segInfo1 | ... | segInfoN |
+----+--+--+--+----+----+----------+-   -+----------+
 1    2     4 5       6 7                   size
```

```
9C        - Hexadecimal 9C
size      - Number of bytes in segLocation block
nSegments - Number of segment descriptors in table.
segInfo1  - First SegLocVariant record
  ...
segInfoN  - Last SegLocVariant record
```

```
iSegLocVariant = record
  SegName: NameString;
  SegNumber: integer;
  Version1, Version2: longint;
  FileNumber: integer;
  FileLocation: FileAddr;
  SizePacked, SizeUnpacked: integer;
end;
```

```
+-+-+-+-+-+-+-+-+----+----+--+--+--+--+--+--+--+---
|     SegName    |SegNumber| Version1  | Version2  |....
+-+-+-+-+-+-+-+-+----+----+--+--+--+--+--+--+--+---
1                9          11          15        18

   ---+-----+-----+---+---+---+-----+-----+-----+------+
   ... | FileNumber|  FileLocation | SizePacked| SizeUnpacked|
   ---+-----+-----+---+---+---+-----+-----+-----+------+
      19         20 21          24 25       26 27          28
```

```
SegName      - Segment Name
SegNumber    - MMU number
Version1     - (reserved)
Version2     - (reserved)
FileNumber   - Index into the FilesBlock file table
FileLocation - Location within file of CodeBlock
SizePacked   - Number of bytes in PackedCode record
SizeUnpacked - Number of bytes in (unpacked) Code record
```

Note:

If SizePacked = 0 then Segment is not packed.

FileLocation may become invalid when variations are allowed in an intrinsic unit or main program file.

SizePacked and SizeUnpacked will become longints.

UnitLocation:

```
            iUnitLocation = record
              nUnits: integer;
              (* UnitLVariant *)
            end;
```

```
+----+--+--+--+---+---+----------+-    -+----------+
| 9D | size     | nUnits|UnitInfo1 | ... |UnitInfoN |
+----+--+--+--+---+---+----------+-    -+----------+
 1    2       4 5      7          23              size
```

```
9D        - Hexadecimal 9D
size      - Number of bytes in unitLocation block
nUnits    - Number of unit descriptors in table.
UnitInfo1 - First UnitLVariant record
  ...
UnitInfoN - Last UnitLVariant record
```

```
            iUnitLVariant = record
              UnitName: NameString;
              UnitNumber: integer;
              FileNumber, UnitType: FileByte;
              DataSize: longint;
            end;
```

```
+-+-+-+-+-+-+-+-+-----+-----+----------+--------+--+--+--+--+
|     UnitName    | UnitNumber|FileNumber|UnitType|  DataSize |
+-+-+-+-+-+-+-+-+-----+-----+----------+--------+--+--+--+--+
 1                9         10 11        12       13       16
```

```
UnitName     - Unit Name
UnitNumber   - Index into data pointer table
FileNumber   - Index into the FilesBlock file table
UnitType     - See UnitTable above
DataSize     - Size in bytes of global data area for unit
```

FilesBlock:

```
          iFilesBlock = record
            nFiles: integer;
            (* Unknown = FilesVariant + string table *)
          end;
```

```
+----+---+--+--+----+----+--+--+--+--+--+--  --+--+--+--+--+--+---------------+
| 9E | Size    | nFiles  | FileInfo1 | ...  | FileInfoN | StringTable ... |
+----+---+--+--+----+----+--+--+--+--+--+--  --+--+--+--+--+--+---------------+
 1     2     4  5    6    7         13                                      size
```

```
    9E          - Hexadecimal 9E
    nFiles      - number of file descriptors in block.  Each Fileinfo record
    FileInfo1   - First FilesVariant record
      ...
    FileInfoN   - Last FilesVariant record
```

```
              iFilesVariant = record
                FileNumber: integer;
                NameAddr: FileAddr;
              end;
```

```
             +-----+-----+--+--+--+--+
             | FileNumber|  NameAddr |
             +-----+-----+--+--+--+--+
              1         2 3         6
```

```
             FileNumber     - Index into the FilesBlock file table
             NameAddr       - File address of name string
```

Note:

Each StringTable entry has the format of a Pascal string, i.e. the strings begins on an even byte and the first byte is a length byte indicating how the length of the string.

4.6 CODE COMPACTION

PackedCode:

```
          iPackedCode = record
            addr: MemAddr;
            csize: longint;
            (* Unknown = packed object code *)
          end;
```

```
     addr       - Absolute address in logical address space
     csize      - Size in bytes of the code when unpacked
```

PackTable:

```
            iPackTable = record
              packversion: longint;
              (* Unknown = translation table *)
            end;
```

Note:

The packversion field was originally intended to indicate changes in the packing algorithm. With the O.S. supporting one PackTable for the system, packversion could also be used to indicate which table.

4.7 THE END

EOFMark:

```
+--+--+--+--+
|00| 000004 |
+--+--+--+--+
 1  2      4
```

The EOFMark block marks the end of an object file (almost).

Note:

Text blocks can occur past the EOFMark.

References

---

[1] Niklaus Wirth, "MODULA-2", Institut fur Informatik der ETH, 1980.

[2] Frank DeRemer and Tom Pennello, "Translator Writing System (TWS) Manual", MetaWare, Inc., 1981.

Appendix A. ObjIOLib Interface

```
(*****************************************************************************)
(*                                                                           *)
(*                         File: LIB:OBJIO                                   *)
(*                                                                           *)
(*                     (C) Copyright 1981, 1982                              *)
(*                        Apple Computer, Inc.                               *)
(*                                                                           *)
(*                                                              9-Jul-82     *)
(*****************************************************************************)

{$S LIB1 }

unit ObjIO;
intrinsic;

  (* ObjIO is a unit defining and providing blockwise and bytewise read/ *)
  (* write access to object-format files.  All I/O goes through FileIO. *)

  interface
    uses
      (*$U PASDEFS.OBJ *) PasDefs,
      (*$U UTILITY.OBJ *) Utility,
      (*$U FILEIO.OBJ *) FileIO;

  (* Note: distinctions --      *)
  (*      OldExecutable (old compilers, either machine) *)
  (*      PhysicalExec  (New compiler, old linker, either machine, physical) *)
  (*      Executabe (New compiler, either linker, new machine, logical) *)
  (* New linker links Intrinsic Units and produces a version control record. *)

    type
      BlockType=(ModuleName, EndBlock, EntryPoint,
          External, StartAddress, CodeBlock,
          Relocation, CommonReloc, CommonDef,
          ShortExternal, QuickLoad, OldExecutable,
          LibModule, LibEntry, UnitBlock, InterfLoc,
          PhysicalExec, Executable, VersionCtrl,
          SegmentTable, UnitTable, SegLocation, UnitLocation, FilesBlock,
          PackedCode, PackTable, DebugSymbols,
          DebugEntry, DebugCommon, EOFMark, UnknownBlock);

      VariantType=(NoVariant, (* must be first *)
          RefVariant, ShortRef, ModVariant, Comments,
          SegVariant, UnitVariant, IntfLocVariant,
          SegLocVariant, UnitLocVariant, FilesVariant,
          JumpTVariant, JTSegVariant, ObjectCode, ProcHeap,
          OldJumpTV, OldJTSegV,
          (* must be last *) UnknownVariant);

       FileAddr = longint;      (* 0 based, byte address within a file *)
```

```
 MemAddr = longint;       (* 24-bit virtual address *)
 SegAddr = longint;       (* 0 based, byte address within a segment *)

          (* Variant Definitions *)

iRefVariant=SegAddr;

iShortRef=Integer;

iModVariant=Integer;

iSegVariant = record
  SegName: NameString;
  SegNumber: integer;
  Version1: longint;
  Version2: longint;
end;

iUnitVariant = record
  UnitName: NameString;
  UnitNumber: integer;
  UnitType: integer;
end;

iIntfLocVariant = record
  UnitName: NameString;
  IfLoc: FileAddr;
end;

iSegLocVariant = record
  SegName: NameString;
  SegNumber: integer;
  Version1: longint;
  Version2: longint;
  FileNumber: integer;
  FileLocation: FileAddr;
  SizePacked: integer;    (* size of PackedCode record *)
  SizeUnpacked: integer;  (* size of CodeBlock record *)
end;

iUnitLVariant = record
  UnitName: NameString;
  UnitNumber: integer;
  FileNumber, UnitType: FileByte;
  DataSize: longint;
end;

iFilesVariant = record
  FileNumber: integer;
  NameAddr: FileAddr;
  (* one per file, followed by string table *)
end;

iJumpTVariant = record
```

```
        JumpL: integer;
        AbsAddr: MemAddr;
    end;

    iOldJumpTV = record
        RelOffset: longint;
        Noop: integer;          (* not in Memory = JMP.L *)
        Jump: integer;          (* not in Memory = Adrress of %%LOADIT *)
        PCRel: integer;
    end;

    iOldJTSegV = record
        Addr1: MemAddr;         (* Address of First Proc Descriptor *)
        FileLoc: FileAddr;
        CodeSize: longint;
        MemLoc: MemAddr;
        RetAddr: MemAddr;
        RefCount: longint;
        ActiveList: MemAddr;  (* -1 = End Of List ?? *)
        Reserved: longint;
    end;

    iJTSegVariant = record
        SegmentAddr: FileAddr;          (* points to CodeBlock or PackedCode *)
        SizePacked: integer;            (* size of PackedCode record *)
        SizeUnpacked: integer;          (* size of CodeBlock record *)
        MemLoc: MemAddr;                (* Logical Addr *)
    end;


              (* Invariant Definitions: *)

    iModuleName=record
      ModuleName,
      SegmentName: NameString;
      CSize: LongInt;
      (* Comments *)
    end;

    iEndBlock=record
      CSize: LongInt;
    end;

    iEntryPoint=record
      LinkName,
      UserName: NameString;
      Loc: SegAddr;
      (* Comments *)
    end;

    iExternal=record
      LinkName,
      UserName: NameString;
```

```
    (* RefVariant *)
  end;

  iStartAddress=record
    Start: SegAddr;
    GSize: LongInt;
    (* Comments *)
  end;

  iCodeBlock=record
    Addr: SegAddr;
    (* ObjectCode *)
  end;

  iRelocation=record
    (* RefVariant *)
  end;

  iCommonRelocation=record
    CommonName: NameString;
    (* RefVariant *)
  end;

  iCommonDefinition=record
    CommonName: NameString;
    DSize: LongInt;
    (* Comments *)
  end;

  iShortExternal=record
    LinkName,
    UserName: NameString;
    (* ShortRef *)
  end;

  iQuickLoad=record
    StartLoc: SegAddr;
    DataSize: LongInt;
    (* ObjectCode *)
  end;

  iLibModule=record
    ModuleName: NameString;
    ModSize: LongInt;
    CodeAddr,
    TextAddr: FileAddr;
    TextSize: LongInt;
    NrMods: Integer;
    (* ModVariant *)
  end;

  iLibEntry=record
    LinkName: NameString;
    Module: Integer;
```

```
    Address: SegAddr;
  end;

  iUnitBlock=record
    UnitName: NameString;
    CodeAddr,
    TextAddr: FileAddr;
    TextSize,
    GlobalSize: LongInt;
    UnitType: integer;      (* 0=Reg, 1=Intrin, 2=Shared *)
    (* comments = interface section of Unit (compressed) *)
  end;

  iInterfLoc= record
    (* IntfLocVariant *)
  end;

  iExecutable=record
    JTLaddr: MemAddr;
    JTSize,
    DataSize,               (* Global Area, Reg Units *)
    MainSize,               (* Global Area, Main Program *)
    JTSegDelta,             (* Jump Table Segment Delta *)
    StkSegDelta,            (* Stack Segment Delta *)
    DynStack,               (* Initial Dynamic Stack Size *)
    MaxStack,               (* Max. Total Stack Size *)
    MinHeap,                (* Initial Heap Size *)
    MaxHeap: LongInt;       (* Max. Total Heap Size *)
    (* Unknown = numSegs + JTSegVariants +
                 numDescriptors + JumpTVariants + other stuff *)
  end;

  iOldExecutable=record
    JTLaddr: MemAddr;
    JTSize,
    DataSize: LongInt;      (* Global Area, Reg Units *)
    (* Unknown = numSegs + OldJTSegVs + OldJumpTVs + other stuff *)
  end;

  iPhysicalExec=record
    JTLaddr: MemAddr;
    JTSize,
    DataSize,               (* Global Area, Reg Units *)
    MainSize,               (* Global Area, Main Program *)
    JTSegDelta,             (* Jump Table Segment Delta *)
    StkSegDelta: LongInt;   (* Stack Segment Delta *)
    (* Unknown = numSegs + OldJTSegVs +
                 DummyPtr + OldJumpTVs + other stuff *)
  end;

  iVersionCtrl = record
    sysNum, minSys,
    maxSys, Reservl,
    Reserv2, Reserv3: longint;
```

```
end;

iSegmentTable = record
  nSegments: integer;
  (* SegVariant *)
end;

iUnitTable = record
  nUnits,
  maxunit: integer;
  (* UnitVariant *)
end;

iSegLocation = record
  nSegments: integer;
  (* SegLocVariant *)
end;

iUnitLocation = record
  nUnits: integer;
  (* UnitLVariant *)
end;

iFilesBlock = record
  nFiles: integer;
  (* Unknown = FilesVariant + string table *)
end;

iPackedCode = record
  addr: MemAddr;
  csize: longint;
  (* Unknown = packed object code *)
end;

iPackTable = record
  packversion: longint;
  (* Unknown = translation table *)
end;

iDebugSymbols=record
  UserName,
  SegName: NameString;
  ProcBase,
  ProcSyms,
  ProcStmt,
  ProcNode,
  UsesSize: LongInt;
  { if UsesSize<>0 then ... these have valid values: }
  HoleBase,
  HoleTop,
  MapBase,
  MapTop: LongInt;
  MapName: NameString;
  { later }
```

```
      (* ProcHeap *)
    end;

    iDebugEntry=record
      UserName: NameString;
      EntrySeg: Longint;
      EntryLoc: SegAddr;
      (* Comments *)
    end;

    iDebugCommon=record
      UnitName: NameString;
      CommonBase: MemAddr;
      (* Comments *)
    end;

    iUnknown=record
       (* UnknownVariant *)
    end;

   ObjBlock=record
      Variant: VariantType;
      NrVariants: LongInt;
      case BlockHeader: BlockType of
        ModuleName:    (bModuleName:             iModuleName);
        EndBlock:      (bEndBlock:               iEndBlock);
        EntryPoint:    (bEntryPoint:             iEntryPoint);
        External:      (bExternal:               iExternal);
        StartAddress:  (bStartAddress:           iStartAddress);
        CodeBlock:     (bCodeBlock:              iCodeBlock);
        Relocation:    (bRelocation:             iRelocation);
        CommonReloc:   (bCommonReloc:            iCommonReloc);
        CommonDef:     (bCommonDef:              iCommonDef);
        ShortExternal:(bShortExternal:           iShortExternal);
        QuickLoad:     (bQuickLoad:              iQuickLoad);
        OldExecutable:(bOldExecutable:           iOldExecutable);
        LibModule:     (bLibModule:              iLibModule);
        LibEntry:      (bLibEntry:               iLibEntry);
        UnitBlock:     (bUnitBlock:              iUnitBlock);
        InterfLoc:     (bInterfLoc:              iInterfLoc);
        PhysicalExec:  (bPhysicalExec:           iPhysicalExec);
        Executable:    (bExecutable:             iExecutable);
        VersionCtrl:   (bVersionCtrl:            iVersionCtrl);
        SegmentTable:  (bSegmentTable:           iSegmentTable);
        UnitTable:     (bUnitTable:              iUnitTable);
        SegLocation:   (bSegLocation:            iSegLocation);
        UnitLocation:  (bUnitLocation:           iUnitLocation);
        FilesBlock:    (bFilesBlock:             iFilesBlock);
        PackedCode:    (bPackedCode:             iPackedCode);
        PackTable:     (bPackTable:              iPackTable);
        DebugSymbols:  (bDebugSymbols:           iDebugSymbols);
        DebugEntry:    (bDebugEntry:             iDebugEntry);
        DebugCommon:   (bDebugCommon:            iDebugCommon);
        UnknownBlock:  (bUnknown:                iUnknownBlock);
```

```
    end;

    ObjVarBlock = record
      case VarHeader: VariantType of
        RefVariant:     (bRefVariant:           iRefVariant);
        ShortRef:       (bShortRef:             iShortRef);
        ModVariant:     (bModVariant:           iModVariant);
        SegVariant:     (bSegVariant:           iSegVariant);
        UnitVariant:    (bUnitVariant:          iUnitVariant);
        IntfLocVariant: (bIntfLocVariant:       iIntfLocVariant);
        SegLocVariant:  (bSegLocVariant:        iSegLocVariant);
        UnitLocVariant: (bUnitLVariant:         iUnitLVariant);
        FilesVariant:   (bFilesVariant:         iFilesVariant);
        OldJumpTV:      (bOldJumpTV:            iOldJumpTV);
        OldJTSegV:      (bOldJTSegV:            iOldJTSegV);
        JumpTVariant:   (bJumpTVariant:         iJumpTVariant);
        JTSegVariant:   (bJTSegVariant:         iJTSegVariant)
    end;

    ObjHandle=^ObjDesc;
    ObjDesc=record
      ObjFile: FileHandle;
      NextBlock: FileAddr;
    end;

  procedure InitObjFile (var ObjPtr: ObjHandle; nBlocks: integer);
    (* InitObjFile initializes ObjPtr and allocates a buffer of nBlocks *)

  procedure OpenObjFile (var ObjPtr: ObjHandle; FileName: LString;
      NewFile: Boolean);
    (* OpenObjFile initializes ObjPtr to the file FileName.  The file is *)
    (* scratched if NewFile is set. *)

  procedure ZeroObjEnd (ObjPtr: ObjHandle);
    (* Zero ObjEnd fills out the current block with zeroes *)

  procedure CloseObjFile (ObjPtr: ObjHandle; Save: Boolean);
    (* CloseObjFile closes an object file.  If Save is set then the file is *)
    (* locked.  Otherwise, the file is left in the state it was in before *)
    (* it was opened. *)

  procedure GetObjPtr (ObjPtr: ObjHandle; var BytePtr: FileAddr);
    (* GetObjPtr returns the position of ObjPtr's "read/write head". *)

  procedure GetObjBlockPtr (ObjPtr: ObjHandle; var BytePtr: FileAddr);
    (* sets BytePtr to the file location of the next ObjBlock to be read *)

  procedure SetObjPtr (ObjPtr: ObjHandle; BytePtr: FileAddr);
    (* SetObjPtr positions the "read/write head" BytePtr bytes from the *)
    (* beginning of ObjPtr.  The invariant access flow is not altered, *)
    (* that is to say the next (Get/Put)ObjInvar accesses the sequentially *)
    (* next invariant following the variant that we're in before calling *)
    (* SetObjPtr. *)
```

```
procedure SetObjBlockPtr (ObjPtr: ObjHandle; BytePtr: FileAddr);
  (* SetObjBlockPtr positions the "read/write head" BytePtr bytes from *)
  (* the beginning of ObjPtr.  BytePtr must point to the beginning of an *)
  (* invariant.  That invariant will be accessed with the next *)
  (* (Get/Put)ObjInvar. *)

procedure SkipObjBytes (ObjPtr: ObjHandle; NrBytes: LongInt);
  (* SkipObjBytes moves the file pointer of file ObjPtr NrBytes bytes. *)

procedure SetObjInvar (var B: ObjBlock; InvarType: BlockType;
    VarSize: LongInt);
  (* SetObjInvar sets some fields in B.  B is of InvarType type with *)
  (* VarSize bytes in its variant. *)

procedure CopyObjSeq (InObj, OutObj: ObjHandle; NrBytes: Integer);
  (* CopyObjSeq copies a sequence of NrBytes bytes from InObj to OutObj. *)

procedure GetObjInvar (ObjPtr: ObjHandle; var Stuff: ObjBlock);
  (* GetObjInvar reads the invariant part of an object block.  *)
  (* The user can read the variant part, if so desired. *)

procedure GetObjVar (ObjPtr: ObjHandle; VarType: VariantType;
                     var Stuff: ObjVarBlock);
  (* GetObjVar reads a variant part of the specified type *)
  (* into the ObjVarBlock *)

procedure GetObjName (ObjPtr: ObjHandle; var N: NameString);
  (* GetObjName reads a name from file ObjPtr. *)

procedure GetObjSeq (ObjPtr: ObjHandle; Stuff: Ptr; NrBytes: Integer);
  (* GetObjSeq moves NrBytes bytes from ObjPtr to the area pointed to by *)
  (* Stuff. *)

procedure GetObjByte (ObjPtr: ObjHandle; var B: Byte);
  (* GetObjByte reads a byte from file ObjPtr. *)

procedure GetObjWord (ObjPtr: ObjHandle; var W: Integer);
  (* GetObjWord reads an integer from file ObjPtr. *)

procedure GetObjLong (ObjPtr: ObjHandle; var L: LongInt);
  (* GetObjLong reads a longint from file ObjPtr. *)

procedure PutObjInvar (ObjPtr: ObjHandle; var Stuff: ObjBlock);
  (* PutObjInvar writes the invariant part of an object block.  *)

procedure PutObjVar (ObjPtr: ObjHandle; VarType: VariantType;
                     var Stuff: ObjVarBlock);
  (* PutObjVar writes a variant part of the specified type *)
  (* from the ObjVarBlock *)

procedure PutObjName (ObjPtr: ObjHandle; N: NameString);
  (* PutObjName writes a name to file ObjPtr. *)

procedure PutObjSeq (ObjPtr: ObjHandle; Stuff: Ptr; NrBytes: Integer);
```

```
      (* PutObjSeq moves NrBytes bytes from the area pointed to by Stuff *)
      (* to ObjPtr. *)

    procedure PutObjByte (ObjPtr: ObjHandle; B: Byte);
      (* PutObjByte writes a byte to file ObjPtr. *)

    procedure PutObjWord (ObjPtr: ObjHandle; W: Integer);
      (* PutObjWord writes an integer to file ObjPtr. *)

    procedure PutObjLong (ObjPtr: ObjHandle; L: LongInt);
      (* PutObjLong writes a longint to file ObjPtr. *)

  implementation
 end.
```

Date: July 17, 1983
From: Ron Johnston
Subj: Format of .SYMBOLS files

The Lisa Assembler can produce a .SYMBOLS file that gives the mapping between symbol names and their locations within a code segment. The file format is very simple:

A .SYMBOLS file is a sequence of 12-byte records of the following structure:

```
+--------------------------------+----------------+
!  Symbol_Name (8 bytes)         !Location(4 byte)!
+--------------------------------+----------------+
```

Symbol_Name - left-adjusted, with names shorter than 8 characters padded on the right with blanks. They are case shifted, if necessary, to be all upper case.

Location - Gives the byte offset within the module from the beginning of code.

The symbol records are alphabetized within the file by Symbol_Name. The file is terminated by a record of all zeros (0). The remainder, if any, of the final block is also zeroed.

I have included a dump of the MONITOR.SYMBOLS file as an example.

