Lisa

Development System

INTERNALS

DOCUMENTATION

apple computer inc

Development Tools Group
FIRST DRAFT--February 1984

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) 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 interfaced 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 conmpiler 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

17/p.2

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 *SOROC* 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 seperate 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.

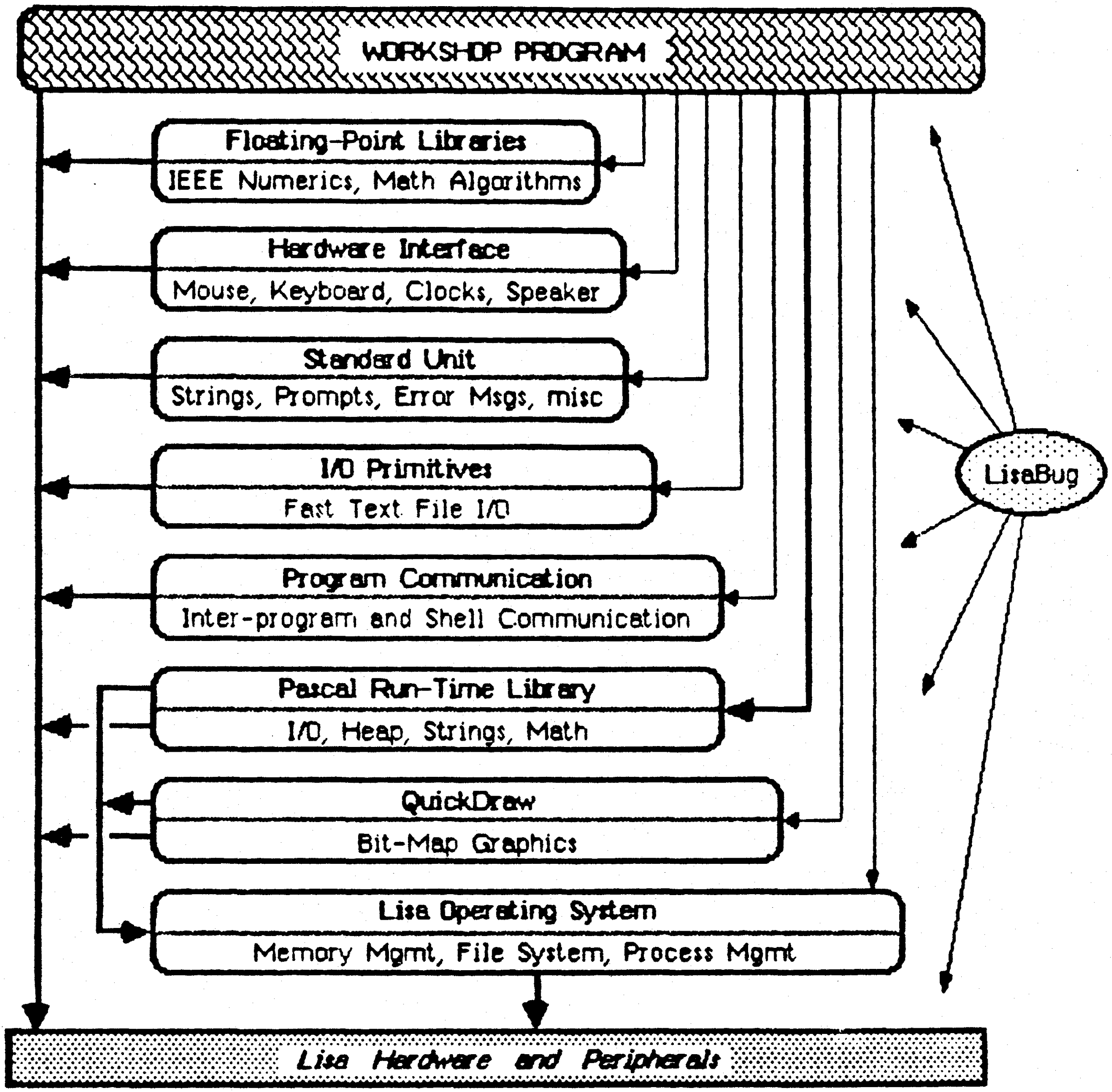
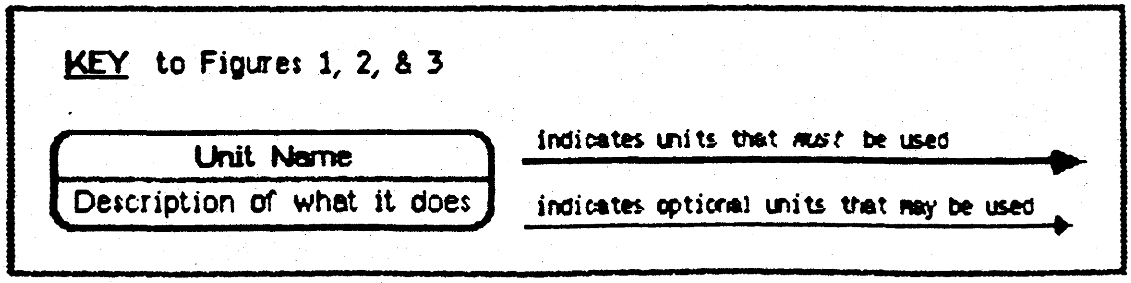