```
File: monitor.symbols          Block #: 0

       0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 4130 544F 534F 2020 0000 3FEE 4143 4546 'AOTOSO  ..?.ACEF'
0010: 4C20 2020 0000 4D02 4143 544E 5442 4C20 'L     ..M.ACTNTBL '
0020: 0000 1D98 4144 4452 3253 4F20 0000 401A '....ADDR2SO ..@.'
0030: 4144 4452 4452 5652 0000 1CFA 414C 4C2E 'ADDRDRVR....ALL.'
0040: 3835 2020 0000 3BF8 414C 5241 4D48 2020 '85  ..;.ALRAMH  '
0050: 0000 430C 4150 4E44 4F42 4A20 0000 499E '..C.APNDOBJ ..I.'
0060: 4150 4E44 5458 5420 0000 48B0 4241 4442 'APNDTXT ..H.BADB'
0070: 4C43 4B20 0000 3B90 424C 4B49 4F20 2020 'LCK ..;.BLKIO   '
0080: 0000 3432 424C 4B52 5849 5420 0000 3428 '..42BLKRXIT ..4('
0090: 424C 4F43 4B38 3020 0000 3BA4 424C 4F43 'BLOCK80 ..;.BLOC'
00A0: 4B38 3120 0000 3C28 424C 4F43 4B38 3220 'K81 ..<(BLOCK82 '
00B0: 0000 3BA4 424C 4F43 4B38 3320 0000 3BA4 '..;.BLOCK83 ..;.'
00C0: 424C 4F43 4B38 3420 0000 3BA4 424C 4F43 'BLOCK84 ..;.BLOC'
00D0: 4B38 3520 0000 3BC0 424C 4F43 4B38 3620 'K85 ..;.BLOCK86 '
00E0: 0000 3C0A 424F 5453 5953 4320 0000 0030 '..<.BOTSYSC ...0'
00F0: 424F 5453 5953 4620 0000 0208 424F 5455 'BOTSYSF ....BOTU'
0100: 5442 4C20 0000 012E 4252 4549 4E49 3220 'TBL ....BREINI2 '
0110: 0000 0E98 4252 4549 4E49 5420 0000 0E4E '....BREINIT ...N'
0120: 4252 564F 4C53 5820 0000 2624 4253 5250 'BRVOLSX ..&$BSRP'
0130: 5554 4220 0000 3F90 4253 5350 4253 2020 'UTB ..?.BSSPBS  '
0140: 0000 46FE 4258 4C50 4C20 2020 0000 46CE '..F.BXLPL   ..F.'
0150: 4258 504C 2020 2020 0000 46D0 432E 5441 'BXPL    ..F.C.TA'
0160: 424C 4520 0000 3B96 4341 4C4C 4348 4B20 'BLE ..;.CALLCHK '
0170: 0000 1D34 4341 4C4C 4C44 5220 0000 4C4A '...4CALLLDR ..LJ'
0180: 4341 4C4C 4D41 4B45 0000 49FE 4341 4E54 'CALLMAKE..I.CANT'
0190: 4445 4620 0000 03A6 4341 4E54 5354 3120 'DEF ....CANTST1 '
01A0: 0000 3250 4341 4E54 5354 5220 0000 313C '..2PCANTSTR ..1<'
01B0: 4341 4E54 5354 5820 0000 3262 4344 4255 'CANTSTX ..2bCDBU'
01C0: 5359 2020 0000 1A10 4344 434C 4541 5220 'SY  ....CDCLEAR '
01D0: 0000 1A16 4344 4552 524F 5220 0000 1A04 '....CDERROR ....'
01E0: 4344 4556 564F 4C20 0000 280C 4344 5245 'CDEVVOL ..(.CDRE'
01F0: 4144 2020 0000 1A1A 4344 5257 5849 5420 'AD  ....CDRWXIT '
```

```
File: monitor.symbols          Block #: 1

       0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 0000 1AA0 4344 534B 4353 5A20 0000 1184 '....CDSKCSZ ....'
0010: 4344 534B 494E 4954 0000 115E 4344 534B 'CDSKINIT...↑CDSK'
0020: 5244 2020 0000 1276 4344 534B 5245 4144 'RD  ...vCDSKREAD'
0030: 0000 1174 4344 534B 5752 2020 0000 1308 '...tCDSKWR  ....'
0040: 4344 534B 5752 5420 0000 117C 4344 5752 'CDSKWRT ...|CDWR'
0050: 4954 4520 0000 1AD8 4348 4543 4B43 4420 'ITE ....CHECKCD '
0060: 0000 198C 4348 4B31 3038 2020 0000 3B32 '....CHK108  ..;2'
0070: 4348 4B42 4C4B 3720 0000 0DE6 4348 4B44 'CHKBLK7 ....CHKD'
0080: 4556 2020 0000 2628 4348 4B45 5252 2020 'EV  ..&(CHKERR  '
0090: 0000 0B70 4348 4B4D 5442 4C20 0000 0E50 '...pCHKMTBL ...P'
00A0: 434A 4D50 5442 4C20 0000 114E 434C 524C 'CJMPTBL ...NCLRL'
00B0: 4E20 2020 0000 4574 434C 524C 4E32 2020 'N   ..EtCLRLN2  '
00C0: 0000 4584 434C 524C 4F47 4E20 0000 440A '..E.CLRLOGN ..D.'
00D0: 434C 5253 4352 2020 0000 458C 434C 5253 'CLRSCR  ..E.CLRS'
00E0: 4352 3220 0000 459C 434D 444C 4F4F 5020 'CR2 ..E.CMDLOOP '
00F0: 0000 4B40 434D 5053 4658 2020 0000 21BC '..K@CMPSFX  ..!.'
0100: 434D 5053 4658 5820 0000 21C8 434F 5059 'CMPSFXX ..!.COPY'
0110: 3620 2020 0000 1468 434F 5059 364C 5020 '6   ...hCOPY6LP '
0120: 0000 1476 4350 5944 4556 4520 0000 0EBC '...vCPYDEVE ....'
0130: 4350 5944 4556 4E20 0000 0EB6 4350 594C 'CPYDEVN ....CPYL'
0140: 4F4F 5020 0000 07E4 4352 4C46 2020 2020 'OOP ....CRLF    '
0150: 0000 46C4 4353 4C41 5348 2020 0000 25AE '..F.CSLASH  ..%.'
0160: 4353 5A45 5849 5420 0000 116E 4445 4352 'CSZEXIT ...nDECR'
0170: 544F 5720 0000 0C84 4445 4C31 4348 2020 'TOW ....DEL1CH  '
0180: 0000 21A2 4445 4C44 4556 2020 0000 2642 '..!.DELDEV  ..&B'
0190: 4445 4C4E 5452 5920 0000 2A82 4445 564A 'DELNTRY ..*.DEVJ'
01A0: 4D50 5420 0000 06FA 4445 564C 4F4F 5020 'MPT ....DEVLOOP '
01B0: 0000 06F0 4445 564E 554D 3020 0000 071A '....DEVNUM0 ....'
01C0: 4445 564E 554D 3120 0000 072A 4445 564E 'DEVNUM1 ...*DEVN'
01D0: 554D 3220 0000 073A 4445 564E 554D 3320 'UM2 ...:DEVNUM3 '
01E0: 0000 0746 4445 564E 554D 3420 0000 0758 '...FDEVNUM4 ...X'
01F0: 4445 564E 554D 3520 0000 0772 4445 564E 'DEVNUM5 ...rDEVN'
```

```
File: monitor.symbols          Block #: 2

       0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 554D 3620 0000 078C 4445 564E 554D 3720 'UM6 ....DEVNUM7 '
0010: 0000 07C0 4445 564E 554D 4220 0000 265E '....DEVNUMB ..&↑'
0020: 4449 5253 5243 4820 0000 2876 4449 5253 'DIRSRCH ..(vDIRS'
0030: 5243 5820 0000 28E2 4449 5253 524C 5020 'RCX ..(.DIRSRLP '
0040: 0000 2892 4449 534B 4552 5220 0000 1124 '..(.DISKERR ...$'
0050: 4449 534B 494F 2020 0000 11AE 4449 534B 'DISKIO  ....DISK'
0060: 494F 5820 0000 126E 444A 4D50 5442 4C20 'IOX ...nDJMPTBL '
0070: 0000 43E8 444F 434F 5059 2020 0000 07E2 '..C.DOCOPY  ....'
0080: 444F 5346 5820 2020 0000 20DC 444F 5355 'DOSFX   .. .DOSU'
0090: 4649 5820 0000 20C6 444F 584F 5242 2020 'FIX .. .DOXORB  '
00A0: 0000 4BAC 4452 4956 4552 5320 0000 0B92 '..K.DRIVERS ....'
00B0: 4452 5652 2020 2020 0000 1C2C 4452 5652 'DRVR    ...,DRVR'
00C0: 5442 4C20 0000 1B28 4453 4B45 5252 2020 'TBL ...(DSKERR  '
00D0: 0000 15B6 4453 4B52 4431 2020 0000 12AC '....DSKRD1  ....'
00E0: 4453 4B52 4432 2020 0000 12BC 4453 4B52 'DSKRD2  ....DSKR'
00F0: 4433 2020 0000 12C0 4453 4B52 4434 2020 'D3  ....DSKRD4  '
0100: 0000 12D0 4453 4B52 4435 2020 0000 12E6 '....DSKRD5  ....'
0110: 4453 4B52 4436 2020 0000 12F0 4453 4B52 'DSKRD6  ....DSKR'
0120: 4541 4420 0000 1118 4453 4B57 5254 2020 'EAD ....DSKWRT  '
0130: 0000 1130 4453 4B57 5254 3220 0000 133E '...0DSKWRT2 ...>'
0140: 4453 4B57 5254 3320 0000 134C 4453 4B57 'DSKWRT3 ...LDSKW'
0150: 5254 3420 0000 135C 4453 4B57 5254 3520 'RT4 ...\DSKWRT5 '
0160: 0000 1376 4453 4B57 5254 3620 0000 137A '...vDSKWRT6 ...z'
0170: 4453 4B57 5254 3720 0000 138A 454C 504C 'DSKWRT7 ....ELPL'
0180: 2020 2020 0000 0A7A 454E 4457 4838 3520 '    ...zENDWH85 '
0190: 0000 3C06 454E 5452 544D 5020 0000 2AD8 '..<.ENTRTMP ..*.'
01A0: 454E 554D 4253 5020 0000 44E4 4550 4C20 'ENUMBSP ..D.EPL '
01B0: 2020 2020 0000 0A7C 4552 524F 5220 2020 '    ...|ERROR   '
01C0: 0000 3D8C 4558 4543 4452 5652 0000 1956 '..=.EXECDRVR...V'
01D0: 4641 4952 5454 4C20 0000 1F70 4641 4B45 'FAIRTTL ...pFAKE'
01E0: 5245 4420 0000 4CEC 4643 4C4F 5345 2020 'RED ..L.FCLOSE  '
01F0: 0000 2EEA 4643 4C4F 5345 3120 0000 311A '....FCLOSE1 ..1.'
```

```
File: monitor.symbols          Block #: 3

       0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
```

```
0000: 4643 4C4F 5345 5820 0000 312A 4647 4554 'FCLOSEX ..1*FGET'
0010: 2020 2020 0000 35B8 4647 4554 314C 2020 '    ..5.FGET1L  '
0020: 0000 37DC 4647 4554 324C 2020 0000 37EC '..7.FGET2L  ..7.'
0030: 4647 4554 324E 4420 0000 3690 4647 4554 'FGET2ND ..6.FGET'
0040: 3352 4420 0000 373E 4647 4554 4E4F 5020 '3RD ..7>FGETNOP '
0050: 0000 37D4 4649 4C31 5354 5220 0000 47A0 '..7.FIL1STR ..G.'
0060: 4649 4C32 5354 5220 0000 0598 4649 4C4C 'FIL2STR ....FILL'
0070: 4255 4620 0000 3CA8 4649 4E44 4250 2020 'BUF ..<.FINDBP  '
0080: 0000 3E90 4649 4E44 4432 2020 0000 141A '..>.FINDD2  ....'
0090: 4649 4E44 4C50 2020 0000 4000 4649 4E44 'FINDLP  ..@.FIND'
00A0: 4D41 5820 0000 2AB8 4649 4E44 5359 5320 'MAX ..*.FINDSYS '
00B0: 0000 4DFA 4649 4E49 5348 2020 0000 15BE '..M.FINISH  ....'
00C0: 4649 4E49 5420 2020 0000 1EB4 4649 4E49 'FINIT   ....FINI'
00D0: 5458 2020 0000 1F04 4649 5846 5649 4420 'TX  ....FIXFVID '
00E0: 0000 27EE 464C 504C 2020 2020 0000 0A6A '..'.FLPL    ...j'
00F0: 464E 4453 4547 4E20 0000 3E9C 464E 4453 'FNDSEGN ..>.FNDS'
0100: 4547 4F20 0000 3EA4 464E 4453 5953 3220 'EGO ..>.FNDSYS2 '
0110: 0000 4DE6 464E 4453 5953 3320 0000 4DF8 '..M.FNDSYS3 ..M.'
0120: 464F 5045 4E20 2020 0000 2C36 464F 5045 'FOPEN   ..,6FOPE'
0130: 4E31 2020 0000 2EBE 464F 5045 4E32 2020 'N1  ....FOPEN2  '
0140: 0000 2E2E 464F 5045 4E4F 4B20 0000 4C94 '....FOPENOK ..L.'
0150: 464F 5045 4E58 2020 0000 2ED6 4650 4C20 'FOPENX  ....FPL '
0160: 2020 2020 0000 0A6C 4650 5554 2020 2020 '    ...lFPUT    '
0170: 0000 37F4 4650 5554 314C 2020 0000 3960 '..7.FPUT1L  ..9`'
0180: 4650 5554 324C 2020 0000 3970 4650 5554 'FPUT2L  ..9pFPUT'
0190: 4E4F 5020 0000 3958 4652 4541 4443 4852 'NOP ..9XFREADCHR'
01A0: 0000 34F2 4652 4541 444C 4E20 0000 3528 '..4.FREADLN ..5('
01B0: 4652 4545 5031 3230 0000 220C 4652 4545 'FREEP120..".FREE'
01C0: 5031 3234 0000 2210 4652 4545 5031 3336 'P124..".FREEP136'
01D0: 0000 2214 4652 4545 5031 3430 0000 2218 '..".FREEP140..".'
01E0: 4652 4553 4554 2020 0000 28EC 4652 4553 'FRESET  ..(.FRES'
01F0: 4554 5820 0000 2924 4652 4F4D 4558 4543 'ETX ..)$FROMEXEC'

File: monitor.symbols          Block #: 4

      0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 0000 47F6 4653 4545 4B20 2020 0000 398E '..G.FSEEK   ..9.'
0010: 4653 4545 4B31 2020 0000 3A9E 4654 4348 'FSEEK1  ..:.FTCH'
0020: 4449 5220 0000 234A 4654 4348 4452 3320 'DIR ..#JFTCHDR3 '
0030: 0000 251A 4654 4348 4452 3420 0000 2528 '..%.FTCHDR4 ..%('
0040: 4654 4348 4452 5820 0000 2566 4654 4348 'FTCHDRX ..%fFTCH'
0050: 4552 5220 0000 2536 4657 5249 5445 4C4E 'ERR ..%6FWRITELN'
0060: 0000 34D8 4657 5254 4348 4152 0000 3498 '..4.FWRTCHAR..4.'
0070: 4657 5254 4348 5820 0000 34D2 4745 5420 'FWRTCHX ..4.GET '
0080: 2020 2020 0000 1B4C 4745 5442 4153 4520 '    ...LGETBASE '
0090: 0000 0C5E 4745 5443 4841 5220 0000 4686 '...^GETCHAR ..F.'
00A0: 4745 5443 4852 5820 0000 46C2 4745 5445 'GETCHRX ..F.GETE'
00B0: 4649 4220 0000 477E 4745 5449 4E44 5820 'FIB ..G~GETINDX '
00C0: 0000 09E8 4745 5449 4E46 4F20 0000 19A0 '....GETINFO ....'
00D0: 4745 544A 5442 4C20 0000 10F2 4745 544D 'GETJTBL ....GETM'
00E0: 5442 4C20 0000 0C42 4745 5452 4449 5220 'TBL ...BGETRDIR '
00F0: 0000 0FBE 4745 5452 4547 5320 0000 4420 '....GETREGS ..D '
0100: 4745 5452 534C 5420 0000 1B68 4745 5453 'GETRSLT ...hGETS'
0110: 5441 5420 0000 14A8 4745 5455 4E49 5420 'TAT ....GETUNIT '
0120: 0000 1C4C 474F 3255 5345 5220 0000 3C7E '...LGO2USER ..<~'
0130: 474F 4F44 5043 2020 0000 3E6C 4752 4F57 'GOODPC  ..>lGROW'
0140: 3220 2020 0000 2326 4752 4F57 4449 5220 '2   ..#&GROWDIR '
0150: 0000 2282 4841 4E44 4C45 5220 0000 09D6 '..".HANDLER ....'
0160: 4844 524C 4F4F 5020 0000 1628 4844 534B 'HDRLOOP ...(HDSK'
0170: 4353 5A20 0000 113E 4844 534B 494E 4954 'CSZ ...>HDSKINIT'
0180: 0000 1108 484F 4D43 5253 5220 0000 4568 '....HOMCRSR ..Eh'
0190: 494C 504C 2020 2020 0000 447E 494E 4458 'ILPL    ..D~INDX'
01A0: 4552 3020 0000 0ACE 494E 4458 4552 5220 'ER0 ....INDXERR '
01B0: 0000 0AD0 494E 4954 4445 5620 0000 06D0 '....INITDEV ....'
01C0: 494E 4954 464C 5320 0000 05A2 494E 4954 'INITFLS ....INIT'
01D0: 4941 4C20 0000 020A 494E 4954 4D49 2020 'IAL ....INITMI  '
01E0: 0000 0618 494E 4954 4D49 4C20 0000 0672 '....INITMIL ...r'
01F0: 494E 4954 4D49 5820 0000 0678 494E 4954 'INITMIX ...xINIT'

File: monitor.symbols          Block #: 5

      0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 5052 4720 0000 43F8 494E 4954 5359 5320 'PRG ..C.INITSYS '
0010: 0000 023A 494E 4954 5359 5346 0000 051A '...:INITSYSF....'
0020: 494E 4954 5554 424C 0000 0508 494E 4954 'INITUTBL....INIT'
0030: 5849 5420 0000 116C 494E 534E 5452 5920 'XIT ...lINSNTRY '
0040: 0000 2A2E 494E 5353 5441 5220 0000 49BE '..*.INSSTAR ..I.'
0050: 494E 5452 4C56 2020 0000 13FC 494F 4348 'INTRLV  ....IOCH'
0060: 4B20 2020 0000 44A6 494F 4348 4B58 2020 'K   ..D.IOCHKX  '
0070: 0000 4512 494F 4558 4954 2020 0000 1CDE '..E.IOEXIT  ....'
0080: 4950 4C20 2020 2020 0000 4480 4954 2E46 'IPL     ..D.IT.F'
0090: 4954 5320 0000 3B44 4A45 5252 4F52 3120 'ITS ..;DJERROR1 '
00A0: 0000 3B3E 4A49 4E44 5845 5252 0000 0A52 '..;>JINDXERR...R'
00B0: 4A4D 5054 424C 2020 0000 091E 4B46 4C55 'JMPTBL  ....KFLU'
00C0: 5348 2020 0000 1CA4 4B49 4C4C 4355 5220 'SH  ....KILLCUR '
00D0: 0000 4B12 4C30 2E38 3520 2020 0000 3BE4 '..K.L0.85   ..;.'
00E0: 4C31 2E38 3520 2020 0000 3C00 4C41 5354 'L1.85   ..<.LAST'
00F0: 5345 4720 0000 3AF0 4C41 554E 4348 2020 'SEG ..:.LAUNCH  '
0100: 0000 40A0 4C44 5041 524D 5320 0000 3974 '..@.LDPARMS ..9t'
0110: 4C44 534B 5244 2020 0000 1514 4C44 534B 'LDSKRD  ....LDSK'
0120: 5752 2020 0000 166E 4C45 4C50 4C20 2020 'WR  ...nLELPL   '
0130: 0000 3D7C 4C45 504C 2020 2020 0000 3D7E '..=|LEPL    ..=~'
0140: 4C4E 4656 4944 3020 0000 2742 4C4F 4144 'LNFVID0 ..'BLOAD'
0150: 4F42 4A20 0000 49DE 4C4F 474E 3244 3520 'OBJ ..I.LOGN2D5 '
0160: 0000 2258 4C4F 4F50 454E 4420 0000 0E9C '.."XLOOPEND ....'
0170: 4C4F 4F50 494F 2020 0000 3274 4C4F 4F50 'LOOPIO  ..2tLOOP'
0180: 5442 4C20 0000 0E68 4C4F 4F50 564F 4C20 'TBL ...hLOOPVOL '
0190: 0000 0E08 4C50 4C20 2020 2020 0000 45FE '....LPL     ..E.'
01A0: 4C50 4C32 2020 2020 0000 45AC 4C53 4C41 'LPL2    ..E.LSLA'
01B0: 5348 2020 0000 259C 4D41 494E 4C4F 4F50 'SH  ..%.MAINLOOP'
01C0: 0000 3B6C 4D41 4B45 544D 5020 0000 1E8E '..;lMAKETMP ....'
01D0: 4D44 534B 4353 5A20 0000 1738 4D44 534B 'MDSKCSZ ...8MDSK'
01E0: 494E 4954 0000 1716 4D44 534B 5244 2020 'INIT....MDSKRD  '
01F0: 0000 1742 4D44 534B 5245 4144 0000 1728 '...BMDSKREAD...('

File: monitor.symbols          Block #: 6

      0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 4D44 534B 5245 5320 0000 16FA 4D44 534B 'MDSKRES ....MDSK'
0010: 5752 2020 0000 184C 4D44 534B 5752 5420 'WR  ...LMDSKWRT '
0020: 0000 1730 4D45 4D41 2020 2020 0000 1D5A '...0MEMA    ...Z'
```

```
0030: 4D45 4D52 4541 4420 0000 1AA4 4D45 4D57 'MEMREAD ....MEMW'
0040: 5249 5445 0000 1AD2 4D47 4F54 4F58 5920 'RITE....MGOTOXY '
0050: 0000 0BE4 4D48 414C 5420 2020 0000 0A08 '....MHALT   ....'
0060: 4D49 4F45 5252 2020 0000 0B8E 4D4A 4D50 'MIOERR  ....MJMP'
0070: 5442 4C20 0000 16EA 4D4D 524B 2020 2020 'TBL ....MMRK    '
0080: 0000 1D26 4D4E 4557 2020 2020 0000 1D12 '...&MNEW    ....'
0090: 4D4F 4E42 4153 4520 0000 0000 4D4F 5645 'MONBASE ....MOVE'
00A0: 3050 2020 0000 04F6 4D4F 5645 4D45 4D20 '0P  ....MOVEMEM '
00B0: 0000 1AA8 4D4F 5646 4153 5420 0000 1ABC '....MOVFAST ....'
00C0: 4D4F 5653 4C4F 5720 0000 1AAE 4D52 4C53 'MOVSLOW ....MRLS'
00D0: 2020 2020 0000 1D2E 4D56 324B 4244 2020 '    ....MV2KBD  '
00E0: 0000 47AA 4D56 4D54 4142 4C20 0000 0F7A '..G.MVMTABL ...z'
00F0: 4E44 534B 5244 2020 0000 1512 4E44 534B 'NDSKRD  ....NDSK'
0100: 5245 4144 0000 13E2 4E44 534B 5752 2020 'READ....NDSKWR  '
0110: 0000 166C 4E44 534B 5752 5420 0000 13EA '...lNDSKWRT ....'
0120: 4E45 5754 4D50 3120 0000 0C7A 4E45 5854 'NEWTMP1 ...zNEXT'
0130: 2E31 2020 0000 3CF6 4E45 5854 2E33 2020 '.1  ..<.NEXT.3  '
0140: 0000 3D06 4E45 5854 2E34 2020 0000 3D18 '..=.NEXT.4  ..=.'
0150: 4E45 5854 4255 4620 0000 3CEA 4E45 5854 'NEXTBUF ..<.NEXT'
0160: 5345 4720 0000 3FF2 4E4A 4D50 5442 4C20 'SEG ..?.NJMPTBL '
0170: 0000 13AE 4E4C 4F41 4445 5220 0000 032C '....NLOADER ...,'
0180: 4E4F 4150 504C 3220 0000 1B5C 4E4F 4150 'NOAPPL2 ...\NOAP'
0190: 504C 4520 0000 1B56 4E4F 4341 5244 2020 'PLE ...VNOCARD  '
01A0: 0000 0FB4 4E4F 4C4F 4144 5220 0000 4C90 '....NOLOADR ..L.'
01B0: 4E4F 4E55 4D42 3420 0000 033E 4E4F 5434 'NONUMB4 ...>NOT4'
01C0: 3420 2020 0000 3F98 4E4F 5436 3048 5A20 '4   ..?.NOT60HZ '
01D0: 0000 0C92 4E4F 5441 4352 2020 0000 4B7E '....NOTACR  ..K~'
01E0: 4E4F 5442 4143 4B20 0000 4BA4 4E4F 5444 'NOTBACK ..K.NOTD'
01F0: 4953 4B20 0000 19F6 4E4F 544F 4B4C 4820 'ISK ....NOTOKLH '
```

File: monitor.symbols Block #: 7

```
      0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 0000 277E 4E4F 5450 4D47 5220 0000 4B96 '..'~NOTPMGR ..K.'
0010: 4E4F 5453 5441 5220 0000 1F96 4E4F 5457 'NOTSTAR ....NOTW'
0020: 4744 5220 0000 0546 4E58 5450 524F 4320 'GDR ...FNXTPROC '
0030: 0000 42E0 4F46 534D 4953 4320 0000 000A '..B.OFSMISC ....'
0040: 4F4B 2E31 2020 2020 0000 3CFC 4F50 4E43 'OK.1    ..<.OPNC'
0050: 4F44 4520 0000 497C 4F50 4E45 5845 4320 'ODE ..I|OPNEXEC '
0060: 0000 488E 4F56 4643 484B 2020 0000 0A3A '..H.OVFCHK  ...:'
0070: 4F56 4643 484B 5820 0000 0A54 502E 4C4F 'OVFCHKX ...TP.LO'
0080: 4F50 2020 0000 3C3A 5041 5443 4842 5020 'OP  ..<:PATCHBP '
0090: 0000 3C64 5041 5443 4849 5420 0000 3C2C '..<dPATCHIT ..<,'
00A0: 5044 534B 4353 5A20 0000 13F2 5044 534B 'PDSKCSZ ....PDSK'
00B0: 494E 4954 0000 13BE 5044 534B 5244 2020 'INIT....PDSKRD  '
00C0: 0000 150C 5044 534B 5245 4144 0000 13D2 '....PDSKREAD....'
00D0: 5044 534B 5752 2020 0000 1666 5044 534B 'PDSKWR  ...fPDSK'
00E0: 5752 5420 0000 13DA 504A 4D50 5442 4C20 'WRT ....PJMPTBL '
00F0: 0000 139E 504C 2020 2020 2020 0000 4600 '....PL      ..F.'
0100: 504C 3220 2020 2020 0000 45AE 504D 4144 'PL2     ..E.PMAD'
0110: 4452 3253 0000 405C 504F 4B45 5843 5020 'DR2S..@\POKEXCP '
0120: 0000 416C 5052 494E 5420 2020 0000 4674 '..AlPRINT   ..Ft'
0130: 5052 4E54 4552 5220 0000 44D0 5052 4F43 'PRNTERR ..D.PROC'
0140: 4E55 4D20 0000 0A8C 5052 4F4D 5054 2020 'NUM ....PROMPT  '
0150: 0000 4650 5052 5442 5553 5920 0000 1DA0 '..FPPRTBUSY ....'
0160: 5052 5443 4C52 2020 0000 1DA6 5052 5444 'PRTCLR  ....PRTD'
0170: 5256 5220 0000 1D90 5052 5445 5849 5420 'RVR ....PRTEXIT '
0180: 0000 1E4C 5052 5449 4E49 5420 0000 118C '...LPRTINIT ....'
0190: 5052 5452 4541 4420 0000 1DA2 5052 5453 'PRTREAD ....PRTS'
01A0: 454E 4420 0000 1E50 5052 5453 4554 5550 'END ...PPRTSETUP'
01B0: 0000 1DBC 5052 5457 5254 2020 0000 1DEE '....PRTWRT  ....'
01C0: 5055 5420 2020 2020 0000 1B3C 5055 5442 'PUT     ...<PUTB'
01D0: 4143 4B20 0000 3FA6 5055 5442 414B 3220 'ACK ..?.PUTBAK2 '
01E0: 0000 3FCA 5055 5442 414B 3320 0000 3FE8 '..?.PUTBAK3 ..?.'
01F0: 5055 5450 5246 3120 0000 4554 5055 5450 'PUTPRF1 ..ETPUTP'
```

File: monitor.symbols Block #: 8

```
      0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 5246 3220 0000 4566 5055 5450 5246 5820 'RF2 ..EfPUTPRFX '
0010: 0000 4532 5155 4954 4453 4B20 0000 0C96 '..E2QUITDSK ....'
0020: 5243 4552 5220 2020 0000 0ADE 5244 4441 'RCERR   ....RDDA'
0030: 5441 2020 0000 1BD2 5244 4D54 4142 4C20 'TA  ....RDMTABL '
0040: 0000 0CEA 5244 4D54 494E 4954 0000 0D5A '....RDMTINIT...Z'
0050: 5244 4D54 4C50 2020 0000 0D60 5244 4E52 'RDMTLP  ...`RDNR'
0060: 4553 2020 0000 155E 5245 4144 2E4F 4B20 'ES  ...^READ.OK '
0070: 0000 3CE8 5245 4144 4844 5220 0000 157C '..<.READHDR ...|'
0080: 5245 4144 4C50 2020 0000 1594 5245 4752 'READLP  ....REGR'
0090: 4553 5420 0000 437A 5245 494E 4954 2020 'EST ..CzREINIT  '
00A0: 0000 0F4A 5245 494E 4954 4A20 0000 0F5E '...JREINITJ ...^'
00B0: 5245 4D41 5020 2020 0000 140C 5245 4D4F 'REMAP   ....REMO'
00C0: 5645 3120 0000 3D2A 5245 5345 5445 5220 'VE1 ..=*RESETER '
00D0: 0000 292C 5245 5345 5452 5820 0000 2A26 '..),RESETRX ..*&'
00E0: 5245 5354 4152 5420 0000 4B24 5245 5649 'RESTART ..K$REVI'
00F0: 5349 4F4E 0000 45A4 524C 504C 2020 2020 'SION..E.RLPL    '
0100: 0000 4A5A 524C 504C 3220 2020 0000 4A66 '..JZRLPL2   ..Jf'
0110: 524E 4E47 5553 5220 0000 3EE8 5250 4C20 'RNNGUSR ..>.RPL '
0120: 2020 2020 0000 4A5C 5250 4C32 2020 2020 '    ..J\RPL2    '
0130: 0000 4A68 5253 4B50 4844 5220 0000 158C '..JhRSKPHDR ....'
0140: 5253 5452 5843 5020 0000 41D2 5254 4E46 'RSTRXCP ..A.RTNF'
0150: 4C44 5220 0000 4C8C 5341 4D45 5345 4720 'LDR ..L.SAMESEG '
0160: 0000 3C70 5343 414E 4148 4420 0000 3552 '..<pSCANAHD ..5R'
0170: 5343 414E 5454 4C20 0000 1F06 5343 414E 'SCANTTL ....SCAN'
0180: 5454 4C58 0000 2174 5343 4552 5220 2020 'TTLX..!tSCERR   '
0190: 0000 0A24 5343 4845 4455 4C20 0000 41FC '...$SCHEDUL ..A.'
01A0: 5343 4E53 5452 4320 0000 1F40 5345 4E44 'SCNSTRC ...@SEND'
01B0: 434D 4420 0000 1B70 5345 4E44 4844 5220 'CMD ...pSENDHDR '
01C0: 0000 1B78 5345 5441 3141 3220 0000 2A70 '...xSETA1A2 ..*p'
01D0: 5345 5444 4953 4B20 0000 10E0 5345 5445 'SETDISK ....SETE'
01E0: 4E49 5620 0000 4388 5345 5448 4453 4B20 'NIV ..C.SETHDSK '
01F0: 0000 10A4 5345 5453 5220 2020 0000 4ED6 '....SETSR   ..N.'
```

File: monitor.symbols Block #: 9

```
      0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 5345 5454 5747 5920 0000 0ED2 5345 5455 'SETTWGY ....SETU'
0010: 5041 3520 0000 441A 5346 5842 4143 4B20 'PA5 ..D.SFXBACK '
0020: 0000 218A 5346 5843 4F44 4520 0000 2184 '..!.SFXCODE ..!.'
0030: 5346 5846 4F54 4F20 0000 219C 5346 5847 'SFXFOTO ..!.SFXG'
0040: 5241 4620 0000 2196 5346 5849 4E46 4F20 'RAF ..!.SFXINFO '
0050: 0000 2190 5346 5854 4558 5420 0000 217E '..!.SFXTEXT ..!~'
```

```
0060: 534B 5053 594E 4320 0000 04CE 534D 5343 'SKPSYNC ....SMSC'
0070: 4E46 4F20 0000 060A 534E 4452 3120 2020 'NFO ....SNDR1   '
0080: 0000 1436 534F 4654 4230 2020 0000 3E32 '...6SOFTB0  ..>2'
0090: 534F 4654 4250 5420 0000 3E2E 534F 4654 'SOFTBPT ..>.SOFT'
00A0: 5849 5420 0000 3EAE 5354 4152 5455 5020 'XIT ..>.STARTUP '
00B0: 0000 084E 5354 4154 3031 2020 0000 1456 '...NSTAT01  ...V'
00C0: 5354 4154 4A4D 5020 0000 10C8 5354 4154 'STATJMP ....STAT'
00D0: 5553 5220 0000 43B4 5354 494C 4C49 4E20 'USR ..C.STILLIN '
00E0: 0000 3D5C 5354 5254 4F42 4A20 0000 4A6E '..=\STRTOBJ ..Jn'
00F0: 5354 5254 5244 2020 0000 14C2 5354 5254 'STRTRD  ....STRT'
0100: 5752 5420 0000 15C8 5355 4D44 4952 2020 'WRT ....SUMDIR  '
0110: 0000 22FA 5355 4D44 4952 4C20 0000 2316 '..".SUMDIRL ..#.'
0120: 5355 4D44 4952 5820 0000 2320 5357 4150 'SUMDIRX ..# SWAP'
0130: 4A54 2020 0000 41E6 5359 5350 524F 4720 'JT  ..A.SYSPROG '
0140: 0000 4D6C 5448 4545 4E44 2020 0000 4F22 '..MlTHEEND  ..O"'
0150: 544C 504C 2020 2020 0000 448A 544F 502E 'TLPL    ..D.TOP.'
0160: 3836 2020 0000 3C16 544F 5053 5953 4320 '86  ..<.TOPSYSC '
0170: 0000 0004 544F 5053 5953 4620 0000 0130 '....TOPSYSF ...0'
0180: 544F 5055 5442 4C20 0000 0032 5450 4C20 'TOPUTBL ...2TPL '
0190: 2020 2020 0000 448C 5452 5032 4D41 5820 '    ..D.TRP2MAX '
01A0: 0000 4EDC 5452 5032 4D43 5320 0000 4EE8 '..N.TRP2MCS ..N.'
01B0: 5452 5943 5256 2020 0000 10DC 5452 5944 'TRYCRV  ....TRYD'
01C0: 4255 4720 0000 4CC6 5452 5945 5845 4320 'BUG ..L.TRYEXEC '
01D0: 0000 48D4 5452 594D 524B 2020 0000 10D0 '..H.TRYMRK  ....'
01E0: 5452 594E 4558 5420 0000 3ADE 5452 594E 'TRYNEXT ..:.TRYN'
01F0: 4A4D 5020 0000 0F3E 5452 5950 4950 2020 'JMP ...>TRYPIP  '

File:  monitor.symbols        Block #:  10

      0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 0000 10D6 5452 5951 2020 2020 0000 4CF4 '....TRYQ    ..L.'
0010: 5452 5958 4551 5420 0000 4BCC 5453 5446 'TRYXEQT ..K.TSTF'
0020: 5649 4420 0000 1FFA 5453 5453 5441 5420 'VID ....TSTSTAT '
0030: 0000 10BC 5454 4C43 4F50 5920 0000 21CC '....TTLCOPY ..!.'
0040: 5542 5553 5920 2020 0000 1CE6 5543 4C52 'UBUSY   ....UCLR'
0050: 2020 2020 0000 1CBC 5549 4F20 2020 2020 '    ....UIO     '
0060: 0000 1CCE 554E 2E4C 4F4F 5020 0000 3D68 '....UN.LOOP ..=h'
0070: 554E 4954 424C 2020 0000 08A0 554E 4954 'UNITBL  ....UNIT'
0080: 4253 5920 0000 1BA2 554E 4954 434C 5220 'BSY ....UNITCLR '
0090: 0000 1BBE 554E 4954 4445 5620 0000 196A '....UNITDEV ...j'
00A0: 554E 4954 4953 4220 0000 221C 554E 4D41 'UNITISB ..".UNMA'
00B0: 5020 2020 0000 226A 5550 5348 4654 2020 'P   .."jUPSHFT  '
00C0: 0000 4516 5550 5348 4654 5820 0000 4526 '..E.UPSHFTX ..E&'
00D0: 5552 4541 4420 2020 0000 1CCC 5553 454C 'UREAD   ....USEL'
00E0: 564C 3720 0000 4F18 5554 3137 4944 5820 'VL7 ..O.UT17IDX '
00F0: 0000 2242 5557 5249 5445 2020 0000 1CC6 '.."BUWRITE  ....'
0100: 5631 3020 2020 2020 0000 3E1C 5631 3120 'V10     ..>.V11 '
0110: 2020 2020 0000 3E22 5632 2020 2020 2020 '    ..>"V2      '
0120: 0000 3DEC 5632 3420 2020 2020 0000 3E28 '..=.V24     ..>('
0130: 5633 2020 2020 2020 0000 3DF2 5633 3120 'V3      ..=.V31 '
0140: 2020 2020 0000 3DB2 5633 3141 2020 2020 '    ..=.V31A    '
0150: 0000 3DAE 5633 3158 2020 2020 0000 3DBE '..=.V31X    ..=.'
0160: 5634 2020 2020 2020 0000 3DF8 5635 2020 'V4      ..=.V5  '
0170: 2020 2020 0000 3DFE 5636 2020 2020 2020 '    ..=.V6      '
0180: 0000 3E04 5637 2020 2020 2020 0000 3E0A '..>.V7      ..>.'
0190: 5638 2020 2020 2020 0000 3E10 5639 2020 'V8      ..>.V9  '
01A0: 2020 2020 0000 3E16 5641 4C49 4442 3720 '    ..>.VALIDB7 '
01B0: 0000 0EA4 564F 4C53 5243 4820 0000 256E '....VOLSRCH ..%n'
01C0: 564F 4C53 5243 5820 0000 2862 564F 4C53 'VOLSRCX ..(bVOLS'
01D0: 5258 5420 0000 286C 5731 4C50 4C20 2020 'RXT ..(lW1LPL   '
01E0: 0000 080C 5731 504C 2020 2020 0000 080E '....W1PL    ....'
01F0: 5732 4C50 4C20 2020 0000 0816 5732 504C 'W2LPL   ....W2PL'

File:  monitor.symbols        Block #:  11

      0    2    4    6    8    A    C    E     0 2 4 6 8 A C E
0000: 2020 2020 0000 0818 5741 4954 2020 2020 '    ....WAIT    '
0010: 0000 4528 5743 524C 4620 2020 0000 46C6 '..E(WCRLF   ..F.'
0020: 5745 4C43 4F4D 3220 0000 086A 5745 4C43 'WELCOM2 ...jWELC'
0030: 4F4D 4520 0000 085A 5746 4231 2020 2020 'OME ...ZWFB1    '
0040: 0000 1422 5746 4E42 3120 2020 0000 1446 '..."WFNB1   ...F'
0050: 5748 494C 4538 3520 0000 3BD0 574C 504C 'WHILE85 ..;.WLPL'
0060: 2020 2020 0000 0A5A 574C 504C 3220 2020 '    ...ZWLPL2   '
0070: 0000 0A64 5750 4C20 2020 2020 0000 0A5C '...dWPL     ...\'
0080: 5750 4C32 2020 2020 0000 0A66 5752 4441 'WPL2    ...fWRDA'
0090: 5441 2020 0000 1BFA 5752 4954 4549 5420 'TA  ....WRITEIT '
00A0: 0000 2BF2 5752 5444 4952 2020 0000 2C00 '..+.WRTDIR  ....'
00B0: 5752 544C 4F4F 5020 0000 1644 5752 544E 'WRTLOOP ...DWRTN'
00C0: 5245 5320 0000 16DC 5752 5450 5254 2020 'RES ....WRTPRT  '
00D0: 0000 1A0A 5753 4B50 4844 5220 0000 163C '....WSKPHDR ...<'
00E0: 5842 4C4B 494F 2020 0000 3314 5845 5155 'XBLKIO  ..3.XEQU'
00F0: 5445 2020 0000 4702 5845 5155 5445 3020 'TE  ..G.XEQUTE0 '
0100: 0000 4708 5845 5155 5445 3120 0000 4734 '..G.XEQUTE1 ..G4'
0110: 5845 5155 5445 3220 0000 4770 5845 5155 'XEQUTE2 ..GpXEQU'
0120: 5445 3320 0000 477C 5845 5155 5445 3420 'TE3 ..G|XEQUTE4 '
0130: 0000 475C 5849 544E 5452 5920 0000 2A66 '..G\XITNTRY ..*f'
0140: 584C 504C 2020 2020 0000 46F0 5850 4C20 'XLPL    ..F.XPL '
0150: 2020 2020 0000 46F2 5853 5452 5455 5020 '    ..F.XSTRTUP '
0160: 0000 0828 592E 4552 524F 5220 0000 2208 '...(Y.ERROR ..".'
0170: 592E 4C45 4156 4520 0000 21F8 592E 4C4F 'Y.LEAVE ..!.Y.LO'
0180: 4F50 2020 0000 21F2 592E 5445 5354 2020 'OP  ..!.Y.TEST  '
0190: 0000 21F4 5A45 524F 4D45 4D20 0000 067C '..!.ZEROMEM ...|'
01A0: 5A5A 494F 5245 5320 0000 1D04 5A5A 4C4F 'ZZIORES ....ZZLO'
01B0: 4144 4954 0000 3AA6 5A5A 554E 4C4F 4144 'ADIT..:.ZZUNLOAD'
01C0: 0000 3D24 0000 0000 0000 0000 0000 0000 '..=$............'
01D0: 0000 0000 0000 0000 0000 0000 0000 0000 '................'
01E0: 0000 0000 0000 0000 0000 0000 0000 0000 '................'
01F0: 0000 0000 0000 0000 0000 0000 0000 0000 '................'
```

Using LisaBug

Ed Birss

What to do when you crash, hang, or loop

When a program crashes in the Office System, and the release has LisaBug, you end up in LisaBug. You can then poke around for a while, but eventually you will want to get on to other things. To get out of LisaBug, you need to know a few things. The register display, on the right of the third line has a piece that says DO=0 (or 1, 2, or 3). The DO stands for domain, and if the domain is nonzero and it does not say overridden to 0, then to resume you should type the LisaBug command G. This is the typical crash found in the Office System, and using the G command forces the process into the terminate exception handler, and things can be put away neatly. If you are in domain 0, or overridden to zero, you should use the OSQUIT command.

If you are stuck and nothing is happening in response to power offs, key input or mouse clicks, you are either looping or are hung. In either case you want to hit NMI. If the display is not in domain 0, you are probably looping. To kill the process, you can type G 0, or PC 0 followed by G. This sets the program counter to 0 and tries to access location 0 which is illegal and causes a bus error. Typing G after this bus error will terminate the process neatly.

If you are in domain 0 and you are sitting on an RTS instruction, type id PC-4. If the result is a STOP instruction, then you may be hung. You should first make sure that you are not doing any I/O. Type G to continue and watch the ProFile lights and listen for diskette I/O. If I/O is in progress, you can wait for the I/O to complete, or you can follow instructions on looping which follow. If however, no I/O is in progress, and when you hit NMI you are still on and RTS instruction and the STOP instruction preceed the RTS, type OSQUIT to clean up the OS and file structures.

If you are in domain 0 and are not on an RTS instruction, you should type G and then NMI again. Eventually you should get out of domain 0 or get to the STOP instruction. You can also use the UBR command as described in the breakpoints section. If you cannot get out of domain 0, Type OSQUIT to clean up.

The ground rules are do everything you can to terminate processes normally. If you blow up in an application, type G to terminate cleanly. After looping, type PC 0 ; G to again terminate the process cleanly. Use OSQUIT as a last resort, and that means only in domain 0. You should never have to reset the machine using the reset button on the back of the machine.

The PU/PL dump

Frequently, a bug report will come with a three page printout that was made with the PU or PL LisaBug command. This command generates output similar to pages 1, 2, and 3. The first page consists of a screen dump of the primary screen, the second page contains the screen dump of the alternate screen, and the third page has some additional stack crawl and memory locations displayed.

There is a wealth of information provided in these three pages. The first page gives us a big hint; some item from the arrangement menu was being executed. The second page gives us additional information. A bus error was detected and the access address is 0. This is a big clue because nil pointers are 0 and generate a bus error if you try to access location 0. Also included on page 2 is a register display, and the most interesting piece of information is that the program counter (PC) was at SUBFMOLS+94 at the time of the bus error. Note that the first line of the register display is Level 7 interrupt. This is basically a worthless piece of information, as far as applications are concerned. This is because NMI, address errors, and bus errors always show level 7 interrupt.

The third page of the dump gives us four distinct groupings of information. The first is a register display, then a stack crawl, then a disassembly of the instructions surrounding the PC, and finally a portion of the stack is displayed. Using these pieces of information we can determine what went wrong.

To find out what the processor is objecting to, we start by looking at SUBFMOLS+94, the location which is at the top of the register display. Looking at the disassembly (marked 5 on page 3) we see instructions at SUBFMOLS+92 and at +96 but not at +94. Actually, it turns out that the PC leads (has already advanced past) the instruction being executed. This time the PC leads by 4, and the instruction being executed is at +90. There we see a MOVE.L (A0),(A1). This is marked 6 on page 3. Looking back to the register display, we can see that A0 looks okay but that A1 is 0. It is the reference via A1 which caused the bus error.

There is also some other handy information on page 3. Register A6 points to the stack frame (marked 1 on page 3). Matching the address contained in A6 with the stack display, we can find the parameters to SUBFMOLS. The address is marked 2. The first 2 words at that address link to the calling stack frame and the return PC for the calling procedure. Following that are the parameters in REVERSE order (marked 3). See the section on Parameters for more details.

One final note on the using the PU and PL commands. These commands use a Parallel printer connected to Slot2Chan2 or Slot2Chan1 respectively. They do not work with serial printers. The commands should be used immediately following an occurrance of a bug so the error display is preserved. If you do a stack crawl and the call is pretty deep, the stack crawl can wipe out the error display, making the information on the alternate screen less valuable.

Finding out what parameters are passed and returned

Page 4 shows a more dynamic tracing of the same bus error. The first command used is the display memory command. Its arguments request using A6 indirectly to display 40 hex words. The next command TD gives the register display. The SC (stack crawl) command gives the trace back of who called whom.

So let's find out what parameters were passed to GEMenuCmd. This routine has the calling sequence written in to the right of the stack crawl command. To find the parameters we find GEMENUCM in the stack crawl display, and look down one line to find out the stack frame. The stack frame is at F7BE88. This part of memory was then displayed using the DM command. The first word contains F7C21E which is the stack frame pointer for GeMenuEvent. The next word is an address, and using the CV (convert) command shown at the bottom of the page, we see that this is the address of GeMenuEvent+492 which is the instruction in GeMenuEvent immediately following the call to GEMenuCmd.

Following the return PC, the stack has 0007 and 0006. The parameters are in reverse order so item is 7 and menu is 6.

This example shows a very simple case, one where two integers were passed by value. Now we'll do a more complicated example. Page 5 shows the calling sequence for the Select routine in the Field Editor. First a breakpoint was set at Select+8 and then the register display is shown when the breakpoint was hit. (See the breakpoint section for more info on breakpoints). Once inside the Select routine (and past the Link instruction -- more on this in breakpoints), we proceed to display memory pointed to by A6. Remembering to skip the stack frame pointer and the return PC, the next word is the LAST parameter to Select, and it is F7F32E. This is an address because it is a var parameter -- so F7F32E is a pointer to t. Continuing, F7EE32 is a pointer to n; D60552 is a handle to a field state; D6054E is a handle to the field; The point consists of the next two integers 85 and 141.

Now let's assume we want to look at the field, and specifically, what the value of the field is currently. To do this, we have the handle to the field, and the record declaration of the field. We can use the DM command to look at D6054E and then access the first longint there D623d4 to get to the field, or we can use the shorthand DM (OD6054E) to get in one step to the field. The () means "indirect".

Examining the field, the coords rectangle is the first 4 words; maxlen is 8; growlen is 8; curlen is 1; align is 3, drawpad is 4 -- both packed into one integer; curvalue is E20802. Now we access curvalue to get the contents of the array. Looking at the display, the first byte is a lowercase g. We know that since curlen is 1 that is all the field contains.

There are a couple of other observations we can make. We can examine where these heaps map to data segments. Looking at the curvalue array we know that it is pointed to by the handle E20802. Knowing that the master pointer and the handle are in the same segment, using the first byte of the address, we can calculate the MMU number. E2/2 gives &113 which corresponds to LDSN 7. (LDSN 1 starts at 107, LDSN 2 is at 108 ...). Doing a Stop-Start calculation we see that the segment is 8K long and the handle, at 802 is at 2K into the segment, a valid address.

Note that 2 heaps (and segments) are being used here by the graphics editor. The field data structure is in one heap D6xxxx addresses and the other is used for the data components of the field and have addresses of E2xxxx. This is not the usual way of using fields and heaps, but see what you can figure out using LisaBug!

Breakpoints

Breakpoints when debugging applications are useful when in the application's domain. This is noted on the register display. Note that domain 0 or another domain overridden to 0 are not application domains, and you cannot set breakpoints in the application there. There is one special case. When in the application process, but in domain 0 (the case indicated with the brace on page 6) you can use the UBR (user break) command. This sets a breakpoint at the first instruction in the user domain, and starts executing. In the case on page 6, the breakpoint is reached at LetOthersRun+34. From this location you are in a user domain (domain 3) and in your process (process id 6) and can set breakpoints. I did a stack crawl to show that the application symbols are available at this point. Next I did a CL PC to clear the breakpoint where I am currently stopped.

There are a few rules to remember to follow when setting breakpoints. First **you should never set breakpoints on IUJSR or the future ILJSR instructions (or any other IUxxx or ILxxx instructions)**. However, you can trace through them if you don't mind seeing all the code for the trap handlers. They do not work, and will give unpredictable results. Many people have wasted hours of time because of this. The second rule is it is frequently desireable to set breakpoints after the LINK and before the UNLK instructions. Page 7 shows why. After the first register display, two breakpoints were set, one at GEMenuCmd, and one at GEMenuCmd+8. I then ran until I reached the first breakpoint. Then I did a stack crawl and displayed the stack frame. Then I ran again, stopping after the Link instruction is executed. Then I did a stack crawl and a display of the stack frame again. Note that they are very different, and the one at +8 gives correct results. I generally set breakpoints at the Procedure+8. Note that this only works for code generated with a TST.W instruction before the UNLK (the usual case, but is not guaranteed).

To set breakpoints at the end of the procedure, you will have to use the IL command to find the end of the procedure. You can usually spot this because UNLK...RTS sequence followed by the procedure name dropped in the code. An example of the end of a procedure is shown on page 4. You can even see the procedure name, although the first character is not visable because the high bit is set to indicate to LisaBug that this is a Pascal procedure. Setting a breakpoint just before executing the UNLK instruction will permit you to examine the var parameters that are being returned in exactly the way the input parameters were determined. However, to use this technique, beware of nested procedures and global gotos.

So far we have always used symbols for setting breakpoints. Sometimes it is not always possible. Sometimes the code is swapped out and LisaBug cannot find the symbols, or the code was compiled without symbols. Then you will have to use a logical address to set the breakpoint. The usual way of finding out the address is to find the IUJSR call to the routine and break on the target address. Another technique suggested by Chris Moeller, is to first let the program fail, then do a CV on the symbolic name, and then rerun the program setting the breakpoint at the logical address.

To set breakpoints when the program is coming up, you have to use a few tricks. First, you'll run the Office System under the OS shell (or Workshop shell if you have compatible libraries). Then you use the Debug command, and respond shell.office sytem for the program, and yes for the question to debug all sons. Then each process launch will give you an opportunity to set breakpoints. These breakpoints may have to be logical addresses because the probability of the code being in memory is very low (unless the OS has left the program loaded). Note that this technique of remembering logical address across process executions only works for the exact same program. Relinking the program will invalidate the logical address assignments and you will have to let it break first, find out the logical address, and then rerun and set the breakpoints.

An alternative suggest by Rod Perkins is to bring up the filer, hit NMI opportunely in domain 0, then set a breakpoint on 0:Declare_Excep_Hdl. When it stops at the breakpoint (in domain 0), issue CL PC to clear the breakpoint, Then issue the UBR command. It will then break in your application.

Local and Global Variables

It is frequently useful to be able to trace through a routine and determine what the value of some variable is. To do this, you need to understand the layout of the stack. Page 8 shows a diagram of the stack. Note that in this diagram, the addresses go from low to high. Global variables are accessed by adding negative numbers to A5, and local variables are accessed by adding negative numbers to A6. Intrinsic unit globals are accessed by first adding positive numbers to A5 to get to the data pointer table entry, and then taking the value found there and adding negative numbers to that.

To show how you can figure out values of local and global variables while stepping through a procedure, I picked out a very small procedure. Its source listing is on page 9. Page 10 contains the disassembly of the procedure. The process of determining where a variable is in memory requires some matching of the source with the code generated. What I usually do is use the IUJSR instructions to determine rough areas of code and then look in more detail at the generated code from there.

Page 10 also sets a breakpoint at copysel+8 and runs until the breakpoint was hit; then the stack frame was printed out. Note that the CutCopyField procedure is not in memory. We can tell that by the fact that at CopySel+5C there is an IUJSR to $8E000E instead of CutCopyS.

Looking on page 11, we see that an ID of $8E000E gives the invalid logical address message. To illustrate setting a breakpoint on a logical address, I set a breakpoint at $8E000E. Also a breakpoint was set at CopySel+14.

At the CopySel+14 breakpoint, we are about to compare TypeofSel with aCellTxTSl. TypeofSel is a Variable of an enumerated type, and aCellTxTSl is one of the values. Its value is 1. So displaying ra4+$ffffffc9 displays the value of TypeofSel. Note that the instruction is CMPI.B #$0001,$ffc9(a4). The DM command uses $ffffffc9 because we want to maintain the fact that it is a negative quantity. RA4+$ffffffc9 yields 0f7cf49, an odd address. Note that LisaBug, however, starts the display at 0f7cf48, so the byte we are testing is the rightmost byte of the first word. Note also that the access is relative to A4, but that A4 was loaded relative to A5. This is because this is a global variable in an intrinsic unit, and A4 contains the pointer to the base of the globals for this unit.

At CopySel+24 we access another intrinsic unit global, this time it is tblpars.editcoltitle. It is again at an odd address. The variable is a boolean, and hence its value is true.

At CopySel+3E we are pushing a parameter to SetPnlPort. It again is an intrinsic unit global. The parameter is an integer, and displaying the value shows it to be 3. Next, the trace command was used to step to the next instruction. Note that the value of A7 has changed, and that A7 points to the value just pushed on the stack.

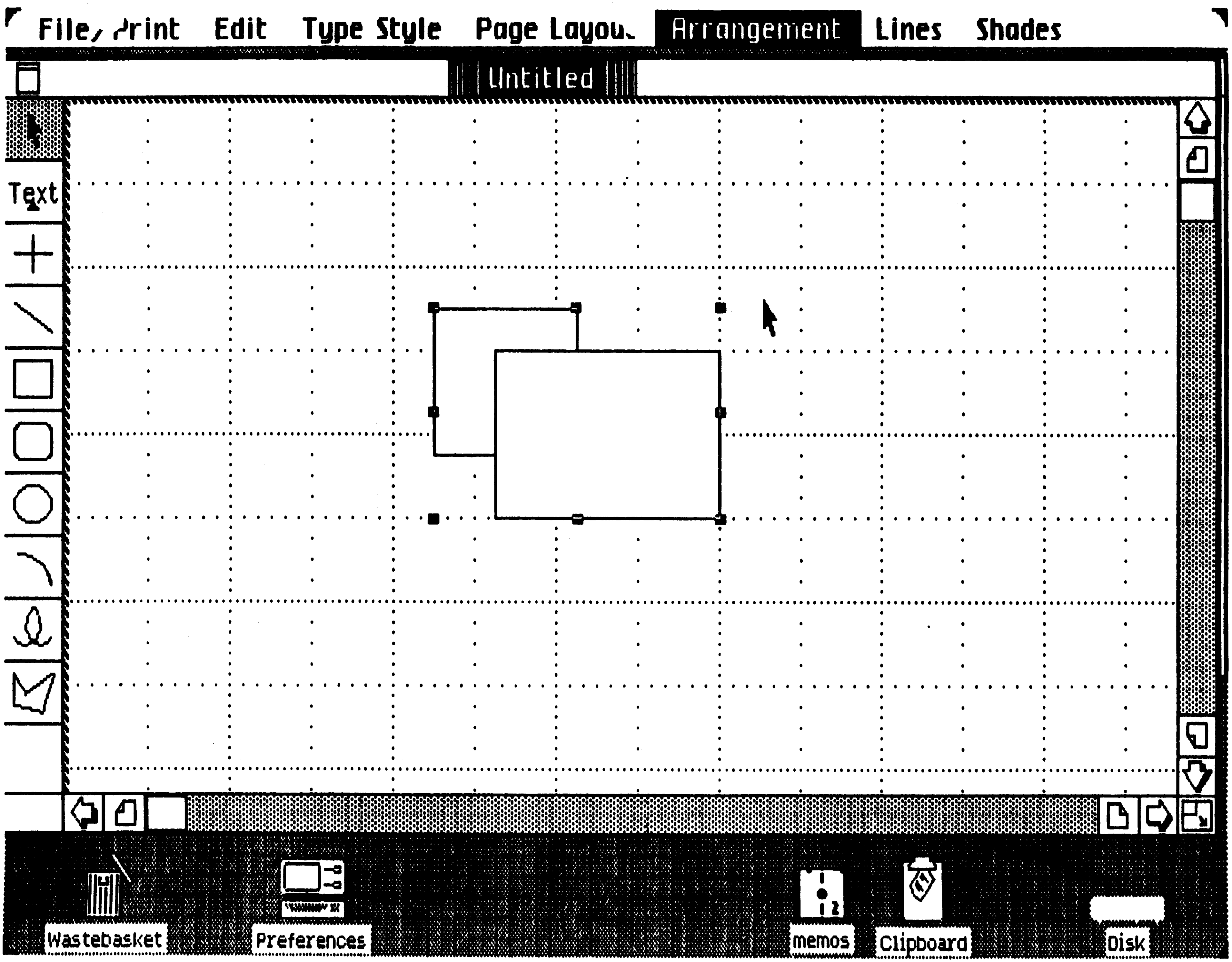
On page 12, we are pushing the effective address of a local variable, errnum. Note that the reference is relative to A6. When the value is displayed, its value is BF52 (garbage since its value is set by the routine).

Continuing on, we hit the breakpoint at $8E000E. This is a digression from the flow of finding out the value returned from CutCopyField, so I'll just show how you can get into CutCopyField and get out. This address where we stopped is actually a jump table entry, so we trace through the instruction and get to CutCopyField. (A jump table is used when calling from one segment to another). After a few more traces to get past the LINK instruction, we check the address of the last parameter passed to CutCopyField, and it is indeed the address of errnum we found before. Next, a breakpoint was set to the return PC.

After continuing, we break in CopySel immediately after the return from CutCopyField. Displaying the location containing errnum, we see CutCopyField returned 0000.

Function returns

It is frequently useful to determine what a function returns. To do this break at the instruction immediately following the JSR or IUJSR to the function. Then the function return is on the top of the stack. DM ra7 will display the returned value.


File/Print
Edit
Type Style
Page Layou.
Arrangement
Lines
Shades
Untitled
Text
Wastebasket
Preferences
memos
Clipboard
Disk

LisaBug - 1

1