Figure 1
The Workshop Run-Time Environment


KEY to Figures 1, 2, & 3
Unit Name
Description of what it does
indicates units that must be used
indicates optional units that may be used

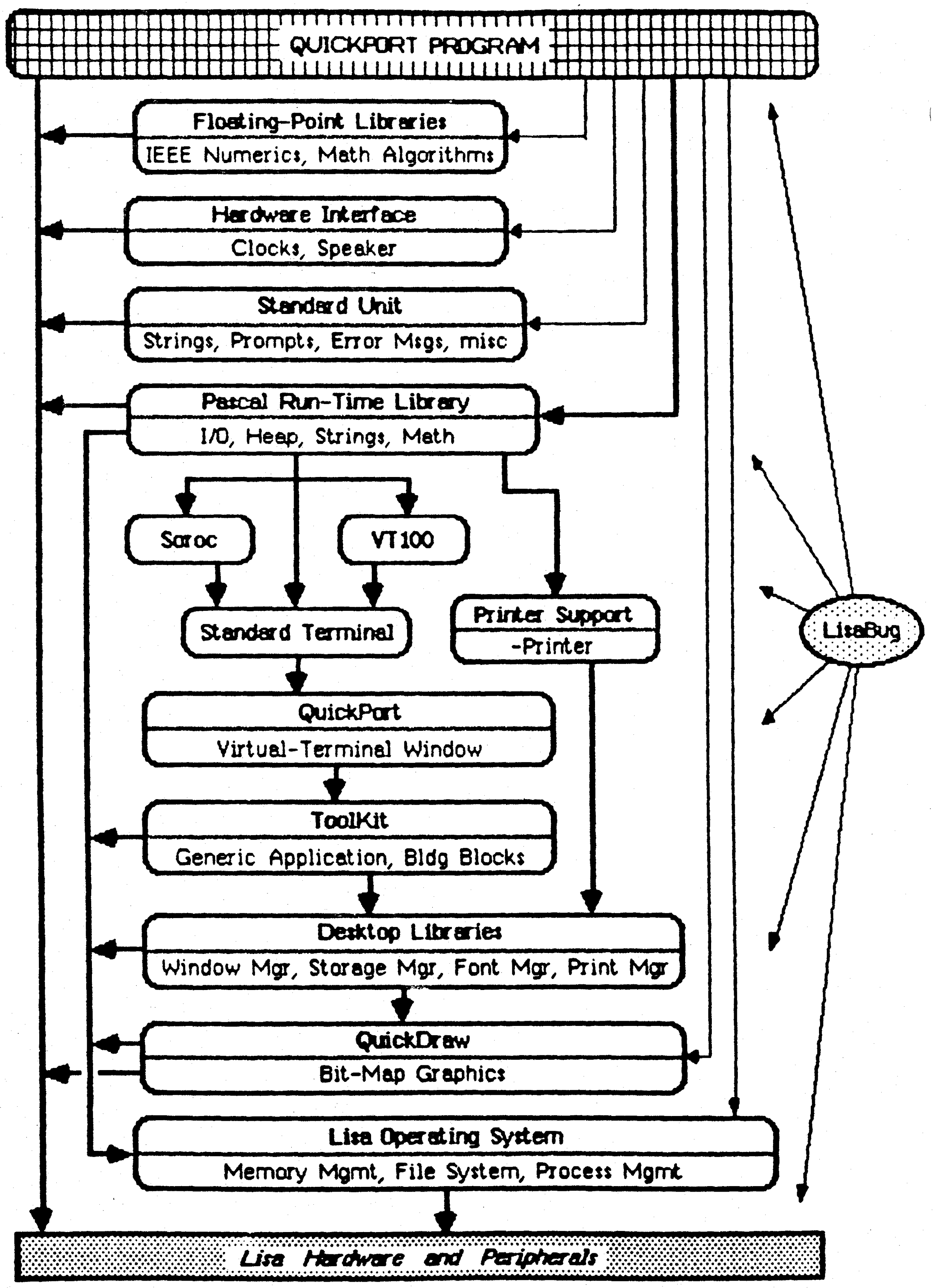


Figure 2
The QuickPort Run-Time Environment

17/p.14

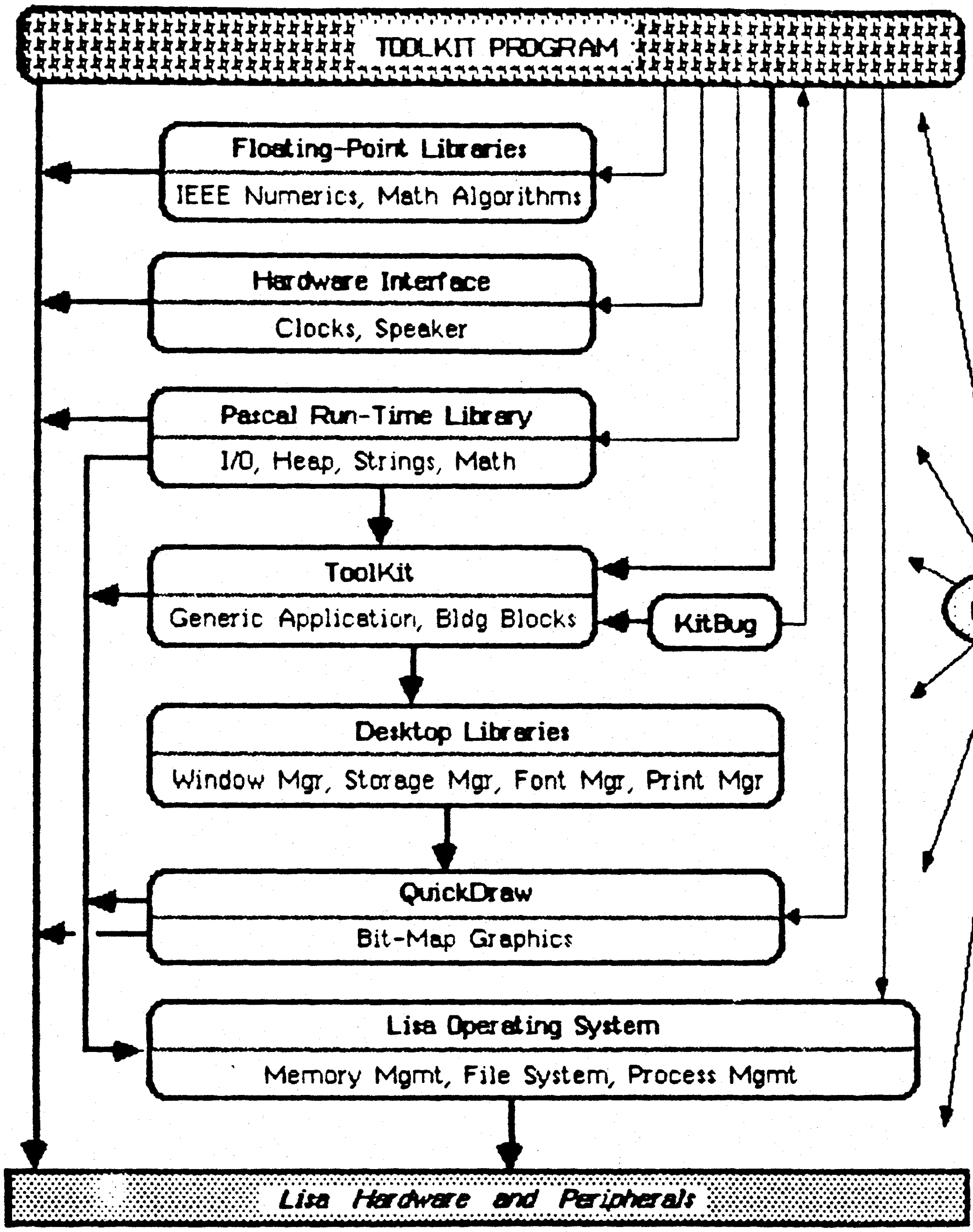


Figure 3
The ToolKit Run-Time Environment

17/p.25

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
O{quit Sys-mgr}
G{generate}-paraport-temp
$IF %1 <> '' THEN
  %1
$ELSE
  %0
$ENDIF
S{Sys-mgr}O{OutputRedirect}-console
O{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;                              PROCEDURE UPPER;
VAR A,B:INTEGER;                              VAR A,B,TEMP:INTEGER;
    PROCEDURE LOWER;                              PROCEDURE LOWER(A,B:INTEGER);
    BEGIN                                         BEGIN
    A := B;              converts to-->           A := B;
                                                  TEMP := A;