```
█
Level 7 Interrupt
SUBFMOLS+0094 0008 5340                 ORI.B   #$5340,A0
PC=00281A32 SR=0000  0  US=00F7BDCC SS=00CBFED8 DO=1 P#=00007
D0=00000000 D1=00000100 D2=0000FFCE D3=00D007E4
D4=0C280005 D5=00145700 D6=0000000A D7=00DA0AB8
A0=00DA0AB8 A1=00000000 A2=00CE004C A3=00F7F466
A4=00F7F466 A5=00F7F4A6 A6=00F7BDD8 A7=00F7BDCC
>pu
█
```

```
BUS ERROR in process of gid       7
Process is about to be terminated.
access address =          0 = mmu#         0  ,    offset        0
 inst reg =    8848 sr =         0  pc = 2628146
 saved registers at 13369270
Going to Lisabug, type g to continue.
█
```

LisaBug-2

```
Level 7 Interrupt
SUBFMOLS+0094 0008 5340       PC       ORI.B   #$5340,A0
r  00281A32 SR=0000  0  US=00F7BDCC SS=00CBFED8 DO=1 P#=00007
 u=00000000 D1=00000100 D2=0000FFCE D3=00D007E4
D4=0C280005 D5=00145700 D6=0000000A D7=00DA0AB8
A0=00DA0AB8 A1=00000000 A2=00CE004C A3=00F7F466
A4=00F7F466 A5=00F7F4A6 A6=00F7BDD8 A7=00F7BDCC
                                   1
4 At SUBFMOLS+0094
Stack frame at 00F7BDD8 called from COMMITLA+033A
Stack frame at 00F7BE14 called from LOCKCMD+007A
Stack frame at 00F7BE2E called from GEMENUCM+02A0
Stack frame at 00F7BE88 called from GEMENUEV+048E
Stack frame at 00F7C21E called from PROCESST+011E
Stack frame at 00F7C258 called from MAINPROG+008A
Stack frame at 00F7C298 called from GRAPHICS+001E
Stack frame at 00F7F4A6
SUBFMOLS+0074 2968 0004 0004           MOVE.L  $0004(A0),$0004(A4)
SUBFMOLS+007A 6016                     BRA.S   *+$0018           ; 00281A30
SUBFMOLS+007C 2047                     MOVE.L  D7,A0
SUBFMOLS+007E 2247                     MOVE.L  D7,A1
SUBFMOLS+0080 2251                     MOVE.L  (A1),A1
SUBFMOLS+0082 2368 0004 0004           MOVE.L  $0004(A0),$0004(A1)
SUBFMOLS+0088 2047                     MOVE.L  D7,A0
SUBFMOLS+008A 2247                     MOVE.L  D7,A1
SUBFMOLS+008C 2269 0004                MOVE.L  $0004(A1),A1
SUBFMOLS+0090 2290                     MOVE.L  (A0),(A1) 6
5 SUBFMOLS+0092 302C 0008              MOVE.W  $0008(A4),D0
SUBFMOLS+0096 5340                     SUBQ.W  #$1,D0
 FMOLS+0098 3940 0008                  MOVE.W  D0,$0008(A4)
SUBFMOLS+009C 4267                     CLR.W   -(A7)
SUBFMOLS+009E 2F07                     MOVE.L  D7,-(A7)
SUBFMOLS+00A0 4EBA F10A                JSR     CNTOFOBJ          ; 00280B4A
SUBFMOLS+00A4 302C 000A                MOVE.W  $000A(A4),D0
SUBFMOLS+00A8 905F                     SUB.W   (A7)+,D0
SUBFMOLS+00AA 3940 000A                MOVE.W  D0,$000A(A4)
SUBFMOLS+00AE 42A7                     CLR.L   -(A7)
00F7BDB8     00F7 BE02 002A 353C  00F7 BDEC 00F7 EB88 .....*5<........
00F7BDC8     00DA 0A6C 0000 0001  00DA 0A64 00F8 04B6 ...l.......d....
2 00F7BDD8   00F7 BE14 002A 2764 3 00DA 0AB8 00F7 F466 .....*'d.......f
00F7BDE8     0000 000A 0000 0197  00DA 0A64 00F8 04B6 ...........d....
00F7BDF8     0002 0088 5FC2 00DA  0AB8 00F7 BE12 002A ...._..........*
00F7BE08     35A8 0000 0000 0000  0000 00F7 00F7 BE2E 5...............
00F7BE18     0062 0888 0014 5700  0000 0001 0000 0007 .b....W.........
00F7BE28     BE30 0036 3316 00F7  BE88 0064 1400 016E .0.63......d...n
```

Lisa Bug-3

PARAMETERS


```
>dm ra6 40
00F7BDD8     00F7 BE14 002A 2764  00DA 0AB8 00F7 F466 .....*'d.......f
00F7BDE8     0000 000A 0000 0197  00DA 0A64 00F8 04B6 ...........d....
  7BDF8      0002 0088 5FC2 00DA  0AB8 00F7 BE12 002A ...._..........*
0uF7BE08     35A8 0000 0000 0000  0000 00F7 00F7 BE2E 5...............
>td
SUBFMOLS+0094 0008 5340       PC        ORI.B   #$5340,A0
PC=00281A32 SR=0000  0  US=00F7BDCC SS=00CBFED8 DO=1 P#=00007
D0=00000000 D1=00000100 D2=0000FFCE D3=00D007E4
D4=0C280005 D5=00145700 D6=0000000A D7=00DA0AB8
A0=00DA0AB8 A1=00000000 A2=00CE004C A3=00F7F466
A4=00F7F466 A5=00F7F4A6 A6=00F7BDD8 A7=00F7BDCC
>sc
At SUBFMOLS+0094
Stack frame at 00F7BDD8 called from COMMITLA+033A
Stack frame at 00F7BE14 called from LOCKCMD+007A
Stack frame at 00F7BE2E called from GEMENUCM+02A0
Stack frame at 00F7BE88 called from GEMENUEV+048E
Stack frame at 00F7C21E called from PROCESST+011E
Stack frame at 00F7C258 called from MAINPROG+008A
Stack frame at 00F7C298 called from GRAPHICS+001E
Stack frame at 00F7F4A6
>dm 0f7be88 30
00F7BE88     00F7 C21E 0064 18DC  0007 0006 0C28 0002 .....d.......(..
00F7BE98     0014 57C0 2F00 4267  2F2E FFD4 201F 0A01 ..W./.Bg/.......
00F7BEA8     00F8 04B6 0001 1453  6574 2041 7369 6465 .......Set.Aside
>il 6418dc-20
GEMENUEV+0472 FFDC                      $$$$
GEMENUEV+0474 486E FFD2                 PEA      $FFD2(A6)
GEMENUEV+0478 486E FFD4                 PEA      $FFD4(A6)
GEMENUEV+047C A088 02B4                 IUJSR    MENUSELE        ; 008861AC
GEMENUEV+0480 4A6E FFD4                 TST.W    $FFD4(A6)
GEMENUEV+0484 670C                      BEQ.S    *+$000E         ; 006418DC
GEMENUEV+0486 3F2E FFD2                 MOVE.W   $FFD2(A6),-(A7)
GEMENUEV+048A 3F2E FFD4                 MOVE.W   $FFD4(A6),-(A7)
GEMENUEV+048E 4EBA F882                 JSR      GEMENUCM        ; 0064115C
GEMENUEV+0492 4267                      CLR.W    -(A7)
GEMENUEV+0494 A088 022A                 IUJSR    HILITEME        ; 00885C18
GEMENUEV+0498 4CDF 18F0                 MOVEM.L  (A7)+,D4-D7/A3/A4
GEMENUEV+049C 4E5E                      UNLK     A6
GEMENUEV+049E 2E9F                      MOVE.L   (A7)+,(A7)
GEMENUEV+04A0 4E75                      RTS
GEMENUEV+04A2 C745 4D45 4E55 4556  0060 2000 0000 0000 .EMENUEV.`......
GEMENUEV+04B2 0000 0000 0000 0000  0000 0000 0000 0000 ................
GEMENUEV+04C2 0000 0000 0000 0000  0000 0000 0000 0000 ................
GEMENUEV+04D2 0000 0000 0000 0000  0000 0000 0000 0000 ................
GEMENUEV+04E2 0000 0000 0000 0000  0000 0000 0000 0000 ................

>cv 6418dc
$6418DC=&6559964=GEMENUEV+0492
>pr 0
```

GemenuCmd (menu, item: Integer);

PARAMETERS

procedure Select (dxy:Point; hf:hndField; hfs:hndFState; var n:Rect; var t:integer);

```
)hr select+8

Break Point
SELECT+0008   *48E7 0118               MOVEM.L D7/A3/A4,-(A7)
PC=008C3FA8 SR=0000  0  US=00F7C010 SS=00CC0000 DO=1 P#=00008
D0=00020001 D1=00E20000 D2=00000002 D3=001FFFFF
D4=0010FFFA D5=00000001 D6=FFFC3900 D7=0007FFFE
A0=00F804B6 A1=00F7F32E A2=00CE004C A3=70061080
A4=00F804B6 A5=00F7F4A6 A6=00F7C018 A7=00F7C010
)dm ra6 40
00F7C018     00F7 C034 002A 08DE  00F7 F32E 00F7 EE32 ...4.*.........2
00F7C028     00D6 0552 00D6 054E  0085 0141 00F7 C07E ...R...N...A...~
00F7C038     0062 1884 0062 0085  0141 00DA 0896 397C .b...b...A....9|
00F7C048     0010 FFFC 397C 0007  FFFE 7006 1080 00F8 ....9|....p.....
)dm 0d6054e
00D6054E     00D6 23D4 00D6 23AE  00D6 2394 00D6 237A ..#...#...#...#z
)dm 0d623d4 40
00D623D4     007E 013B 008A 014D  0008 0008 0001 0304 .~.;...M........
00D623E4     00E2 0802 0001 0001  0001 00E2 0806 002E ................
00D623F4     4012 054A 001E FFFD  002A 0025 0008 0008 @..J.....*.%....
00D62404     0004 0304 00D6 053A  0001 0001 0001 00D6 ......:.........
)dm (0d6054e) 40
00D623D4     007E 013B 008A 014D  0008 0008 0001 0304 .~.;...M........
00D623E4     00E2 0802 0001 0001  0001 00E2 0806 002E ................
00D623F4     4012 054A 001E FFFD  002A 0025 0008 0008 @..J.....*.%....
00D62404     0004 0304 00D6 053A  0001 0001 0001 00D6 ......:.........
       (0e20802) 10
00E20832     67E2 1FF4 00E2 1FF4  0000 0006 00E2 0846 g..............F
)cv e2/2
$71=&113=00000071
)mm &113
D[1] Segment[71] Origin[65C] Limit[F0] Control[7] Start[0CB800] Stop[0CD7FF]
)cv 0cd7ff-0cb800
$1FFF=&8191=00001FFF
)cv 802
$0802=&2050=00000802
```

```
field = record                             { static field characteristics  }
    coords:      Rect;                     { bounding rectangle            }
    maxLen:      integer;  492             { maximum number of chars       }
                                           {     (should equal size of     }
                                           {     curvalue array)           }
    growLen:     integer;  82              { size by which to grow value   }
                                           {     array - don't grow if 0   }
    curLen:      integer;  c4B7            { current number of chars       }
    align:       byte;                     { alignment of chars when field }
                                           {     is displayed              }
    drawPad:     byte;                     { # of pixels to draw from left }
                                           {     or right (depending on    }
                                           {     alignment)                }
    curValue:    hndData;                  { handle of array of contents   }
    maxFmts:     integer;                  { maximum # of format records   }
    growFmts:    integer;                  { # of format records by which  }
                                           {     to grow - don't grow if 0 }
    curFmts:     integer;                  { current # of format records   }
    fmtInfo:     hndRuns;                  { handle to array of runs       }
    protect:     boolean;                  { true => changes not allowed   }
    end;
ptrField = ^field;
hndField = ^ptrField;
```

Setting Breakpoints / Domains

6

```
D0=00000013 D1=G0000000 D2=00000002 D3=001F2714
D4=2D48F900 D5=0010A84E D6=2D48FE00 D7=00000000
A0=00004004 A1=00CBFF42 A2=00208C04 A3=0020A022
A4=00CCB10E A5=0000057A A6=00CBFF6A A7=00CBFF36
>g

  el 7 Interrupt
00220E62     4CDF 08E0                MOVEM.L (A7)+,D5-D7/A3
PC=00220E62 SR=2004  0  US=00F7C25A SS=00CBFFB8 DO=0
D0=00000000 D1=0000FFFF D2=000006A5 D3=00CE07F3
D4=0010FFFA D5=00020000 D6=00CC4F86 D7=00A80700
A0=0036024E A1=00A84270 A2=00D08000 A3=00000400
A4=00A8426C A5=00CC4088 A6=00CBFFE4 A7=00CBFFB8
>g

Level 7 Interrupt
QUEUE_PR+0066 D280                    ADD.L   D0,D1
PC=00260BF4 SR=0700  0  US=00F7DC32 SS=00CC0000 DO=0 P#=00006
D0=FFFFB481 D1=00CCA083 D2=00000002 D3=00D007E4
D4=2D48F900 D5=0010B004 D6=2D480078 D7=00F7DC5E
A0=00CCA832 A1=00F7DC62 A2=00CE004C A3=0020A022
A4=00CCB10E A5=00CC4088 A6=00F7DC4C A7=00F7DC32
>ubr

Break Point
LETOTHER+0034*4E5E                    UNLK    A6
PC=00883D3C SR=0000  0  US=00F7DC72 SS=00CC0000 DO=3 P#=00006
D0=00002000 D1=00000002 D2=00000002 D3=001FFFFF
D4=2D48F900 D5=0010B004 D6=2D48FE00 D7=4AAE0000
A0=00F7DC72 A1=00CCA083 A2=00CE004C A3=0020A022
A4=02E46010 A5=00F7F9A4 A6=00F7DC74 A7=00F7DC72
>sc
.  LETOTHER+0034
Stack frame at 00F7DC74 called from MAINLOOP+0196
Stack frame at 00F7DCCA called from 00240030
Stack frame at 00F7F9A4
>cl pc
>g

Level 7 Interrupt
00208C68     4840                     SWAP    D0
PC=00208C68 SR=2700  0  US=00F7DC72 SS=00CBFF4A DO=3 overridden 0
D0=00FE00FE D1=40000000 D2=00000000 D3=00D02704
D4=2D48F900 D5=0010C08D D6=2D48FE00 D7=4AAE0000
A0=0000402C A1=00004000 A2=00208C2C A3=0020A022
A4=02E46010 A5=0000057A A6=00CBFF7E A7=00CBFF4A
>sc
At 00208C68
Stack frame at 00CBFF7E called from 0020A9A4
Stack frame at 00CBFFA8 called from 0020AA36
Stack frame at 00CBFFB0 called from 0020CC9A
Stack frame at 00CBFFDC called from 00208466
Stack frame at 00CBFFFC
>ubr

Level 7 Interrupt
00208474     4E75                     RTS
PC=00208474 SR=2000  0  US=00F7DC72 SS=00CBFFEC DO=3 overridden 0
D0=00000002 D1=00000002 D2=00000002 D3=00D007E4
D4=2D48F900 D5=0010C6B9 D6=2D48FE00 D7=4AAE0000
A0=0020CD14 A1=00000414 A2=00CE004C A3=0020A022
A4=02E46010 A5=00CC4088 A6=00CBFFFC A7=00CBFFEC
>ubr

Level 7 Interrupt
```

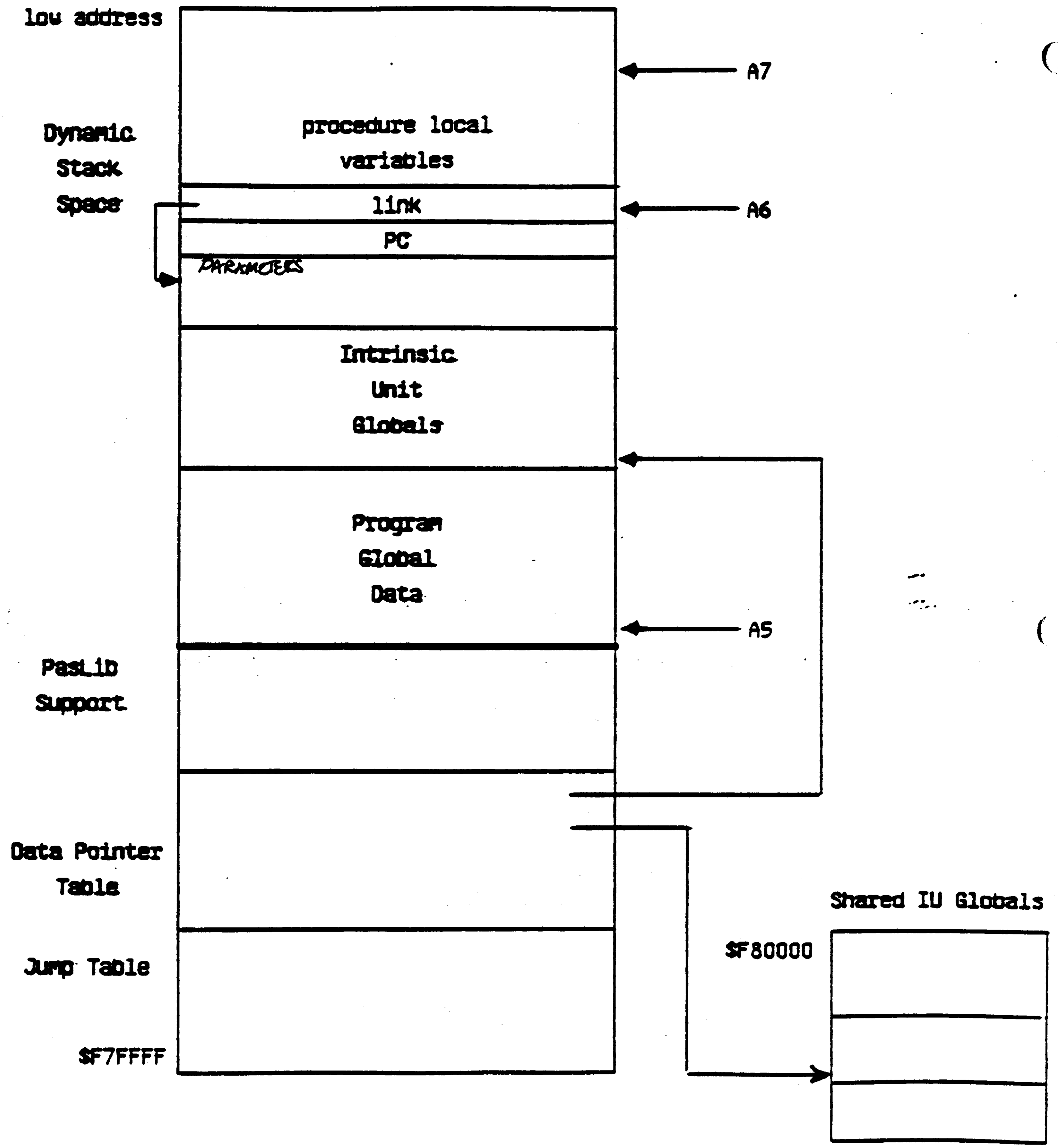
BREAKPOINTS

(7)

```
>g

Level 7 Interrupt
f32%pack+0008 4A02                    TST.B    D2
  00AC0D76 SR=0001  0  US=00F7C1DC SS=00CC0000 DO=1 P#=00008
D0=FFFF0002 D1=0000000F D2=80000000 D3=00D007E4
D4=0010FFFA D5=397C0010 D6=FFFC397C D7=43180000
A0=00F7EB58 A1=00F7C226 A2=00CE004C A3=00F7E054
A4=00F7EB64 A5=00F7F4A6 A6=00F7C20C A7=00F7C1DC
>br gemenucm
>br gemenucm+8
>g

Break Point
GEMENUCM+0000*4A6F EFB4       GEMENUCM TST.W    $EFB4(A7)
PC=0064115C SR=0010  0  US=00F7BE8C SS=00CC0000 DO=1 P#=00008
D0=000000FF D1=000000FF D2=00000002 D3=001FFFFF
D4=00100005 D5=397C0000 D6=FFFC3900 D7=0000000E
A0=006418CA A1=00F8054A A2=00CE004C A3=70061080
A4=00F804B6 A5=00F7F4A6 A6=00F7C21E A7=00F7BE8C
>sc
At GEMENUCM+0000
Stack frame at 00F7C21E called from PROCESST+011E
Stack frame at 00F7C258 called from MAINPROG+008A
Stack frame at 00F7C298 called from GRAPHICS+001E
Stack frame at 00F7F4A6
>dm ra6 30
00F7C21E      00F7 C258 0064 1F90  00F7 C22E 206E FFFC ...X.d.......n..
00F7C22E      00F8 0548 0001 0007  00C5 0010 706A 0000 ...H........pj..
00F7C23E      0000 0000 0100 06AC  0000 0000 0008 0000 ................

Break Point
GEMENUCM+0008*2F07                    MOVE.L  D7,-(A7)
PC=00641164 SR=0010  0  US=00F7BE3C SS=00CC0000 DO=1 P#=00008
D0=000000FF D1=000000FF D2=00000002 D3=001FFFFF
D4=00100005 D5=397C0000 D6=FFFC3900 D7=0000000E
A0=006418CA A1=00F8054A A2=00CE004C A3=70061080
A4=00F804B6 A5=00F7F4A6 A6=00F7BE88 A7=00F7BE3C
>sc
At GEMENUCM+0008
Stack frame at 00F7BE88 called from GEMENUEV+048E
Stack frame at 00F7C21E called from PROCESST+011E
Stack frame at 00F7C258 called from MAINPROG+008A
Stack frame at 00F7C298 called from GRAPHICS+001E
Stack frame at 00F7F4A6
>dm ra6 30
00F7BE88      00F7 C21E 0064 18DC  000B 0004 0010 FFFA .....d..........
00F7BE98      397C 0010 FFFC 397C  0007 FFFE 7006 1080 9|....9|....p...
00F7BEA8      00F8 04B6 00AC 1453  6574 2041 7369 6465 .......Set.Aside
>il gemenucm
GEMENUCM+0000*4A6F EFB4       GEMENUCM TST.W    $EFB4(A7)
GEMENUCM+0004 4E56 FFB4                LINK     A6,#$FFB4
GEMENUCM+0008*2F07            PC       MOVE.L   D7,-(A7)
GEMENUCM+000A 3E2E 0008                MOVE.W   $0008(A6),D7
GEMENUCM+000E 4EAD 096E                JSR      SETWRKOR         ; 003632DE
GEMENUCM+0012 302E 000A                MOVE.W   $000A(A6),D0
GEMENUCM+0016 5540                     SUBQ.W   #$2,D0
GEMENUCM+0018 6B00 02C2                BMI      *+$02C4          ; 00641438
GEMENUCM+001C 0C40 000A                CMPI.W   #$000A,D0
GEMENUCM+0020 6E00 02BA                BGT      *+$02BC          ; 00641438
GEMENUCM+0024 E348                     LSL.W    #$1,D0
GEMENUCM+0026 303B 0006                MOVE.W   *+$0008(D0.W),D0 ; 0064118A
```

LisaBug-7

Stack Segment Layout



VAR

```
{$S smgrLoUse}
PROCEDURE CopySel(status : integer);
   VAR errnum : integer;
   BEGIN
   IF TraceSMGR then Writeln('%tmsprocs CopySel' );
   if (typeofSel = aCellTxTSl) or
   (tblPars.EditColTitle and (typeOfSel = aColHedSl)) or
   (tblPars.EditRowTitle and (typeOfSel = aRowHedSl)) then
      Begin
      SetPnlPort(WidePnl);
      CutCopyField(wavFieldH, wavFstateH, false, true, errnum);
      Status := errnum;
      CutCopyField(selFieldH, selFstateH, false, false, errnum);
      END;
   END;
```

```
.il cppysel
COPYSEL+0000  4A6F EFFE       COPYSEL  TST.W    $EFFE(A7)
COPYSEL+0004  4E56 FFFE                LINK     A6,#$FFFE
COPYSEL+0008  48E7 0018                MOVEM.L A3/A4,-(A7)
  PYSEL+000C  286D 02A0                MOVE.L   $02A0(A5),A4
COPYSEL+0010  266D 029C                MOVE.L   $029C(A5),A3
COPYSEL+0014  0C2C 0001 FFC9           CMPI.B   #$0001,$FFC9(A4)
COPYSEL+001A  57C0                     SEQ      D0
COPYSEL+001C  0C2C 0009 FFC9           CMPI.B   #$0009,$FFC9(A4)
COPYSEL+0022  57C1                     SEQ      D1
COPYSEL+0024  C22B FFDB                AND.B    $FFDB(A3),D1
COPYSEL+0028  8001                     OR.B     D1,D0
COPYSEL+002A  0C2C 0008 FFC9           CMPI.B   #$0008,$FFC9(A4)
COPYSEL+0030  57C1                     SEQ      D1
COPYSEL+0032  C22B FFE2                AND.B    $FFE2(A3),D1
COPYSEL+0036  8001                     OR.B     D1,D0
COPYSEL+0038  0240 0001                ANDI.W   #$0001,D0
COPYSEL+003C  673A                     BEQ.S    *+$003C          ; 0056065C
COPYSEL+003E  3F2B FFCC                MOVE.W   $FFCC(A3),-(A7)
COPYSEL+0042  A050 0170                IUJSR    SETPNLPO         ; 0050089E
COPYSEL+0046  2F2C F442                MOVE.L   $F442(A4),-(A7)
>il
COPYSEL+004A  2F2C F43C                MOVE.L   $F43C(A4),-(A7)
COPYSEL+004E  4267                     CLR.W    -(A7)
COPYSEL+0050  1F3C 0001                MOVE.B   #$0001,-(A7)
COPYSEL+0054  486E FFFE                PEA      $FFFE(A6)
COPYSEL+0058  A08E 000E                IUJSR    $008E000E
COPYSEL+005C  206E 0008                MOVE.L   $0008(A6),A0
COPYSEL+0060  30AE FFFE                MOVE.W   $FFFE(A6),(A0)
COPYSEL+0064  2F2C FFC4                MOVE.L   $FFC4(A4),-(A7)
 PYSEL+0068   2F2C F44E                MOVE.L   $F44E(A4),-(A7)
COPYSEL+006C  4267                     CLR.W    -(A7)
COPYSEL+006E  4267                     CLR.W    -(A7)
COPYSEL+0070  486E FFFE                PEA      $FFFE(A6)
COPYSEL+0074  A08E 000E                IUJSR    $008E000E
COPYSEL+0078  4CDF 1800                MOVEM.L (A7)+,A3/A4
COPYSEL+007C  4E5E                     UNLK     A6
COPYSEL+007E  2E9F                     MOVE.L   (A7)+,(A7)
COPYSEL+0080  4E75                     RTS
COPYSEL+0082  C34F 5059 5345 4C20  0000 4A6F EFFE 4E56 .OPYSEL...Jo..NV
CUTSEL+0000   4A6F EFFE       CUTSEL   TST.W    $EFFE(A7)
CUTSEL+0004   4E56 FFFE                LINK     A6,#$FFFE
>br copysel+8
>g

Break Point
COPYSEL+0008 *48E7 0018                MOVEM.L A3/A4,-(A7)
PC=005605EC SR=0000  0  US=00F7BEE8 SS=00CC0000 DO=1 P#=00005
D0=00000000 D1=00000000 D2=00000000 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
A0=005214C0 A1=00F7BEE0 A2=00885C00 A3=00F804B6
A4=00F7D766 A5=00F7F73A A6=00F7BEEA A7=00F7BEE8
>dm ra6 40
00F7BEEA      00F7 BF52 0052 150C  00F7 BF50 A03C 009E ...R.R.....P.<..
00F7BEFA      4EAD 0005 00F7 D766  00F8 04B6 00F7 BF00 N......f........
 0F7BF0A      000E 2F2D 4E01 0002  00F8 04B6 00F7 DF26 ../-N..........&
 0F7BF1A      00F7 BF38 0088 5D0E  00F7 0000 0002 001F ...8..].........
>pr 0
```

```
     id 8e000e

Invalid log addr
>br 8e000e
>br copysel+14
>g

Break Point
COPYSEL+0014 *0C2C 0001 FFC9           CMPI.B  #$0001,$FFC9(A4)
PC=005605F8 SR=0000  0  US=00F7BEE0 SS=00CC0000 DO=1 P#=00005
D0=00000000 D1=00000000 D2=00000000 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
A0=005214C0 A1=00F7BEE0 A2=00885C00 A3=00F7D766
A4=00F7CF80 A5=00F7F73A A6=00F7BEEA A7=00F7BEE0
>dm ra4+$ffffffc9
00F7CF48       0101 000C 0010 00D8  072E 00D8 0746 00D8 .............F..
>br copysel+24
>g

Break Point
COPYSEL+0024 *C22B FFDB                AND.B   $FFDB(A3),D1
PC=00560608 SR=0009  0  US=00F7BEE0 SS=00CC0000 DO=1 P#=00005
D0=000000FF D1=00000000 D2=00000000 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
A0=005214C0 A1=00F7BEE0 A2=00885C00 A3=00F7D766
  =00F7CF80 A5=00F7F73A A6=00F7BEEA A7=00F7BEE0
>dm ra3+$ffffffdb
00F7D740       0101 0100 0001 0108  0000 0001 0000 0048 ...............H
>cv ra3+$ffffffdb
$F7D741=&16242497=00F7D741
>cv ra4+$ffffffc9
$F7CF49=&16240457=00F7CF49
>br copysel+3e
>g

Break Point
COPYSEL+003E *3F2B FFCC                MOVE.W  $FFCC(A3),-(A7)
PC=00560622 SR=0000  0  US=00F7BEE0 SS=00CC0000 DO=1 P#=00005
D0=00000001 D1=00000000 D2=00000000 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
A0=005214C0 A1=00F7BEE0 A2=00885C00 A3=00F7D766
A4=00F7CF80 A5=00F7F73A A6=00F7BEEA A7=00F7BEE0
>dm ra3+$ffffffcc
00F7D732       0003 0002 0001 0000  0001 0100 0101 0101 ................
>t

Trace Point
COPYSEL+0042  A050 0170                IUJSR   SETPNLPO        ; 0050089E
PC=00560626 SR=8000  0  US=00F7BEDE SS=00CC0000 DO=1 P#=00005
D0=00000001 D1=00000000 D2=00000000 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
 J=005214C0 A1=00F7BEE0 A2=00885C00 A3=00F7D766
A4=00F7CF80 A5=00F7F73A A6=00F7BEEA A7=00F7BEDE
:>dm ra7
00F7BEDE       0003 00F8 04B6 00F7  D766 BF52 00F7 BF52 .........f.R...R
>br copysel+54
>g
```

11

LisaBug-11

```
COPYSEL+0054 *486E FFFE                   PEA     $FFFE(A6)
PC=00560638 SR=0000  0  US=00F7BED4 SS=00CC0000 DO=1 P#=00005
D0=00000000 D1=00000000 D2=00000000 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
A0=0056062A A1=00F20BEC A2=00885C00 A3=00F7D766
A4=00F7CF80 A5=00F7F73A A6=00F7BEEA A7=00F7BED4
>dm ra6+$fffffffe
00F7BEE8       BF52 00F7 BF52 0052  150C 00F7 BF50 A03C .R...R.R.....P.<
>dm ra6+$fffffffe
>g

Break Point
008E000E      *4EF9 008E 068E             JMP     $008E068E
PC=008E000E SR=0008  0  US=00F7BECC SS=00CC0000 DO=1 P#=00005
D0=00000000 D1=00000000 D2=00000000 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
A0=0056062A A1=00F20BEC A2=00885C00 A3=00F7D766
A4=00F7CF80 A5=00F7F73A A6=00F7BEEA A7=00F7BECC
>t

Trace Point
CUTCOPYF+0000 4A6F EFD0       CUTCOPYF TST.W   $EFD0(A7)
PC=008E068E SR=8008  0  US=00F7BECC SS=00CC0000 DO=1 P#=00005
D0=00000000 D1=00000000 D2=00000000 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
A0=0056062A A1=00F20BEC A2=00885C00 A3=00F7D766
A4=00F7CF80 A5=00F7F73A A6=00F7BEEA A7=00F7BECC
:>t

Trace Point
CUTCOPYF+0004 4E56 FFD0                   LINK    A6,#$FFD0
PC=008E0692 SR=8000  0  US=00F7BECC SS=00CC0000 DO=1 P#=00005
D0=00000000 D1=00000000 D2=00000000 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
A0=0056062A A1=00F20BEC A2=00885C00 A3=00F7D766
A4=00F7CF80 A5=00F7F73A A6=00F7BEEA A7=00F7BECC
:>t

Trace Point
CUTCOPYF+0008 48E7 0318                   MOVEM.L D6/D7/A3/A4,-(A7)
PC=008E0696 SR=8000  0  US=00F7BE98 SS=00CC0000 DO=1 P#=00005
D0=00000000 D1=00000000 D2=00000000 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
A0=0056062A A1=00F20BEC A2=00885C00 A3=00F7D766
A4=00F7CF80 A5=00F7F73A A6=00F7BEC8 A7=00F7BE98
:>dm ra6
00F7BEC8       00F7 BEEA 0056 0640  00F7 BEE8 01F8 0000 .....V.@........
>br 560640
>g

Break Point
COPYSEL+005C *206E 0008                   MOVE.L  $0008(A6),A0
PC=00560640 SR=0000  0  US=00F7BEE0 SS=00CC0000 DO=1 P#=00005
D0=00002700 D1=00000000 D2=00000002 D3=001FFFFF
D4=000E2F2D D5=FAEA3F07 D6=A03C0005 D7=4EAD0005
A0=00560640 A1=00CC94CC A2=00CE004C A3=00F7D766
A4=00F7CF80 A5=00F7F73A A6=00F7BEEA A7=00F7BEE0
>dm ra6+$fffffffe
00F7BEE8       0000 00F7 BF52 0052  150C 00F7 BF50 A03C .....R.R.....P.<
>cl
>g
```

Shell-Writer's Guide

This document contains information you need to know to write a shell for the Lisa. It describes the things a shell must do when it starts up and when it terminates. To use this document, you should be familiar with the *Operating System Reference Manual* and have some knowledge of Pascal. To do any graphics, you will have to use QuickDraw, described in the *Pascal Reference Manual*. You may also want to use calls in the **PaslibCall** and **PPaslibC** units.

The System.shell

When the OS is booted, it starts the 'root' process, which searches the boot disk for a shell called '**system.shell**'. The **system.shell** is automatically started, and will be the ancestor of all other shell processes (see Figure 1). All shells must be "plug-compatible" with each other so that any shell can be the **system.shell** without special support from the OS. In this way, a turn-key boot disk could be prepared that didn't include a selector shell.

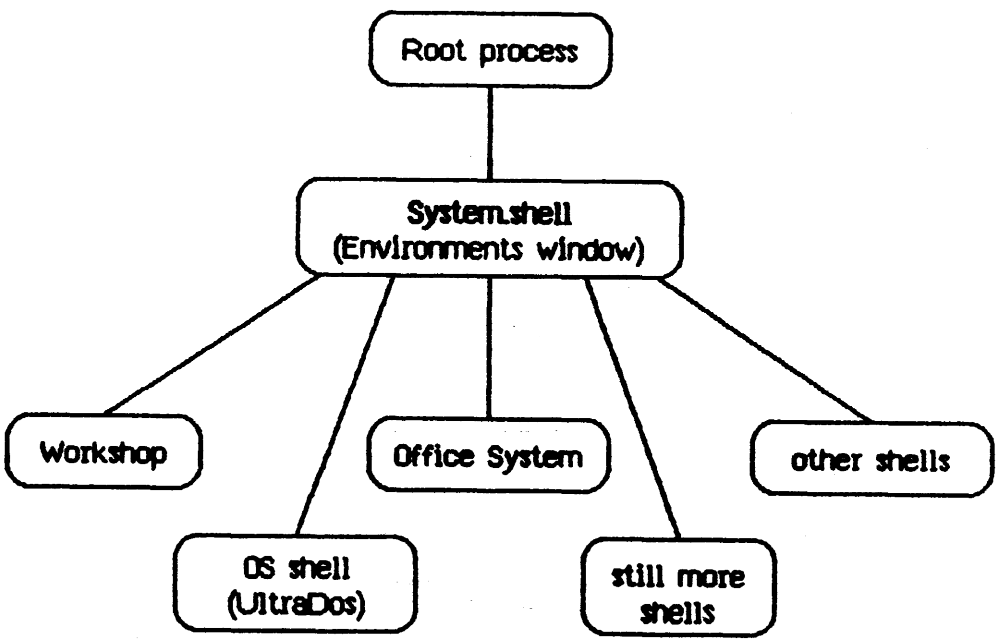


Figure 1
Process Picture

If your shell is the first process (the **system.shell**), you must make the following system initialization calls. Normally, the selector shell takes care of this for you.

Startup: **procedure BlockIOInit;** Initializes Pascal I/O. (Note: if you don't have the privileged PASLIB interface, declare **BlockIOInit** external.)

procedure PMinit (var error: integer); Initializes parameter memory. (Note: you have to be able to link against the **pmm** unit to make this call.)

function enableDbg (on: boolean): boolean; Activates LisaBug if you want to use it.

procedure setNMIkey (keyCap: integer); Makes LisaBug accessible through the NMI key.

Termination: **procedure BlockIODisInit;** PASLIB cleanup. (Note: if you don't have the privileged PASLIB interface, declare **BlockIOInit** external.)

To tell if your shell is the **system.shell**, call:

info_process (OSErr, My_Id, PInfo)

If **PInfo.father_id** is 1 (the root process), then you're in the **system.shell**.

The Environments window is the standard **system.shell**. It scans the directory of the startup disk for files whose names begin with '**shell.**'. For your shell to be recognized and available from the Environments window, the name of its object file must start with '**shell.**'.

Interprocess Communication

Event channels are used for communication between processes. The root process and the selector shell expect information from their son processes through a **SYS_SON_TERM** event channel, telling why the son terminated, and whether the father should restart the son, select or start another shell, turn the power off, or restart the machine. The OS guarantees that this event will always be sent back to the father of a terminated process via the local event channel, even if the son process was unwillingly aborted.

At Shell Startup

FATHER: A process that starts a shell must do the following:

1) Establish a local event channel to allow its son to communicate with it (**OPEN_EVENT_CHN**).
2) Start the son shell (**MAKE_PROCESS**).
3) Wait for a **SYS_SON_TERM** event (**WAIT_EVENT_CHN**).

SDN: The shell that was started must do the following:

1) Declare a **SYS_TERMINATE** exception handler (**DECLARE_EXCEP_HDL**).

   This exception will be signalled when the shell process is about to be terminated for any reason: because **KILL_PROCESS** or **TERMINATE_PROCESS** has been called; because the process ran to completion; because there has been a bus error, address error, illegal instruction, privilege violation, or line 1010 or 1111 emulator error.

   If this procedure is declared, the OS will *always* give it a chance to run before the process is terminated.

It is recommended that new shells not assume anything about the state of the machine (e.g. the console setting, etc.).

For more information on event channels and on starting up other processes from a shell, refer to the *Operating System Reference Manual.*

At Shell Termination

SDN: It is the shell's responsibility to make the operating system call to **TERMINATE_ PROCESS** to open an event channel, **s_eventblk** (an array of longints). The first entry of this block (**s_eventblk[1]**) contains the event that tells the shell's father what to do. The chosen meanings for these values are:

1--Restart same shell (shell crashed and needs to be restarted). To avoid infinite loops of START - CRASH - RESTART - CRASH..., the user will be able to intervene when the selector shell is reached.

2--Select another shell (**SELECT_ANOTHER** command).

3--Start the specified shell. The remaining longints in the event text block (**s_eventblk[2..9]**) are interpreted as a packed array [1..32] of characters (with no length field), containing the file name of the shell to be started. The unused portion of the array is packed with spaces.

4--Turn machine off (white power button clicked, or **POWER_OFF** command).

5--Reboot the machine.

other -- Unspecified.

It will be the job of the shell's terminate exception handler (which is just a procedure the shell owns) to guarantee that the proper **SYS_SDN_TERM** event text is set before the shell actually terminates. It can do this by calling **TERMINATE_PROCESS**, one of whose parameters is a pointer to this block.

FATHER: The father of the shell that just terminated should:

1) Reawaken because it has received the **SYS_SON_TERM** event via its local event channel.
2) Check the event text to see what to do.

Examples:

Following are code segments from both a father shell and a son shell showing the start-up and termination of the son.

These constant and type definitions are used throughout the following examples:

```
CONST
    aRestart = 1;             {Restart me }
    aSelectAnother = 2;       {Select another shell }
    aStartAnother = 3;        {Start the shell named in the event text }
    aOff = 4;                 {Turn off Lisa}
    aReset = 5;               {Reset the machine }

TYPE
   { this is a variant record which allows us to address the packed array of char }
   trix = RECORD CASE BOOLEAN OF
              TRUE:    (evblk: s_eventblk);
              FALSE:   (zeroth: longint;
                        first: longint;
                        rest: packed array [1..max_ename] of char;);
          END;  {trix}
```

FATHER: This code shows a father shell starting up a son shell and waiting for its termination.

```
PROCEDURE ShellLoop;

VAR OSerr: integer;
    procID: longint;
    fname: pathname;
    entry: namestring;
    nextToDo: integer;
    ex_name: t_ex_name;
    ev_chan_refnum: integer;
    ev_ch_name: pathname;
    WaitList: t_waitlist;
    ev_ptr: r_eventblk;

    PROCEDURE SelectShell(VAR fName: pathname);
    BEGIN
        WRITE('Next Shell ?');
        READLN(fname);
    END; {SelectShell}
```

```
    PROCEDURE StuffName(ev_blk: s_eventblk; VAR fname: pathname);
    VAR block: trix;
        i: INTEGER;
    BEGIN
        block.evblk := ev_blk;
        i := 1;
        fname := '';  {null string}
        WHILE i<=32 DO BEGIN
            IF fname[i] = ' ' {space} THEN BEGIN
                fname[0] := chr(i-1);  {stuff length field}
                EXIT(StuffName);
            END;  {IF}
            fname[i] := block.rest[i];
            i := i + 1;
        END;  {WHILE}
        fname[0] := chr(32);   {stuff length field}
    END;  {StuffName}

BEGIN {ShellLoop}

    entry := '';
    ev_ch_name := '';
    ex_name := ''
    Open_Event_Chn(OSerr,ev_ch_name,ev_chan_refnum,ex_name,receive);

    SelectShell(fname);

    REPEAT
        Make_Process(osErr,procID,fname,entry,ev_chan_refnum);
        IF (osErr <= 0) THEN BEGIN
            waitList.length := 1;
            waitList.refnum[0] := ev_chan_refnum;
            Wait_Event_Chn(osErr,waitList,which,@ev_ptr);

            {code for father shell bringing down son starts here}
            Kill_Process(osErr,procID);

            IF ev_ptr.event_text[0]=call_term THEN  {called terminate_process}
                NextToDo := ev_ptr.event_text[1]
            ELSE
                NextToDo := aSelectAnother;

        END; {made the process successfully}

        CASE NextToDo OF
            aRestart: {do nothing};
            aSelectAnother: SelectShell(fname);
            aStartAnother: StuffName(ev_ptr.event_text,fname); {get name of NextShell out of event_text}
            aOff: ShutDown(aOff);           {4--turn the machine off}
            aReset: ShutDown(aReset);       {5--reset the machine}
        OTHERWISE SelectShell;
    END;  {case NextToDo}

    UNTIL HellFreezesOver;
END; {ShellLoop}
```

SON: This procedure makes the necessary calls for the start-up of a shell.