    END;                                          END;
BEGIN                                         BEGIN
LOWER;                                        LOWER;
{other statements}                            A := TEMP;
{frequent uses of A and B}                    {frequent uses of A and B}
END;                                          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);                FUNCTION PROC(N:INTEGER):INTEGER;
PROCEDURE LOCAL;                              FUNCTION LOCAL(A,B:INTEGER):INTEGER
...                          becomes-->       ...
```

```
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 <u>pages</u>.
- One 1024-byte <u>header page</u> 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

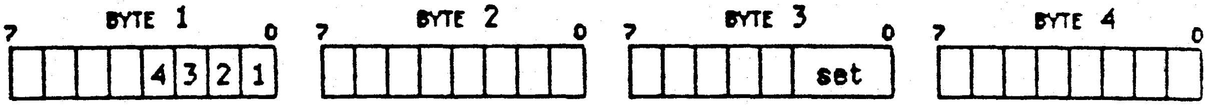
Packed Records

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.

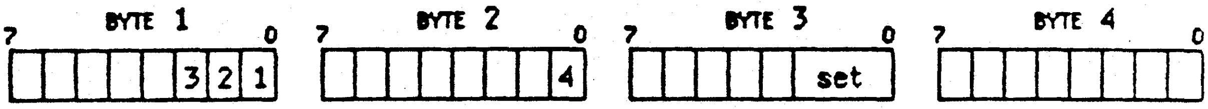