```
PROCEDURE ShellInit;
VAR OSerr: INTEGER;
    PInfo: ProcInfoRec;

BEGIN
    info_process(OSerr,My_ID,PInfo);
    IF PInfo.father_ID = 1 {root} THEN BEGIN
        BlockIOinit;     {from PPasLibC}
        PMinit;          {from PMM}
        IF EnableDBG THEN SetNMIkey(33);  {standard NMI keycap}
    END;  {IF}
END;  {ShellInit}
```

This code shows the shutdown of a shell. If the **ShutDown** procedure is declared as the **Sys_Terminate** exception handler, it will properly communicate to its father its reason for terminating.

```
PROCEDURE ShutDown (why: INTEGER);
TYPE
VAR
   block: trix;          { the variant record }
   NextShell: e_name;
   i: INTEGER;           { for the for loop }
   OSerr: INTEGER;       { required parameter for the call to terminate_process }

BEGIN
    block.evblk[1] := why;
    IF why = aStartThisOne THEN BEGIN
        NextShell := 'shell.next';
        {copy string without length field}
        FOR i := 1 TO length(nextshell) DO block.rest[i] := nextshell[i];
        FOR i := length(nextshell) + 1 TO Max_ename DO block.rest[i] := ' ';
    END;
    terminate_process(OSerr,@block.evblk);
END;  {ShutDown}
```

******Exerpts from Fred's memo 4-23-84******

Summary

The Workshop's file manager has been extended to take advantage of some new features provided by the OS — password protection and hierarchical catalog structures. The file manager has been beefed up to allow convenient copy/backup/transfers onto more than one diskette (a more frequent occurance with Sonys).

The Details

File Manager

Changes to the File manager have revolved around three issues: the new OS hierarchical catalog structures, password protection, and backup to multiple volumes.

Following are details on how the various File Manager commands have changed.

- o **AddCatalog command.** The AddCatalog command allows you to create new catalogs. The pathname you specify for a catalog should refer to a volume which has been initalized with the new OS's B-tree file system structures. A catalog specified by a pathname without a volume part will be created with respect to the current main prefix.

  The **dash** is the catalog delimiter, so a file name referring to a file in a catalog might look like "-vol-cat-file" or "-vol-cat1-cat2-file", and so on. A file name of the form "cat-file" is interpreted relative to the current prefix and thus might refer to "-vol-cat-file" or "-vol-cat1-cat-file" depending on whether the current prefix was set to a volume or to a catalog.

  There is no special command to put a file in a catalog. Once a catalog has been created, newly created files will get put into it in two ways: (1) if the new file's name is specified by a full pathname with volume and catalog parts, in which case the file is put in the specified catalog (which must exist before a file can be put into it); and (2) if the new file's name does not have a volume part (i.e., it is a partial pathname) and the current prefix is to a catalog, in which case the file is created in the current catalog (or the appropriate sub-catalog if the new file's pathname includes a catalog part).

  Note that when the OS tries to find a file given a partial pathname, the file will be found only if (1) the pathname has no catalog part and is located in the current prefix volume or catalog, or (2) if the pathname has a catalog part that corresponds to a path starting with a catalog at the top level in the current prefix volume or catalog.

- o **Backup/Copy/Transfer to multiple diskettes.** The Backup, Copy and Transfer commands share a common file duplication mechanism that has been modified to allow backups (or whatever) to mutilple volumes. If a list of files is being copied to a diskette and you run out of space, you will be told what file didn't fit and how many more blocks were needed, and you will be asked whether you want to continue on another diskette. If you answer Yes you will be lead through a diskette change and the operation will continue. Note that the volume names of the subsequent disks need not match the first, even if the original destination was specified with a particular volume name as opposed to a generic device name.

- **List and Names commands.** Two new attributes are indicated for items in the List command display. The **D** attribute indicates a directory (C for catalog would have been nicer but was already in use for closed-by-OS) and the * attribute indicates a password protected file (see the next section).

  The List and Names commands will now indent names to show the catalog structure when listing B-Tree catalogs. The one exception to this case is when you do a "non-contiguous" or partial list, that is, when you use a wildcard specification with something to match following the wildcard character, causing only some of a contiguous subset of files to be listed. A wildcard specification of the form "<left pattern><wildcard char>" will select a "contiguous" subset of files matching <left pattern>, while a wildcard pattern of the form "<left pattern><wildcard char><right pattern>" will select only some of the set of files matching <left pattern>, resulting in a list with any number of discontinuities. Since a partial list is not assured of containing enough files to indicate the catalog hierarchy via indentation, the List and Names commands will print an unindented list of complete pathnames matching the wildcard specification.

  NOTE: In the past the Workshop has truncated file names in the displays of several commands (such as the List command which has a limited field in which to print the name, and commands like Copy which display "<source file> copied to <destination file>"). In some cases the names would be simply truncated and in others the last two characters would be replaced with two periods. The new Workshop should now indicate truncation by replacing the last character displayed with "..." (i.e., the ellipses character).

- **Prefix command.** Catalogs have changed the prefix command so that prefixes may now be to arbitrary catalogs in addition to volumes. Prefixes must be specified with complete pathnames; that is, if you are prefixing to a subcatalog, you must specify the complete path to the catalog.

  The effect of the current prefix on the interpretation of file names was discussed in the previous section.

  WARNING: Due to a recent change in the OS, the act of setting the main prefix (or working directory) has greater consequences than it used to. In particular, it may cause problems in running programs which use intrinsic units (this includes all the Workshop tools). The OS loader used to load a program's intrinsic libraries *from the boot volume* using the library names in INTRINSIC.LIB (which it makes a copy of at boot time). The library names used to be partial pathnames without a volume specification. Now the OS loader tries to find the libraries according to the pathnames it found in INTRINSIC.LIB, which means it will look on the prefix volume (or catalog) if the names in INTRINSIC.LIB were partial pathnames. There are two solutions to this problem: (1) copy the intrinsic.libraries to the prefix catalog, which could result in a proliferation of library files, or (2) change the names of the libraries in INTRINSIC.LIB to pathnames of the form "-#BOOT-libname", and then reboot so that the OS will cache the new names. The latter solution is the best in general, but requires tampering with INTRINSIC.LIB (which makes many people nervous, so I've written an exec file to do it ... see me if you're interested). The first solution points out the flexibility of the new scheme, that is, you may support several different library environments on the same volume via prefixing.

- o **AddPassword and RemovePassword commands.** The two new commands supporting password protection are found under the FileAttributes command. AddPassword allows you to password protect a file (or files via wildcards). Don't forget the password! RemovePassword allows you to remove passwords from files, but you must know the password to remove it.

  A key point to note about password protected files in the Workshop is that the Workshop tools will not be able to open a file once it is password protected, so passwords must be removed to make the files useable. Admittedly this is a less-than-optimal password protection scheme, but short of a major redesign of the file access methods of the Workshop and all its tools, it does provide reasonable protection at little expense.

- o **Initialize command (the new file system).** Although this command has not changed, it is useful to note that volumes initialized under the new Workshop and OS have a new structure (B-trees) which allows for hierarchical catalogs. Since these structures cannot be applied retroactively to old volumes, a device must be reinitialized in order to take adavantage of these features.

  The following fact may be of interest to speed freaks and Priam users. Since the names in a B-tree catalog are already sorted, the shell knows enough to not sort the files coming from B-tree volumes when performing file manager commands which operate on lists of files. This means, for example, that running the List command on a reinitialized Priam should be much faster than before, since the potentially very large list of files does not need to be sorted. Incidentally, the bubble sort of days of yore has been replaced with a Shell sort (aptly enough), which is many times faster, so life should be greatly improved even if you don't reinitialize your Priam.

- o **OnLine command.** The OnLine command has changed in one immediately obvious way -- the **new device names** used by the new OS. For the sake of convenience (to make the device names intelligible to humans) the OnLine command has been altered to also display the old device names which the new OS supports as device aliases. The point to note is that the aliases are no longer the real device names, so while the new names and aliases are accepted going into the OS, only the new names come back out.

  The OnLine command has been modified in another less obvious way. The prefix attribute (P) is now sometimes displayed in lower case (p). The uppercase **P** indicates that the main prefix is to the indicated volume, while a lowercase **p** indicates that the prefix is to a catalog somewhere on that volume.

  NOTE: It is possible to confuse the Online command into thinking that devices are configured that when they are not. A typical example is getting an error in the middle of the Online output which says that it could not find #11 (i.e., paraport) on a Pepsi. The problem is eliminated by using Preferences or the CDConfig program to detach the non-existant device. Similarly, instead of an error, you may find that the Workshop pauses unexpectedly in the middle of Online output. This problem is also caused by a device being configured but not present (the pause in the Online output is the device driver timing out while trying to access the device). The point to note is that the Online command no longer iterates through a fixed list of devices as it did before; instead, it must rely on the information supplied by the Parameter Memory manager (which is set when you run Preferences). So make sure that

Preferences' idea of how the system is configured is correct!

o **File Selection.** The File Manager uses a common mechanism for file selection for all of the commands which operate on lists of files (list, copy, delete, rename, etc.). Lists of files are specified via **wildcard patterns** against which file names are matched. These wildcard patterns have the general form:

<catalog part><left pattern><wildcard char><right pattern>

Various combinations of the wildcard pattern elements can be omitted.

The wildcard characters are "?" and "=". These will now operate on all files in a B-tree catalog and on any files in subcatalogs, that is, the wildcard matching mechanism will "go down into" subcatalogs as it attempts to find files satisfying the wildcard specification. New variants of "?" and "=" have been introduced to allow file selection to take place only on the top level of a B-tree catalog (without going into subcatalogs). The new variants are enabled by pressing the option key while typing "?' or "=", resulting in "¿" or "≠".

Please note (from the general form of a wildcard pattern given above) that wildcards are not permitted in the <catalog part> of a wildcard specification.

Apple Computer Inc.

Macintosh Division
Development Tools Clump ???

April 23, 1984

TO: Macintosh Software Engineering

FROM: Fred Forsman

SUBJECT: Workshop Enhancements for Spring Release
(or what's new in the old shell game)

A number of enhancements to the Workshop Shell have been implemented for the Spring release. The next section summarizes the changes, and the remainder of the memo the details.

Summary

The Workshop's file manager has been extended to take advantage of some new features provided by the OS -- password protection and hierarchical catalog structures. The file manager has been beefed up to allow convenient copy/backup/transfers onto more than one diskette (a more frequent occurance with Sonys). The resident process mechanism has been removed, having become obsolete with the new OS. A number of convenience features have been added, such as the remembering Run command. A unit has been provided for communication between programs and the shell or between cooperative programs.

Last, but not least, the Exec File processor has been extended so that it now provides a fairly powerful interpretive language for controlling development scripts. The usefulness of the exec processor has been greatly enhanced by converting it from a preprocessor into a truly interactive processor, by allowing it to stay present while Workshop commands are executed and programs are run so that the exec script can be resumed after the non-exec workshop commands have been executed. (Formerly all exec processing took place first and then the resulting script was run.) The exec language has been enhanced to include looping constructs, named variables, file I/O, a directory search capability, screen control functions, and functions to perform arithmetic operations. The performance has also be greatly improved, due in part to a new file caching mechanism. (A word of reassurance: your old exec files will work just as before, only faster.)

The Details

Note that the following description assumes knowledge of the Release 1.0 Workshop and Pepsi Workshop (virtually identical to 1.0 but with support for the new hardware).

Remembering Run Command

The Run command will remember what you ran last and offer it as a default. Even if you don't always want to run the same thing again, it serves as a convenient reminder of what you did last.

Run commands in exec files will not be remembered.

No More Resident Processes

Improvements in the OS have obviated the need for the Workshop's old resident-process mechanism (which would allow certain specified processes to be suspended rather than killed so that they could be rerun with less swapping).

As a result, the System manager's Process manager subsystem has been simplified by removing the commands to support the list of resident programs. (Note that the file **LDS_RES_PROCS.TEXT** that once saved this list between invocations of the Workshop is no longer used.) The process manager is still useful for monitoring and killing suspended and background processes.

Programs can still achieve the more interesting effects of residency (such as continuing from where they last were, as does the Mouse Editor) by suspending themselves. When the program is reinvoked, the shell will detect that a suspended instance of the process is still around and will reactivate it.

File Manager

Changes to the File manager have revolved around three issues: the new OS hierarchical catalog structures, password protection, and backup to multiple volumes.

NOTE: The discussion below assumes familiarity with the breakdown of pathnames into volume, catalog and filename components. The following examples of the various forms of valid pathnames should make the division into components clear. The possible forms of pathnames before catalogs were two:

-volname-filename	{ full pathname }
filname	{ partial pathname; no volume }

The new forms of pathnames now possible with catalogs are:

-volname-catname-filename	{ full with catalog }
-volname-catname-catname2-filename	{ full with catalogs }
catname-filename	{ partial with catalog(s) }

Following are details on how the various File manager commands have changed.

o **File Selection.** The File manager uses a common mechanism for file selection for all of the commands which operate on lists of files (list, copy, delete, rename, etc.). Lists of files are specified via **wildcard patterns** against which file names are matched. These wildcard patterns have the general form:

  <catalog part><left pattern><wildcard char><right pattern>

  Various combinations of the wildcard pattern elements can be omitted.

  The wildcard characters are "?" and "=". These will now operate on all files in a B-tree catalog and on any files in subcatalogs, that is, the wildcard matching mechanism will "go down into" subcatalogs as it attempts to find files satisfying the wildcard specification. New variants of "?" and "=" have been introduced to allow file selection to take place only on the top level of a B-tree catalog (without going into subcatalogs). The new variants are enabled by pressing the option key while typing "?' or "=", resulting in "¿" or "≠".

Please note (from the general form of a wildcard pattern given above) that wildcards are not permitted in the <catalog part> of a wildcard specification.

- **Initialize command (the new file system).** Although this command has not changed, it is useful to note that volumes initialized under the new Workshop and OS have a new structure (B-trees) which allows for hierarchical catalogs. Since these structures cannot be applied retroactively to old volumes, a device must be reinitialized in order to take adavantage of these features.

  The following fact may be of interest to speed freaks and Priam users. Since the names in a B-tree catalog are already sorted, the shell knows enough to not sort the files coming from B-tree volumes when performing file manager commands which operate on lists of files. This means, for example, that running the List command on a reinitialized Priam should be much faster than before, since the potentially very large list of files does not need to be sorted. Incidentally, the bubble sort of days of yore has been replaced with a Shell sort (aptly enough), which is many times faster, so life should be greatly improved even if you don't reinitialize your Priam.

- **AddCatalog command.** The AddCatalog command allows you to create new catalogs. The pathname you specify for a catalog should refer to a volume which has been initalized with the new OS's B-tree file system structures. A catalog specified by a pathname without a volume part will be created with respect to the current main prefix.

  The **dash** is the catalog delimiter, so a file name referring to a file in a catalog might look like "-vol-cat-file" or "-vol-cat1-cat2-file", and so on. A file name of the form "cat-file" is interpreted relative to the current prefix and thus might refer to "-vol-cat-file" or "-vol-cat1-cat-file" depending on whether the current prefix was set to a volume or to a catalog.

  There is no special command to put a file in a catalog. Once a catalog has been created, newly created files will get put into it in two ways: (1) if the new file's name is specified by a full pathname with volume and catalog parts, in which case the file is put in the specified catalog (which must exist before a file can be put into it); and (2) if the new file's name does not have a volume part (i.e., it is a partial pathname) and the current prefix is to a catalog, in which case the file is created in the current catalog (or the appropriate sub-catalog if the new file's pathname includes a catalog part).

Note that when the OS tries to find a file given a partial pathname, the file will be found only if (1) the pathname has no catalog part and is located in the current prefix volume or catalog, or (2) if the pathname has a catalog part that corresponds to a path starting with a catalog at the top level in the current prefix volume or catalog.

- o **Prefix command**. Catalogs have changed the prefix command so that prefixes may now be to arbitrary catalogs in addition to volumes. Prefixes must be specified with complete pathnames; that is, if you are prefixing to a subcatalog, you must specify the complete path to the catalog.

  The effect of the current prefix on the interpretation of file names was discussed in the previous section.

  WARNING: Due to a recent change in the OS, the act of setting the main prefix (or working directory) has greater consequences than it used to. In particular, it may cause problems in running programs with use intrinsic units (this includes all the Workshop tools). The OS loader used to load a program's intrinsic libraries *from the boot volume* using the library names in INTRINSIC.LIB (which it makes a copy of at boot time). The library names used to be partial pathnames without a volume specification. Now the OS loader tries to find the libraries according to the pathnames it found in INTRINSIC.LIB, which means it will look on the prefix volume (or catalog) if the names in INTRINSIC.LIB were partial pathnames. There are two solutions to this problem: (1) copy the intrinsic.libraries to the prefix catalog, which could result in a proliferation of library files, or (2) change the names of the libraries in INTRINSIC.LIB to pathnames of the form "-#BOOT-libname", and then reboot so that the OS will cache the new names. The latter solution is the best in general, but requires tampering with INTRINSIC.LIB (which makes many people nervous, so I've written an exec file to do it ... see me if you're interested). The first solution points out the flexibility of the new scheme, that is, you may support several different library environments on the same volume via prefixing.

- o **OnLine command**. The OnLine command has changed in one immediately obvious way -- the **new device names** used by the new OS. For the sake of convenience (to make the device names intelligible to humans) the OnLine command has been altered to also display the old device names which the new OS supports as device aliases. The point to note is that the aliases are no longer the real device names, so while the new names and aliases are accepted going into the OS, only the new names come back out.

  The OnLine command has been modified in another less obvious way. The prefix attribute (P) is now sometimes displayed in lower case (p). The uppercase **P** indicates that the main prefix is to the indicated volume, while a lowercase **p** indicates that the prefix is to a catalog somewhere on that volume.

  NOTE: It is possible to confuse the Online command into thinking that devices are configured that when they are not. A typical example is getting an error in the middle of the Online output which says that it could not find #11 (i.e., paraport) on a Pepsi. The problem is eliminated by using Preferences or the CDConfig program to detach the non-existant device. Similarly, instead of an error, you may find that the Workshop pauses unexpectedly in the middle of Online output. This problem is also caused by a device being configured but not present (the pause in the Online output is the device driver timing out while trying to access the device). The point to note is that the Online command no longer iterates through a fixed list of devices as it did before; instead, it must rely on the information supplied by the Parameter Memory manager (which is set when you run Preferences). So make sure that Preferences' idea of how the system is configured is correct!

o **List and Names commands**. Two new attributes are indicated for items in the List command display. The D attribute indicates a directory (C for catalog would have been nicer but was already in use for closed-by-OS) and the * attribute indicates a password protected file (see the next section).

  The List and Names commands will now indent names to show the catalog structure when listing B-Tree catalogs. The one exception to this case is when you do a "non-contiguous" or partial list, that is, when you use a wildcard specification with something to match following the wildcard character, causing only some of a contiguous subset of files to be listed. A wildcard specification of the form "<left pattern><wildcard char>" will select a "contiguous" subset of files matching <left pattern>, while a wildcard pattern of the form "<left pattern><wildcard char><right pattern>" will select only some of the set of files matching <left pattern>, resulting in a list with any number of discontinuities. Since a partial list is not assured of containing enough files to indicate the catalog hierarchy via indentation, the List and Names commands will print an unindented list of complete pathnames matching the wildcard specification.

  NOTE: In the past the Workshop has truncated file names in the displays of several commands (such as the List command which has a limited field in which to print the name, and commands like Copy which display "<source file> copied to <destination file>"). In some cases the names would be simply truncated and in others the last two characters would be replaced with two periods. The new Workshop should now indicate truncation by replacing the last character displayed with "..." (i.e., the ellipses character).

o **AddPassword and RemovePassword commands**. The two new commands supporting password protection are found under the FileAttributes command. AddPassword allows you to password protect a file (or files via wildcards). Don't forget the password! RemovePassword allows you to remove passwords from files, but you must know the password to remove it.

  A key point to note about password protected files in the

Workshop is that the Workshop tools will not be able to open a file once it is password protected, so passwords must be removed to make the files useable. Admittedly this is a less-than-optimal password protection scheme, but short of a major redesign of the file access methods of the Workshop and all its tools, it does provide reasonable protection at little expense.

- o **Delete Command.** Those of you who look closely at the behavior of the Delete command operating on B-tree catalogs may notice a new wrinkle in the command's operation. While all the other File manager commands perform their operations on an alphabetically sorted list of files, the Delete command must delay the deletions of catalogs which are not yet empty. Thus the Delete commands works in two passes: in the first pass all files are deleted in alphabetical order, as are catalogs which are empty; in the second pass, any catalogs not deleted in the first pass are now deleted in *reverse* alphabetical order (to take care of catalogs contained in other catalogs).
- o **Backup/Copy/Transfer to multiple diskettes.** The Backup, Copy and Transfer commands share a common file duplication mechanism that has been modified to allow backups (or whatever) to mutilple volumes. If a list of files is being copied to a diskette and you run out of space, you will be told what file didn't fit and how many more blocks were needed, and you will be asked whether you want to continue on another diskette. If you answer Yes you will be lead through a diskette change and the operation will continue. Note that the volume names of the subsequent disks need not match the first, even if the original destination was specified with a particular volume name as opposed to a generic device name.

Program-Shell Communication

An intrinsic unit (**ProgComm**) has been added to SULib which allows programs to communicate with the shell and with other programs. Three basic mechanisms are provided.

- o **Set Next Run Command.** A programmatic call is provided which allows a program to tell the Workshop shell what to run next. The specified program will be run next (after the current program is done), taking precedence even over an exec file in progress.
- o **The Program Return String.** A string is provided which can be set programmatically and which can be accessed from the exec processor (via the RETSTR function). This allows exec scripts to be written which make choices based on program results.
- o **The Communication Buffer.** A 1K byte buffer (global to the Workshop) has been provided for communication between programs. The buffer can be used in any number of ways; however, a set of primitives supporting character and line-oriented I/O to and from the buffer is provided.

More detailed information of the program communication unit can be found in the ProgComm appendix to this document.

Note that the above mechanisms can be used in conjunction with each other. For example, a program could write a series of invocation arguments to the communication buffer and then tell the shell to run a particular program next (via the set-next-run command). That program could then know to check the communication buffer to find its arguments. (In general, programs might be written so that they check the communication buffer for their arguments first and prompt for arguments from the console only if the arguments are not found in the buffer).

ProgComm's program-program communication facility has been used by several of the Spring release Workshop tools:

- **Compiler-Generator communication.** The Pascal compiler will now automatically invoke the Generator to perform the second step of the compilation process. This behavior can be suppressed by specifying the "$G-" option in response to the compiler input prompt. The third compiler prompt is now for a .OBJ output file rather than a .I output file (although a .I is generated when the generator is called automatically).

  NOTE: The above change will probably mean that you will have to change your "Compile" exec file (either to eliminate the generate step or to use the $G- option). If you haven't been using a common compile exec file, then you probably have more editing in store.

- **Compiler-Editor communication.** The compiler now provides the option of going to the editor in the event of a compilation error (the choices offered by the error prompt are SPACE to continue, CLEAR to escape, and E to go to the editor). If you go to the editor the point of error will be displayed in the appropriate source file and the compiler error message will be displayed.

Exec Files

Major extensions have been made to the Exec File processor, as enumerated below:

- **Alternate "$" convention.** Now that the exec command language is filling out, you can create meaningful exec files with many more exec command lines than workshop (non-exec) command lines. Up until now these two types of lines have been distinguished by a "$" as the first significant character of exec lines. As a consequence, exec files consisting of mostly exec command lines become unreadable or annoying with all of the dollar signs, which is unfortunate since the dollar signs mess up the lines which are inherently more readable and intelligible.

  Now exec files which begin with "EXEC" rather than "$ EXEC" will be accepted and processed with the "$" convention reversed, that is, workshop lines would then begin with a dollar and exec lines would not. This makes exec files consisting of mainly exec commands look more normal and readable, and in no way affects files written using the other convention. In fact the two conventions can be mixed, that is, a file written in one convention can call a file written in the other convention. In the new convention, workshop lines begin immediately following the "$" (although leading and trailing blanks will be removed unless the "B" option is in effect).

- **Named parameters and variables**. Names can now be associated with the %n variables, allowing meaningful names to be used to make exec files more readable and intelligible. Parameters can still be referred to in the old "%n" fashion, or they can be referred to with new names, or both. The names are declared (associated positionally with the "%n" parameters) by having an optional parenthesized list of names on the exec command line, as in

```
EXEC (volName, fileName)
  IF UPPERCASE (volName) = '-PARAPORT' THEN
  <etc.>
ENDEXEC
```

The parameter names as specified on the EXEC command line must begin with an alphabetic character, may include subsequent alphabetic and numeric characters, and may be as long as you like, although only the first eight characters are significant (as in Pascal). The parameter list is not allowed to have "holes" in it, that is, you **cannot** do something like:

```
EXEC (pName0, , pName2)
```

Once the names are declared on the EXEC line, named parameters can then be used as you would expect in exec lines (see "volName" in the second line of the example above). In workshop (non-exec) lines the name should be surrounded by square braces so that it can be distinguished from the surrounding text as in:

```
EXEC (file)
  $F{filer}D{delete}[file]
  $Y{yes}Q{Quit}
  <etc.>
ENDEXEC
```

The rule is that square braces are required to offset a parameter name in contexts where processing is done in a text-oriented mode (i.e., when in workshop as opposed to exec lines). Otherwise, the names cannot cannot be distinguished (from the exec processor's point of view) from the text in which they appear. Note that [...] constructs in non-exec lines will be copied into the temporary file as is if the stuff between the braces is not recognized as a parameter name.

Symbolic names must also be enclosed in square braces in order to be recognized in SUBMIT commands and in function calls. This is required since SUBMIT and function arguments lists are scanned as if the arguments were pure text instead of string expressions. (This form of argument scanning was chosen to be compatible with the scanning of arguments on the exec invocation line. Unfortunately, this is one area that cannot be cleaned up without breaking everyone's exec files, or else by introducing alternate versions of SUBMIT and function calls that take string expression arguments.) The following example demonstrates situations in which a name does and does not need to be enclosed in square braces.

```
EXEC (file)
  $F{filer}C{copy}[file]
  $-lower-backup/[file]        { name with braces }
  IF file <> '' THEN           { name without braces }
    SUBMIT compile ([file])    { name with braces }
  <etc.>
ENDEXEC
```

The scope of names is the body of the defining exec file. Up-level name references are not allowed, that is, name references are always local (as they were before).

o **WHILE and REPEAT commands**. These commands allow for repetition of command sequences under the control of an arbitrary boolean condition. The syntax for the WHILE command is as follows:

```
WHILE <boolean expr> DO
  <arbitrary stuff>
ENDWHILE
```

The behavior of the WHILE construct is the same as the comparable Pascal construct. The <boolean expr> may be a condition of arbitrary complexity. The <arbitrary stuff> between the WHILE and the ENDWHILE may be anything: exec commands (including nested WHILEs) or Workshop command lines.

Similarly, the REPEAT command syntax is:

```
REPEAT
  <arbitrary stuff>
UNTIL <boolean expr>
```

o **RESETCAT command and NEXTFILE function**. These allow an exec file to get files from an OS directory (based on a wildcard pattern if desired). These new constructs are illustrated in the following example:

```
EXEC (file)
  RESETCAT '-paraport-=.text'
  REPEAT
    SET file TO NEXTFILE
    <whatever>
  UNTIL file = ''
ENDEXEC
```

For those of you familiar with the OS calls, RESETCAT is comparable to RESET_CATALOG and NEXTFILE is comparable to GET_NEXT_ENTRY. The RESETCAT command takes a <string expression> argument which specifies the directory and the search pattern (if any). If a filename part is specified in addition to a volume name, the filename part will be used as a search pattern for subsequent calls to the NEXTFILE function. If the wildcard character (=) is present standard wildcard matching takes place. If there is a filename part but no wildcard, the file name part is used as a search prefix (that is, "RESETCAT 'foo'" is equivalent to "RESETCAT 'foo='"). The NEXTFILE function returns an empty string when there are no more entries in the directory. The RESETCAT command also has the side effect of setting the value of the IORESULT function described below.

o **IORESULT function**. This works in conjunction with the RESETCAT command and the NEXTFILE function, indicating whether an error occured in the operation (similar to the IORESULT function in Pascal). IORESULT returns the empty string if no error occurred in the last significant operation (RESETCAT, NEXTFILE, OPENIN, OPENOUT). If an error occured, then a string with the error number, and the appropriate textual message is returned. An example:

```
EXEC (dir, ioErr)
  REPEAT
    REQUEST dir WITH 'Search what directory ?'
    RESETCAT dir
    IF IORESULT = '' THEN { successful RESETCAT }
      <search directory, etc.>
    ELSE                  { unsuccessful RESETCAT }
      SET ioErr TO IORESULT
      WRITELN 'Bad directory specification'
      WRITELN 'OS error: ', ioErr
    ENDIF
  UNTIL FALSE
  <etc.>
ENDEXEC
```

o **HALT and ABORT commands**. These commands stop the exec processor; the difference between HALT and ABORT is whether any accumlated Workshop commands will be processed. The HALT command will stop exec processing and will execute the commands that have been sent so far to the intermediate file. The ABORT command will stop exec processing and will not execute any accumlated commands. In a nut shell, if something really goes wrong you probably want to ABORT; if you have valid commands generated but not executed and you want to stop exec processing but still execute the queued commands, you probably want to HALT.

Both commands take an optional "string expression" argument which will be printed to the console (replacing an "Exec processing aborted." message in the case of the ABORT command).

- **EVAL function and numeric expressions**. The EVAL function is used to evaluate arithmetic expressions, returning a string containing the result of the evaluation. While the exec language still deals only with objects which are strings, this feature introduces the capability of dealing with a string as a number. The syntax of the EVAL function is

  ```
  EVAL ( <numeric expression> )
  ```

  where <numeric expression> is your usual arithmetic expression allowing the +, -, *, /, MOD and (...) operators. The numeric elements can be supplied via unquoted numeric constants (decimal only), parameters or variables (with string values which must be numeric constants), string functions returning numeric string values, or functions which return numeric string values such as LENGTH, ORD, and POS.

  It is important to keep in mind the differences between numeric and string expressions. You should also be aware of the contexts in which each is required. For example, you should understand why "EVAL(1)" is valid and "EVAL('1') is not.

  Observe that the result type of the EVAL function is a string (not a number, not a numeric string, just a string). The point to keep in mind is that all data objects in the exec processor are still strings. Only within the context of a <numeric expression> are strings treated as numbers.

  Arithmetic is done with LONGINTs with no overflow detection except when numeric constants are too large.

  Following is an example of a loop using a counter:

  ```
  SET N TO '0'{note 0 is expressed as a string constant}
  WHILE N <> '10' DO
    <whatever>
    SET N TO EVAL (N + 1)
  ENDWHILE
  ```

- **More string functions: LENGTH, COPY, POS, LOWERCASE, CHR, and ORD**. A number of new string funtions have been added. Some of these take advantage of the numeric expression capability introduced by the EVAL function. Note that some of the functions may be used in numeric expressions (since they return strings with numbers) in addition to string expressions.

  LENGTH (<str expr>)
  LENGTH takes a string expression argument and returns a string with a number in it. LENGTH may be used in both string and numeric expressions.

COPY (<str expr>, <num expr>, <num expr>)
COPY takes three arguments: a string expression and two numeric expressions. It returns the appropriate substring of the first argument, as in PASCAL with the exception that if the third argument is too large it will return what is available rather than the empty string. COPY can be used in string expressions but not numeric expressions (since it typically does not return a number). Keep in mind the differences between the two types of arguments taken by the copy function -- string and numeric expressions. An example:

```
EXEC (foo, n, ch)
  SET n TO LENGTH(foo)
  SET ch to COPY(foo,1,1)     {ch := first char of foo}
  SET foo TO COPY(foo,n/2,n) {foo := last half of foo}
  <etc.>
```

POS (<str expr>, <str expr>)
POS takes two string expression arguments, and returns a string with a number in it. The number is the position of the first occurrance of the first string within the second. If the first string does not appear in the second '0' is returned. POS may be used in both string and numeric expressions.

LOWERCASE (<str expr>)
LOWERCASE takes a single string expression argument and returns that string lowercased. We have UPPERCASE already so it seemed only fair to give equal time to lowercase.

CHR (<num expr>)
CHR takes a numeric expression and returns a one-character string with the character value corresponding to the numeric value MOD 255.

ORD (<str expr>)
ORD takes a string expression argument. An exec-time error will be generated if the string does not have a length of one. ORD returns a string with a number representing the integer value of the character. ORD may be used in both string and numeric expressions.

o **New string comparison operators**. Previously only the = and <> string comparison operators were supported. To this the >, <, >=, and <= operators have been added. These all function in the expected way. Now for the confusing part. Since the EVAL function has introduced strings which function as numbers, we need operators which compare strings as if they were numbers (instead of as strings). The new numerical string compare operators are EQ, NE, LT, GT, LE, and GE. For example, try comparing 9 and 16 with the following exec procedure.

```
EXEC (n1, n2)
 IF n1 > n2 THEN
```

```
 WRITELN n1,' is alphabetically greater than ',n2
ELSE
 WRITELN n1,' is not alphabetically greater than ',n2
ENDIF
IF n1 GT n2 THEN
 WRITELN n1,' is numerically greater than ',n2
ELSE
 WRITELN n1,' is not numerically greater than ',n2
ENDIF
ENDEXEC
```

- **TRUE and FALSE constants** in boolean expressions. Just as you would expect. Useful for "WHILE TRUE DO" and similar constructs.

- **Screen control commands: GOTOXY, CLEAR and CURSOR**. A number of commands have been added to allow screen-oriented exec procedures

  GOTOXY <num expr>, <num expr>

  GOTOXY takes two numeric expression arguments separated by a comma. The behavior is the same as Pascal's GOTOXY. Values which are beyond the upper or lower limits for coordinates will peg at the limit.

  CLEAR <clear option>

  CLEAR takes a <clear option> (SCREEN, ENDSCREEN, and ENDLINE) as an argument. SCREEN will clear the screen and leave the cursor at <0,0>. ENDSCREEN will clear to the end of the screen from the current cursor position. ENDLINE will clear to the end of the line from the current cursor position.

  CURSOR <cursor option> [<num expr>]

  CURSOR takes a <cursor option> (HOME, UP, DOWN, RIGHT, LEFT) as an argument, followed by an optional numeric expression. The results of the various cursor options should be obvious, and the optinal numeric expression can be used to supply a repetition count.

- **File I/O: RESET, REWRITE, and CLOSE commands**. The current READs and WRITEs have been extended to work with files in addition to the console. In order to support this new functionality three new commands have been introduced for opening and closing files. Note that these file-oriented commands work only on text files.

  RESET <id> , <str expr>

  RESET opens a file for input. An <id> (an identifier, as in Pascal, with only the first eight characters being significant) is used to establish a file variable (which is used to identify the file for subsequent reads, writes, and closes). The RESET command serves as a dynamic declaration of the file variable which becomes known globally for the duration of exec processing or until the file is CLOSEd. The string expression argument is used to specify the pathname for the file. The value of the IORESULT function will be set appropriately after the operation.

REWRITE <id> , <str expr>

REWRITE opens a file for output and is otherwise like the RESET command.

CLOSE <id>

CLOSE closes the file associated with the file variable, and causes the file variable to be deallocated.

The READCH, READLN, WRITE, and WRITELN commands have been extended to deal with files by adding an optional file specifier. The form of the file specifier is:

(<id>)

where <id> is a file variable. The file specifier should follow the command keyword, preceeding the normal command arguments, as in the following examples:

```
READCH (inFile) Char
READLN (inFile) Line
WRITE (outFile) 'This is a test: ', message, '.'
WRITELN (outFile) { write a CR }
```

o **I/O to the ProgComm Communication Buffer**. The I/O commands defined in the previous section (REWRITE, RESET, CLOSE, READCH, READLN, WRITE, and WRITELN) can also be used to write to or read from the communications buffer provided by the ProgComm unit (see the appendix on ProgComm). There is a predefined <id> -- 'CommBufr' -- which serves as a pseudo-file identifier for the communications buffer. With the exception of CLOSE, all of the I/O commands are the same are the same as the file-oriented forms, as in the following examples:

```
RESET CommBufr, 'key'
REWRITE CommBufr, 'key'
READCH (CommBufr) Char
READLN (CommBufr) Line
WRITE (CommBufr) 'This is a test: ', message, '.'
WRITELN (CommBufr) { write a CR }
```

Note that when openning the communication buffer, the second arguments of RESET and REWRITE are the **access key** instead of a file name. CLOSE is syntactically different in that it also requires a second argument specifying an access key, as in:

```
CLOSE CommBufr, 'key'
```

WARNING: CLOSE on the CommBufr has the effect of flushing the CommBufr. Consequently, **CLOSE should not be called after writing to the CommBufr**. It should be called after reading if the buffer is not intended to be read by somebody else, and it should be called when you want flush the buffer. Note that the CLOSE will only succeed if you specify right key or if the buffer was not keyed, thus a CLOSE with a key is in effect a conditional flush of the buffer. An unconditional flush can be achieved with a REWRITE, which always clobbers the buffer, regardless of the key.

o **DOIT command**. The DOIT command transforms the exec processor into more than just a preprocessor. When a DOIT is encountered all commands that have accumulated in the exec temporary file will be executed and then control will return to the current exec file following the DOIT (with the temporary file emptied). This allows you to execute Workshop commands and to run programs from an exec file and then to base further exec processing on the results of these commands. The concept is simple, yet powerful. A trivial example of something you could not do before is print a message after some workshop commands in an exec file have executed, as in:

```
EXEC (fromVol, toVol)
  WRITELN 'Now starting backup ...'
  $F{filer}B{backup}[fromVol]-=,[toVol]-$
  $Q{quit the filer}
  DOIT
  WRITELN 'Backup of ', fromVol, ' to ',
          toVol, ' completed'
ENDEXEC
```

One point to note about the DOIT command is that it causes immediate execution of what has accumulated in the temporary file, which you may find surprising initially if you are stepping through an exec file via the "S" option. As a result, the accumulated commands will be executed and then you will return to stepping following the DOIT.

o **RUN command**. The RUN command allows a program to be run immediately from an exec file without affecting commands being accumulated in the temporary file. The simplest form of the RUN command is:

```
RUN  <str expr>
```

where the <str expr> gives the pathname of the program to run. Note the RUN exec command gets executed immediately at exec time, whereas an embedded Workshop "R" command will get executed at run time.

Since programs often require input from the console, the following form of the RUN command is provided:

```
RUN  <str expr>  INPUT
  <arbitrary stuff>
ENDRUN
```

Here the "stuff" between INPUT and ENDRUN is put into **another** temporary file to use as exec input while the program is being run. This "stuff" will not affect any commands accumulating in the normal temporary file. If the program being run requires more input that provided by the "stuff", input will revert to the console to complete the program's input requirements. If too much "stuff" is provided, the excess will be ignored.

- **The RETSTR function**. The RETSTR function returns what is in the ProgComm unit's return string. Thus a return string set by a program using the ProgComm unit can be accessed from an exec file. For example:

```
EXEC
  RUN 'foo'
  IF RETSTR <> 'SUCCESS' THEN
    ABORT 'Foo failed'
ENDEXEC
```

- **The "G" invocation option**. The "G" (or generate only) invocation option allows you to test out your exec files without actually running them. Note that the "G" option disables the DOIT and RUN commands.

- **The "E" invocation option**. The "E" (or continue even with errors) invocation option allows you to run exec files which run workshop programs which have errors which would normally stop exec file execution. When running under this option, run-time errors will not stop exec processing. In using this option you run a higher-than-normal risk of your exec file becoming out-of-synch and doing things you did not intend. But the option can be very useful if you must run test suites which contain errors.

- **The "K" invocation option (formerly "T")**. The old "T" option, indicating that the generated temporary file should be saved rather than deleted after being run, has been renamed to "K" for Keep. This change was made because the new documentation for exec files (which will appear someday) does not refer to the generated file as a temporary file, so the "T" no longer makes any sense (not that it was a good choice for an option name in the first place).

- **Improved performance and file caching**. A file caching mechanism has been added to the exec processor. The cache currently consists of 5 pages (where a page is two blocks). The caching mechanism can cache 5 small files at at time where "small" is defined as having a listed size of 4 blocks (1 header page and 1 page of significant text). Small files will be put in the cache, and subsequent SUBMITs or function calls to that file will be read from the cache. The cache is maintained on a LRU (least recently used) basis. This means, for example, that if you call a sub-exec file to compile many times from a build exec file, the compile exec file will typically only be read once.

  To further boost performance the exec processor's handling of text files now goes through a unit developed by Ira Ruben (IOPrimitives in SULib).

These changes, along with numerous other tweaks to low-level routines in the exec processor, have resulted in more than doubling (sometimes tripling) of the exec processor's speed (although you may find the performance to be better or worse than this depending on road conditions and how your exec files are structured).

Appendix 1

ProgComm: the Program Communication Unit

Summary

An intrinsic unit (**ProgComm**) has been added to SULib which allows programs to communicate with the shell and with other programs. Three basic mechanisms are provided.

- o **Set Next Run Command.** A procedure is provided which allows a program to tell the Workshop shell what to run next. The specified program will be run next (after the current program is done), taking precedence even over an exec file in progress.
- o **The Program Return String.** A string is provided which can be set programmatically and which can be accessed from the exec processor (via the RETSTR function). This allows exec scripts to be written which make choices based on program results.
- o **The Communication Buffer.** A 1K byte buffer (global to the Workshop) has been provided for communication between programs. The buffer can be used in any number of ways; however, a set of primitives supporting character and line-oriented I/O to and from the buffer is provided.

Note that the above mechanisms can be used in conjunction with each other. For example, a program could write a series of invocation arguments to the communication buffer and then tell the shell to run a particular program next (via the set-next-run command). That program could then know to check the communication buffer to find its arguments. (In general, programs might be written so that they check the communication buffer for their arguments first and prompt for arguments from the console only if the arguments are not found in the buffer).

The Details.

The following describes the interface to the ProgComm unit. The following procedure initializes the ProgComm unit so that a program may use it.

PROCEDURE PCInit;
PCInit should be called before using the ProgComm unit. One effect of note is that the program's return string (RETSTR in the exec language) is initialized to the null string.

The following two procedures give a program the ability to set what program will run next and to pass back a return string to the exec processor. Note that the SUStr type comes from the "standard unit" -- StdUnit in SULib -- which provides, among other things, a number of string manipulation routines.

PROCEDURE PCSetRunCmd (RC : SUStr);
PCSetRunCmd enables a program to tell the shell what

program (or exec file) to run after the current program terminates, which allows program "chaining". RC, the run command you pass to PCSetRunCmd, should be a string with the same program pathname or exec file invocation you would give in response to the Workshop Run command prompt. The run command set in this way will take precedence over any keyboard type-ahead and over any pending exec file commands.

There is an added complication when you want to use PCSetRunCmd to run a Workshop tool that is normally invoked from the Workshop menu line. (Note that only some of items in the Workshop menu are actually separate tools which can be "run".) The complication arises from the fact that typing 'E' to invoke the editor is not always the same as typing 'R' for run and specifying 'editor.obj' as the program to run. The difference is that the Run command will look for 'editor.obj' using the three level of prefixes, while the 'E' menu command will look on the Workshop boot volume first and then at the three prefix volumes. If you want to get the effect of the menu command, your argument to PCSetRunCmd should be a two character string with an escape (CHR(27)) as the first character and the appropriate menu command as the second character.

Another subtlety, which you are unlikely to run into unless you are doing tricky things with exec files, is that starting to run an exec file while you are already running another exec file will cause the first exec file to be terminated in order to allow the second to be run. This means that if you run program P from exec file A, and P calls PCSetRunCmd to run exec file B, then, when program P terminates, exec file A will also be terminated so that exec file B can be run. Exec file A will not be resumed when exec file B has completed. This is another instance of the "exec file chaining" effect.

PROCEDURE PCSetRetStr (RS : SUStr);

PCSetRetStr allows a program to set a return string which may be accessed via the exec processor's RETSTR function. This allows exec files to make choices based on information passed back to the shell by cooperating programs. How the return string should be used and interpreted is up to you, and will depend on what sort of information you want to pass back to the exec processor. (But in order to be a good citizen it is probably best to follow whatever system-wide conventions emerge and prevail.)

The following procedures and functions operate on the **communication buffer**, which is a 1K byte buffer which is global to the Workshop shell (that is, it stays around between program invocations). The buffer can hold essentially any type of information, but a standard set of functions is provided for Pascal-like character or line-oriented access to the buffer.

Following are some CONST, TYPE, and VAR declarations from the

ProgComm interface which relate to the communication buffer.