   Note that only scalar values and subranges are considered packable; everything else must go on a word boundry.

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



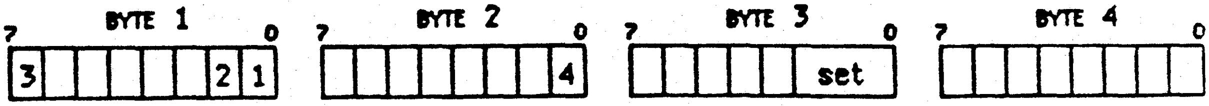
2. Any empty bytes are filled by moving the previous field into the empty byte if:

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



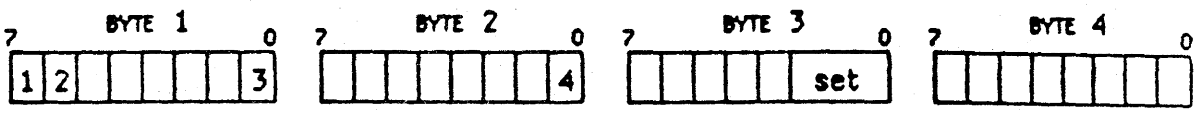
3. Any field that fits in a byte or word and does not share that space with other fields is now designated "unpacked".

   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.

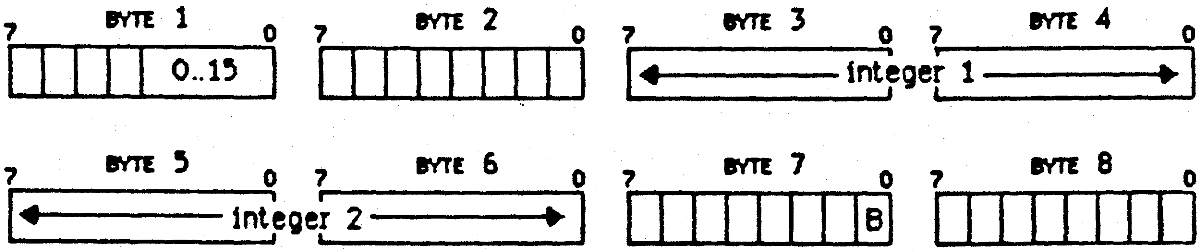
   Only boolean or subrange types can be packed; all other types start on word boundries, so steps 2 and 3 only apply to these types.
2. If a byte would be left empty (so the next field can start on a word boundry), and there is more than one field in the previous byte, the last (low-ordered) field is moved 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