```
CONST
  { communication buffer content types }
  PCNone    = -1;   { nothing in buffer }
  PCAny     = 0;    { for PCReset to match any content type }
  PCText    = 1;    { text, as supported by PCGets & PCPuts }
  PCBufrMax = 1023; { max buffer index, ie, bufr is 1K bytes }
TYPE
  PCBufrP   = ^PCBufr; { ptr to bufr }
  PCBufr    = PACKED ARRAY [0..PCBufrMax] OF CHAR;
VAR
  PCBufrPtr : PCBufrP; { points to bufr after successful open }
```

The communication buffer is given a **type** when it is opened for writing with PCReWrite. This type will be used to determine whether a potential reader trying to open the buffer with PCReset will be successful. The intent is to prevent reading of the buffer when the contents are not of the type expected by the reader. Three predefined constants are provided for buffer typing: **PCNone** means that the buffer has no contents; **PCText** means that the buffer contains standard text with CR line delimiters; and **PCAny** matches any type, allowing a reader to override the typing mechanism. Other buffer content types (such as mouse events) may be defined users, choosing some number to identify the new type which does not conflict with the predefined types. We make no attempt here to provide a complete set of predefined types; the issue is simply one of having compatible conventions (agreement) between communicating programs. To use the buffer for something other than text, the variable **PCBufrPtr** may be used to access the buffer (using whatever means of interpretation of the buffer is desired).

The buffer also has an **access key**, which functions in very much the same way as the content type (i.e., writers set it and readers must match it to gain access to the buffer). The intent of the access key is to prevent programs from reading the buffer when they are not the intended recipient. The access key, again, is something that should be established by agreement between the communicating programs. If a buffer writer does not care about preventing unintended access to the buffer, the null string can be used for the access key. Note that the access key is case sensitive.

Following are the procedures and functions which open and close the communication buffer.

PROCEDURE PCReWrite (WriteType: INTEGER; Key: SUStr);

PCReWrite opens the communication buffer for writing. The content type and access key are set. PCBufrPtr is set to point to start of the communication buffer. A PCReWrite will override any previous use of the buffer, i.e., it will flush any previous buffer contents. **WriteType** should be an integer identifying the type of data you plan to write to the buffer. If you are planning to use the text-oriented primitives provided, WriteType should be PCText; otherwise, WriteType should be some integer established by agreement between the communicating programs. **Key** should be a string also established by agreement between the communicating programs. A useful form of key is one that identifies the intended recipient, so that things that get left in the buffer do not get read inadvertently by programs for which they were not intended.

FUNCTION PCReset (ReadType: INTEGER; Key: SUStr): BOOLEAN;
PCReset opens the buffer for reading. The boolean result will indicate whether the open was successful. The open will fail if **ReadType** does not match the type set by the last buffer writer or if Key does not match the key set by the last writer.

FUNCTION PCClose (KillBufr: BOOLEAN; Key: SUStr): BOOLEAN;
PCClose will close (or empty) the communication buffer. If **KillBufr** is true the buffer will be emptied. In general, the buffer can be read more than once (by multiple readers) if desired. If a reader is finished with the buffer and knows that no one else should read the buffer, PCClose should be called with KillBufr set to true. The call to PCClose will fail if the access key does not match. Note that PCClose may be used to flush buffers that were written by someone else, as long as you know the access key. PCClose may be called without calling PCReset or PCReWrite first.

The following functions provide a text-oriented buffer facility with Pascal-like character and line-oriented reads and writes.

FUNCTION PCPutCh (Ch: CHAR): BOOLEAN;
PCPutCh will put a character into the buffer. The boolean result will indicate whether the operation was successful. It will fail if the buffer is full or if the buffer was never opened successfully for writing. Note that PCPutCh(CR) is equivalent to PCPutLine('').

FUNCTION PCGetCh (VAR Ch: CHAR): BOOLEAN;
PCGetCh will get a character from the buffer. The boolean result will indicate whether the operation was successful. It will fail if the buffer is empty or if the buffer was never opened successfully for reading.

FUNCTION PCPutLine (L: SUStr): BOOLEAN;
PCPutLine will put a line into the buffer. A CR is put in the buffer following the string passed to PCPutLine. The boolean result will indicate whether the operation was successful. It will fail if the buffer is full or if the buffer was never opened successfully for writing.

FUNCTION PCGetLine (VAR L: SUStr): BOOLEAN;
PCGetLine will get a line from the buffer, where a line is the text from the current buffer pointer up to the next CR or the end of file (whichever comes first). The boolean result will indicate whether the operation was successful. It will fail if the buffer is empty or if the buffer was never opened successfully for reading.

You will notice the following function in the ProgComm interface; it is used for special-purpose communication between the Workshop shell and various Workshop tools.

FUNCTION PCShellCmd (Cmd: INTEGER; P: SUStrP): BOOLEAN;
For internal use by Workshop development system tools only. Contact me if you have a need to know about this function.

Release 3.0 Notes
CHAPTER 2, THE FILE MANAGER

Overview of Changes to the File Manager

The significant changes to the File Manager involve:

- The Operating System's new hierarchical catalog structure.
- Transfer operations onto more than one micro diskette.
- Password protection.
- The new OS device names.

The Operating System uses new physical device names, but still supports the old names as device aliases. You can specify a device using either the name or the alias; the OS refers to devices by name. The new names are:

Name	Alias	Device
#10#1	RS232A	Serial Port A
#10#2	RS232B	Serial Port B
#11	PARAPORT	Parallel Connector (Lisa 1)
#12	UPPER or PARAPORT	Built-in hard disk (Lisa 2)
#13	LOWER	Micro diskette drive
#15#1	ALTCONSOLE	Alternate console
#15#2	MAINCONSOLE	Main console
#x	SLOTx	Peripheral at expansion slot x
#x#y	SLOTxCHANy	Peripheral at expansion slot x, connector y
#x#y#z	SLOTxCHANyDEVz	Peripheral at expansion slot x, connector y, device z

AddCatalog Command

Files on a volume can now be arranged under catalogs and subcatalogs. The AddCatalog command lets you create new catalogs. The pathname you specify for a catalog should refer to a volume that has been initialized using the Release 3.0 software.

The *hyphen* is the catalog delimiter, so a file name referring to a file in a catalog might look like "-vol-cat-file" or "-vol-cat1-cat2-file", and so on. A file name of the form "cat-file" is interpreted relative to the current prefix and thus might refer to "-vol-cat-file" or "-vol-cat1-cat-file", depending on whether the prefix is set to a volume or to a catalog. A catalog specified by a pathname without a volume part will be created using the current main prefix.

There is no special command to put a file in a catalog. Once a catalog has been created, new files get put into it in two ways:

1. If the new file's name is specified by a full pathname with volume and catalog parts, the file is put in the specified catalog. (A catalog must exist before a file can be put into it.)

2. If the new file's name is a partial pathname without a volume part, and the current prefix is a catalog, the file is put in the prefix catalog (or a subcatalog, if the file's pathname includes a catalog part).

When the OS tries to find a file given a partial pathname, the file will be found only if (1) the pathname has no catalog part and is located in the prefix volume or catalog, or (2) the pathname has a catalog part corresponding to a path starting with a catalog at the top level of the prefix volume or catalog.

Backup/Copy/Transfer to Multiple Micro Diskettes (See Sections 2.3.1, 2.3.2, and 2.3.7)

The Backup, Copy and Transfer commands now allow backups, copies, and transfers to multiple volumes. If a list of files is being copied (or backed up, or transferred) to a micro diskette and you run out of space, you will be told which file didn't fit and how many more blocks were needed, and you will be asked whether you want to continue on another diskette. If you answer Yes, you will be led through a diskette change and the operation will continue. Note that the volume names of the subsequent diskettes need not match the first, even if the original destination was specified with a particular volume name (instead of a device name).

List and Names Commands (See Sections 2.3.4 and 2.3.13)

There are two new attributes for items in the List display. The **D** attribute indicates a directory (a catalog object) and the * attribute indicates a password-protected file (see Password Protection, below).

The List and Names commands now indent names to show the catalog structure whenever you list a contiguous set of files. If you specify a wildcard character followed by a string to match, the files shown will not necessarily be contiguous, and will not be indented.

When a file name has to be truncated to fit into a limited field of the display (as in the List command), the missing characters are now indicated by an elipsis (...).

Prefix Command (See Section 2.3.5)

Prefixes may now be set to catalogs in addition to volumes. A prefix to a catalog or subcatalog must be specified with a complete pathname.

The effect of the current prefix on the interpretation of file names is discussed under AddCatalog Command, above.

WARNING

Setting the main prefix (or working directory) may cause problems when running programs that use intrinsic units (this includes all the Workshop tools). The OS loader tries to find a program's intrinsic libraries using the pathnames it finds in INTRINSIC.LIB; if these names are partial pathnames, it looks on the prefix volume or catalog, *not the boot volume*. To assure that your program's intrinsic libraries are found, you can do one of two things:

1. Copy the intrinsic libraries to the prefix catalog. This way, you can support several different library environments on the same volume, though you could end up with a proliferation of library files.
2. Change the names of the libraries in INTRINSIC.LIB to pathnames of the form "-#BOOT-libname" (using the IUManager, described in Chapter 11, Utilities), then reboot so the OS will store the new names. This method is better, but be careful changing things in INTRINSIC.LIB.

If you unmount the main prefix volume by ejecting the diskette, Scavenging the volume, or using the Unmount command, the boot volume becomes the prefix volume.

Rename Command (See Section 2.3.6)

To rename a file to a name that only differs from the original in the *case* of the letters (e.g., DEMOGRAPHICS.OBJ to DemoGraphics.Obj), you must first Rename the file to a temporary name, then Rename that to the name you want.

Password Protection (See Section 2.3.10, FileAttributes)

Two new commands for password protection are found under the FileAttributes command. AddPassword allows you to password-protect a file (or files, using wildcards). RemovePassword allows you to remove a file's password, but you must know the password to remove it.

The Workshop tools can't open a file once it is password-protected; you must remove the password before you can use the file.

Initialize Command (See Section 2.3.11 and 2.4.1)

Volumes initialized under the new Workshop and OS have a hierarchical catalog structure. Since this structure cannot be applied retroactively, an existing volume must be reinitialized in order to take advantage of these features. Commands that operate on a list of files (e.g, List) run much faster on a reinitialized disk, because in the new structure names are already sorted.

Online Command (See Section 2.3.14)

The Online command now displays both the new OS device names and the old names, which are now device aliases. The new device names are listed in the Overview at the beginning of this section, and shown in the syntax diagrams under File Specifiers, below.

The prefix attribute **P** is now sometimes displayed as a lowercase **p**. Uppercase **P** indicates that the main prefix is the indicated volume, while lowercase **p** indicates that the prefix is a catalog on that volume.

NOTE

The Online command uses the configuration information set by Preferences. If Online output says that it could not find #11 (PARAPORT) on a Lisa 2/10, use Preferences to detach the non-existent device. If the Workshop pauses unexpectedly in the middle of Online output, it means a device is configured but not present. Make sure that Preferences' idea of how the system is configured is correct.

File Specifiers (See Section 2.4.2)

File specifiers have changed to allow for subcatalogs, new device names, and the new wild card characters. The diagrams that follow show the new format of file specifiers, replacing those on pages 2-9 and 2-10 of the manual. (The logical device names have not changed, but the diagram is repeated here for convenience.)

<ART> syntax diagrams: file-specifier,
file-name-or-pattern,
volume/catalog-spec,
physical-device,
physical-device-name & -alias,
logical-device,
wild-card-spec.

New ≠ and ¿ Wildcard Characters (See Section 2.5)

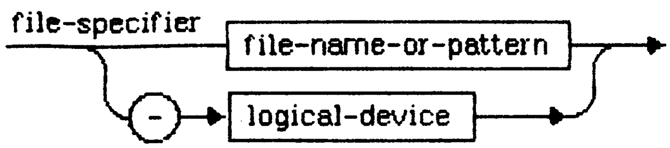
Because of the new hierarchical catalog structure, the meanings of the = and ? wildcard characters have changed, and the new analogous wildcards ≠ (OPTION =) and ¿ (OPTION ?) have been added. The plain = and ? wildcards mean search for a match only across the top level of the catalog, while the option wildcards mean search through all levels. The way in which the matches are made is the same:

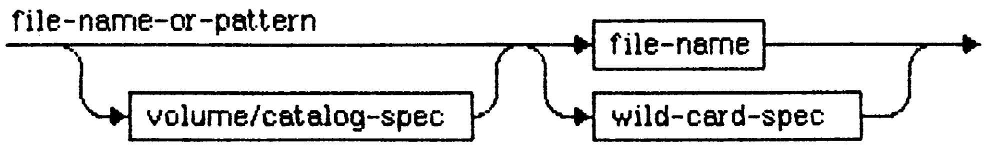
= matches any string in the top level of the catalog.

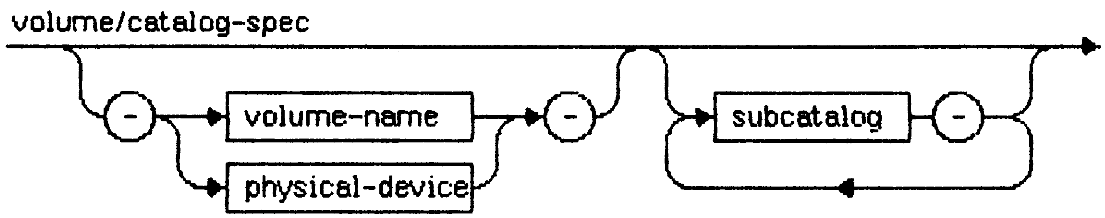
≠ matches any string throughout all levels of the catalog.

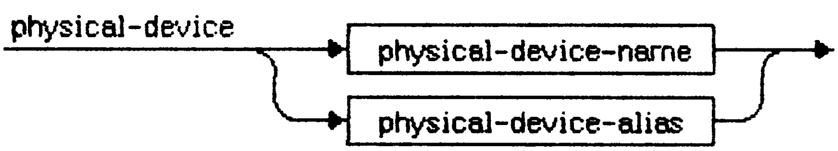
? matches any string in the top level of the catalog, asking for confirmation of each file name before performing the operation.

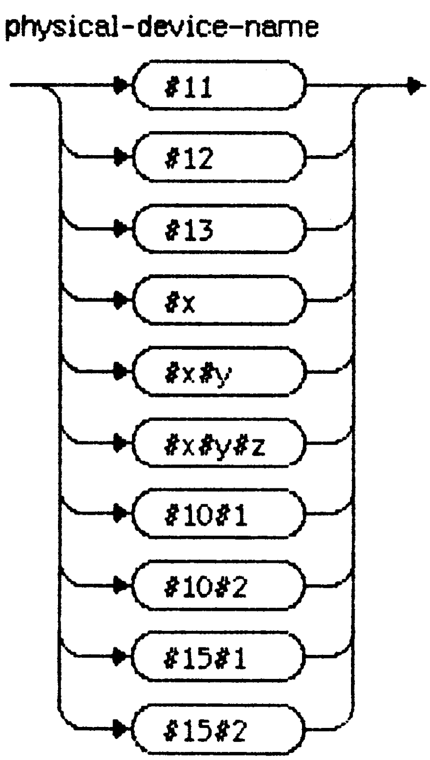
¿ matches any string throughout all levels of the catalog, asking for confirmation of each file name before performing the operation.

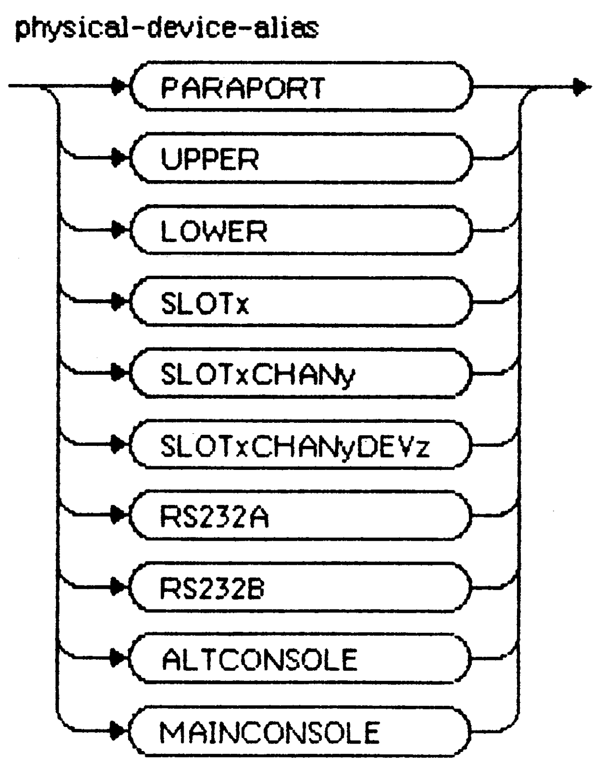




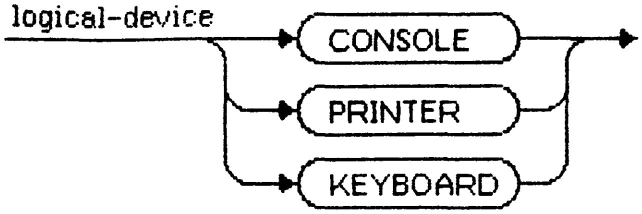


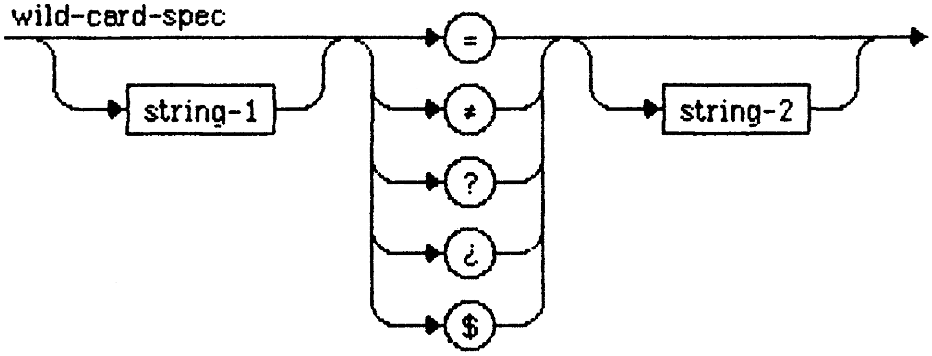






(The device names on the left correspond to the device aliases on the right.)


logical-device
CONSOLE
PRINTER
KEYBOARD


wild-card-spec
string-1
=
≠
?
¿
$
string-2

DUMPICODE

I-Code Definition

The first pass of the Pascal compiler generates a .I file. Its contents are described in this document. Please note that this information is likely to change without notice; there is no guarantee that it is correct.

Abbreviations:

```
     expr => expression
     addr => address
      (W) => size of operand is a word
      (B) => size of operand is a byte
```

of other I-codes

(W) = 16 bits } stuff offset

Global Def's

```
00
```

unused

Variable references:

```
    01 +offset(W)              Global variable reference
    02 +offset(W)              Local variable reference
    03 lev(B) +offset(W)       Intermediate level variable reference
    04 com(B) +offset(W)       Common variable reference
    05 reg(B) O(B)             Register reference
 or 05 reg(B) loadSize(B) expr
 or 05 reg(B) loadCount(B) loadSize(B) expr
          reg=register number (0..15)
          loadCount=number to bump count by (only significant with temp
           registers)
             0=none (last use of reserved register)
             1=sustaining use or first&last use
             2=first use and reservation for future use
          loadSize=size of expression to load register with
             0=no load
             1=byte
             2=word
             3=long
    06 ???????                 String temp
    07 ???????                 Set temp
 08-0B Multiple Bytesize       1/2/4/8 byte temp {09=>2 byte operand}
```

06/07: ~~???????~~ unused — ~~String temp~~, ~~Set temp~~

Addressing operators:

```
    0C addr                    '^'  - Dereference operator
    0D addr                    '^'  - File dereference operator
    0E addr                    '^'  - Text file dereference operator
    0F +offset(W) addr         '.'  - Record field offset
 10-13 Wordsize addr expr      '[]' - 1/2/4/8 byte array index
```

```
14 Wordsize Wordsize addr expr     '[]' - Long array index
15 Bytesize Wordsize addr expr     '[]' - Packed array access
16 addr                            '@'  - Address of operator
```

Constants:

```
   17                           nil
18-1B Multiple Bytesize         1/2/4/8 byte constant {19=>2 byte operand}
   1C stringSize(B) 'ABC...'    String constant
   1D stringSize(B) 'ABC...'    PAOC Constant
   1E setSize(B)
        for 1 to bytesize do
           getnextoperand(B)    Set constant
   1F                           [] - Null set
```

Assignment operators:

```
20-22 flippable(B) addr expr       ':=' - 1/2/4/8 byte assignment
           0 or 1
```

Flippable is ~~true~~ 1 if the assignment left hand side can be computed after the right hand side. If true, we have expr addr.

```
20-22  2(B) addr expr  Binary in-line assignment of byte/word/long
                       expression. Evaluate addr, then expr, then assign
                       value of expr to location addr.  Return expr.

20-22  3(B) expr1 addr expr2  Triple in-line assignment of byte/word/long
                              expression.  Evaluate expr1, then addr, then
                              expr2, then assign value of expr2 to location
                              addr.  Return expr1.
```

23 - reserved - 8 -

```
   24 Bytesize Wordsize addr expr    ':=' - Multiple byte assignment
   25 Bytesize addr expr             ':=' - Set assignment
   26 1stBytesize 2ndBytesize Wordsize
        if 1stBytesize =21 then        {PCKDARR}
          addr expr expr
                      else
        Bytesize expr expr             ':=' - Packed assignment
   27 Bytesize addr expr             ':=' - String assignment
   28 Bytesize Bytesize addr expr    ':=' - PAOC Assignment
   29 Bytesize addr expr             ':=+' - Add to
   2A Bytesize addr expr             ':=-' - Subtract from
   2B Bytesize                       WITH field reference, level nnn
   2C lev(B) isptr(B) addr           Begin WITH statement, level nnn
   2D lev(B)                         End WITH statement, level nnn
   2E lo-----(W) hi-----(W) expr     2 Byte Range Check
   2F hi-(B) expr                    String Range Check-assignment, not index
```

Data Conversion:

30-32 expr	1->2, 2->4, 1->4 integer
33-35 expr	2->1, 4->2, 4->1 integer
36-37 expr	4->8, 8->4 real conversion
38-39 expr	4->4, 4->8 Float
3A-3B expr	4->4, 8->4 Trunc
3C-3D expr	4->4, 8->4 Round
3E Bytesize expr	Extract unsigned field
3F Bytesize expr	Extract signed field

Scalar operators:

40-41 expr expr	2/4 Scalar Addition
42-43 expr expr	2/4 Scalar Subtraction
44-45 expr expr	2/4 Scalar Multplication
46-47 expr expr	2/4 Scalar Division
48-49 expr expr	2/4 Scalar Modulus
4A-4B expr	2/4 Scalar Negate
4C-4D expr	2/4 Scalar Absolute Value
4E-4F expr	2/4 Scalar Square
50-52 expr expr	1/2/4 Scalar AND
53-55 expr expr	1/2/4 Scalar OR
56-58 expr expr	1/2/4 Scalar XOR
59-5B expr	1/2/4 Scalar NOT
5C-5E expr expr	1/2/4 Scalar <
5F-61 expr expr	1/2/4 Scalar >
62-64 expr expr	1/2/4 Scalar <=
65-67 expr expr	1/2/4 Scalar >=
68-6A expr expr	1/2/4 Scalar =
6B-6D expr expr	1/2/4 Scalar <>
6E expr	Boolean NOT
6F expr	ODD
70-71 expr expr	4/8 Real Addition
72-73 expr expr	4/8 Real Subtraction
74-75 expr expr	4/8 Real Multiplication
76-77 expr expr	4/8 Real Division
78-79 expr expr	4/8 Real Modulus
7A-7B expr expr	4/8 Real <
7C-7D expr expr	4/8 Real >
7E-7F expr expr	4/8 Real <=
80-81 expr expr	4/8 Real >=
82-83 expr expr	4/8 Real =
84-85 expr expr	4/8 Real <>
86-87 expr	4/8 Real Negation
88-89 expr	4/8 Real Absolute Value

```
8A-8B expr                  4/8 Real Square
   8C expr                  TRAPV
   8D
   8E   } unused
   8F
```

String Operators:

```
   90 expr expr                                String <
   91 expr expr                                String >
   92 expr expr                                String <=
   93 expr expr                                String >=
   94 expr expr                                String =
   95 expr expr                                String <>
   96 stringsize(B) stringsize(B) expr expr    PAOC <
   97 stringsize(B) stringsize(B) expr expr    PAOC >
   98 stringsize(B) stringsize(B) expr expr    PAOC <=
   99 stringsize(B) stringsize(B) expr expr    PAOC >=
   9A stringsize(B) stringsize(B) expr expr    PAOC =
   9B stringsize(B) stringsize(B) expr expr    PAOC <>
   9C
   9D   } unused
   9E
   9F
```

Set Operators:

```
   A0 setsize(B) expr expr      Set +
   A1 setsize(B) expr expr      Set -
   A2 setsize(B) expr expr      Set *
   A3 setsize(B) expr expr      IN
   A4 setsize(B) expr expr      Set <=
   A5 setsize(B) expr expr      Set >=
   A6 setsize(B) expr expr      Set =
   A7 setsize(B) expr expr      Set <>
   A8 setsize(B) expr           Singleton Set
   A9 setsize(B) expr expr      Set Range
   AA setsize(B) Bytesize expr  Adjust Set
   AB
   AC   } unused
   AD
   AE
   AF
```

Procedure/Function Calls:

```
    B0 index(W)                   User Function Call
    B1 index(W)                   User Procedure Call
    B2 key(B)                     Standard Function Call
    B3 key(B)                     Standard Procedure Call
    B4 addr                       Parametric Function Call
    B5 addr                       Parametric Procedure Call
    B6 room(B)                    Make Room for Function Result
    B7 addr                       Reference Parameter
 B8-BB expr                       1/2/4/8 Byte Value Parameter
    BC size(W) expr               Large Value Parameter
    BD setsize(B) expr            Set Value Parameter
    BE                            Begin Parameter List
    BF index(W)                   User Function/Procedure Parameter
```

Control:

```
    C0 label(W)                             Define Internal Label
    C1 label(W)                             Jump
    C2 label(W) expr                        Jump False
    C3 label(W) expr                        Jump True
    C4 usernum(W) label(W)                  Define Local User Label
    C5 usernum(W) label(W) linknum(W)       Define Global User Label
    C6 usernum(W) label(W)                  Jump to Local User Label
    C7 lev(B)  linknum(W)                   Jump to Global User Label
    C8 expr                                 Case Jump
    C9 0(B) lobound(W) hibound(W)
       elseLabel(W) loLabel(W) hiLabel(W)   Case Table--must follow case jump
 or C9 1(B) lobound(W) hibound(W)
       elseLabel(W) count(W)
       [value(W), label(W)]                 If expr list--must follow case jump
    CA ctrsize(B) addr expr1 expr2 expr3    FOR statement
           ctrsize - size of loop counter (1,2,4)
           addr    - counter
           expr1   - start
           expr2   - end
           expr3   - increment
    CB                                      FOR end
    CC                                      CASE end
    CD linenum(W)                           Line number
        or if linenum = -1 then
    CD -1(W) length(B) filename             To open an INCLUDE or USES file
    CD -2(W) bool(B)                        Assembly listing control switch
    CE regset(W)                            Temp registers mask
          regset=set of register (0..15)
              bit on=reserve register
              bit off=make register available for codegen temp use
```

```
    CF

D0--DF

E0--EF

    F0 (ln) (un) (cfn) (sn) fnsw
        lev(B) varsize prmbyts
        glb regmask               Begin Module
                                  (ln)     - 8-byte Linker name
                                  (un)     - 8-byte User name
                                  (cfn)    - 8-byte class father name
                                  (sn)     - 8-byte Segment name
                                  fnsw     - function switch (fn or proc)
                                  lev      - level (1=global)
                                  varsize - Number of bytes of local variables
                                  prmbyts - Bytes of parameters + 8
                                  glb      - Global Label Flag is Bit 0
                                             Stack Expan. Flag is Bit 1
                                  regmask - register mask for MOVEM
    F1 (ln) (un) num(W) lev1(B)   External Reference Definition
    F2 (ln) num(B) kind(B)        Common Reference Definition
    F3 (cn) nnnnnnn               Common Area Definition
    F4 (un) Bytesize textaddr4(W) Bytesize Bytesize textsize4(W) Bytesize
        globsize2(W) untType(B)    Unit File Header
  F5-FB
    FC fn(B) size(W) const(W)     In-line function switch
    FD fn(B) level(B) method(B)   Method call
    FE debugflag(B)               End of module (D compiler option)
    FF                            End of file
```

Apple Computer Inc.
APPLE-32 DEVELOPMENT TOOLS - A1

Feb. 22, 1984

New I-codes for Optimization

The I-code changes and new I-codes for the current optimization project include:

Code Name	Operands	Definition
$05 Register	Reg LoadCount LoadSize Expr... (B) (B) (new!) (B)	Register reference.

Reg=register number (0..15)
Loadcount=number to bump 'count' by (only significant with temp registers):
- 0 = none (last use of reserved register)
- 1 = sustaining use or first&last use
- 2 = first use and reservation for future use

LoadSize=size of expression to load register with.
- 0 = no load.
- 1 = byte.
- 2 = word.
- 3 = long.

Code Name	Operands	Definition
$CE TempStmt	Regset (W)	Temp Registers Mask.

Regset=set of register (0..15)
bit on = reserve register
bit off= make reg available for codegen temp use.

Binary Inline Assign's:

$20	2	Addr...	Expr...	Inline assignment of byte expr.
$21	2	Addr...	Expr...	Inline assignment of word expr.
$22	2 (B)	Addr...	Expr...	Inline assignment of long expr.

Evaluate "addr", then "expr", then assign value of "expr" to location "addr". Return "expr".

Triple Inline Assign's:

$20	3	Expr1...	Addr...	Expr2...	Inline assignment of byte expr.
$21	3	Expr1...	Addr...	Expr2...	Inline assignment of word expr.
$22	3 (B)	Expr1...	Addr...	Expr2...	Inline assignment of long expr.

Evaluate "expr1", then "addr", then "expr2", then assign value of "expr2" to location "addr". Return "expr1".