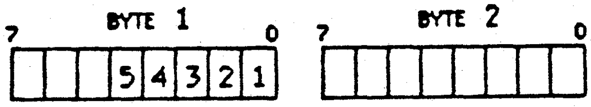
Packed arrays are also code-expensive, except for packed arrays of char. (These are treated as a special case, and the code associated with them is compact.)

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:

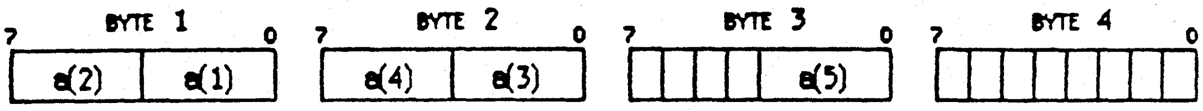
A	B
0..1	1
0..2	2
0..3	2
0..4	4
0..10	4
0..20	8
0..255	8
0..395	16

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 modul
                           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.

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.

Rich Page
Apple Computer, Inc.
May 4, 1983

Execution Environment of the Lisa Pascal Compiler

Registers:

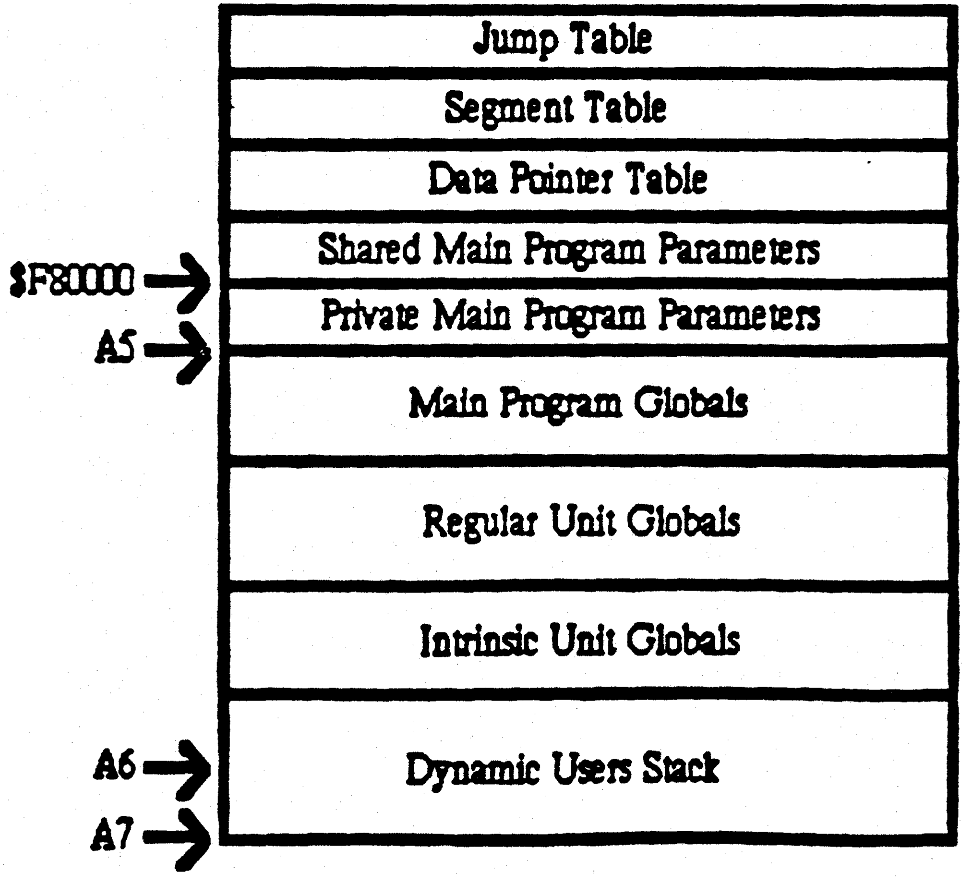
D0-D2/A0-A1	User temporaries
D0-D3/A0-A2	Compiler temporaries
D4-D7/A3-A4	Compiler uses for locals & pointers
A5	Pointer to global frame
A6	Pointer to local frame
A7	Pointer to top of stack

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).

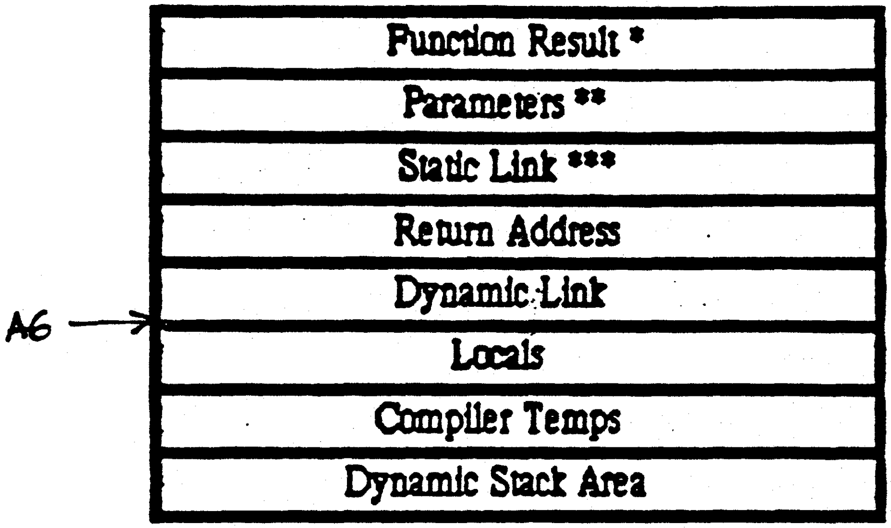
Initially the Loader allocates enough space to cover these areas and the user min stack requirements. The system also enforces a upper limit (ie. max stack).

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:



* 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
        MOVEL       A7,A0
        SUBL        #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:

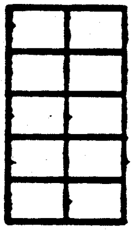
1) Automatic stack expansion.
2) Range checking for values, indexes and strings.
3) Debuggung info (ie. the procedure name).

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.

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.

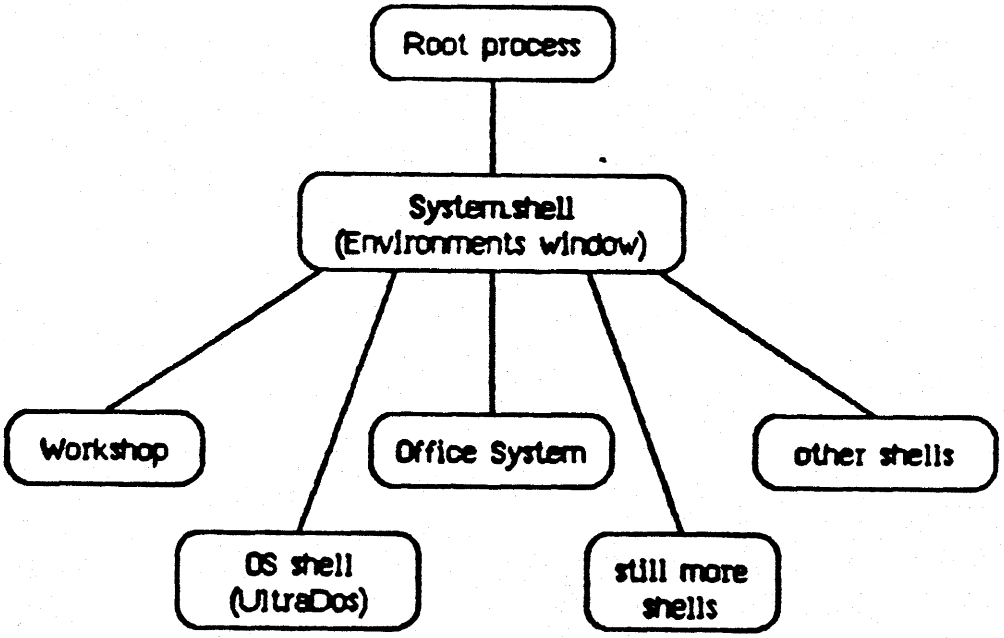


Figure 1
Process Picture

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** BlockIDInit; 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**).

SON: 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

SON: 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_SON_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 varient record which allows us to address the packed arrey 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)
```

Toolkit 3.0