HP-UX Assembler Reference and Supporting Documents

HP 9000 Series 300 Computers

HP Part Number 98597-90020

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

March 1986...Edition 1

May 1986...Update

July 1986...Edition 2. Update incorporated.

July 1987...Edition 3

April 1988...Edition 4. Minor bug fixes. New information on how to determine which processor is used at run time. Tutorial added for atime assembly timing facility.

December 1988...Edition 5. Information added to Chapter 7 on the new version pseudo-op.

Table of Contents

Introduction 1

The two HP-UX assemblers for Series 300 computers, *as10* and *as20*, enable you to write assembly language programs for Model 310, 320, 330, 350, 360, and 370 computers. The Model 310 computer uses the MC68010 processor; the Models 320, 330, 350, 360, and 370 use the MC68020/30 processor and the MC68881 floating point coprocessor. The HP 98248 Floating-Point Accelerator is also supported by the *as20* assembler. Both assemblers can be accessed through the general-purpose HP-UX assembler command *as*(1). This reference manual describes how to use both assemblers so that the full capabilities of supported hardware can be effectively used.

Manual Contents

Chapter 1: Introduction identifies related processor manuals, lists various precautions related to using the assemblers, introduces the two assemblers, *as10* and *as20*, and the *as* command driver that can be used to access either assembler as appropriate, and provides a brief description of how to invoke the assembler and use its different command options, how to use the C compiler, *cc*(1), to assemble programs, and finally, provides a brief overview of how the assembler operates.

Chapter 2: Assembly Language Building Blocks discusses the basic building blocks of *as* assembly language programs: **identifiers**, **register identifiers**, and **constants**.

Chapter 3: Assembly Language Syntax describes the syntax of *as* assembly language programs and introduces **labels**, **statements**, and **comments**.

Chapter 4: Segments, Location Counters, and Labels provides a detailed discussion of the **text**, **data**, and **bss** segments, and their relation to location counters and labels.

Chapter 5: Expressions defines the rules for creating expressions in *as* assembly language programs.

Chapter 6: Span-Dependent Optimization describes optional optimization of branch instructions.

Chapter 7: Pseudo-Ops describes the various pseudo-ops supported by the *as* assembler. Pseudo-ops can be used to select a new segment for assembly output, initialize data, define symbols, align the assembly output to specific memory boundaries, set the rounding mode mode for floating point input, and set the floating point co-processor id.

Chapter 8: Address Mode Syntax defines the syntax to use for the addressing modes supported by *as*. Helpful notes on using various addressing modes are given. It also discusses how the *as20* assembler optimizes address formats and displacement size.

Chapter 9: Instruction Sets describes instructions sets for the MC68000, MC68010, and MC68020/30 microprocessors, the MC68881 Floating-Point Co-processor, and the HP 98248 Floating-Point Accelerator.

Chapter 10: Assembler Listing Options describes use of the assembler listing options (*-a* and *-A*).

Appendix A: Compatibility Issues discusses issues that you should consider if you wish to write code that is compatible between MC68010- and MC68020/30-based computers.

Appendix B: Diagnostics provides information on diagnostic error messages output by the assembler.

Appendix C: Interfacing Assembly Routines to Other Languages describes how to write assembly language routines to interface to C, FORTRAN, and Pascal languages.

Appendix D: Examples contains examples of *as* assembly language source code.

Appendix E: Translators describes translators which can be used to convert PLS (Pascal Language System) and old Series 200/300 HP-UX assembly code to *as*-compatible format.

Appendix F: Unsupported Instructions for Series 300s provides information on MC680XX instructions that are not supported by the Series 300 machines.

Related Documentation

This manual deals mainly with the use of the *as*(1) assembler. This manual does *not* contain detailed information about the actual instructions, status register bits, handling of interrupts, processor architecture, and many other issues related to the M68000 family of processors. For such information, you should refer to the appropriate processor documentation for your computer.

MC68010 Documents

The HP 9000 Model 310 computer uses an MC68010 microprocessor. Therefore, you will need the *MC68000 16/32-Bit Microprocessor Programmer's Reference Manual*, which contains detailed information on the MC68010 instruction set, status register bits, interrupt handling, and other issues related to using the MC68010 microprocessor.

MC68020/30 Documents

The HP 9000 Models 320, 330, 350, 360, and 370 computer uses an MC68020/30 microprocessor and MC68881 Floating-Point Coprocessor. The Models 330, 350, 360, and 370 will also support an optional HP 98248 Floating-Point Accelerator. You will need the following:

- *MC68020 32-Bit Microprocessor User's Manual*, which describes the MC68020 instruction set, status register bits, interrupt handling, cache memory, and other issues
- *MC68030 32-Bit Microprocessor User's Manual*, which describes the MC68030 instruction set, status register bits, interrupt handling, cache memory, and other issues
- *MC68881 Floating-Point Coprocessor User's Manual*, which describes the floating-point coprocessor, its instruction set, and other related issues.
- *HP 98248 Floating-Point Accelerator Manual*, which describes the floating-point accelerator, its instruction set and other related issues.

IMPORTANT

The reference manuals described above are not provided with the standard HP-UX Documentation Set. If you intend to use the HP-UX Assembler on your system, you can order these manuals through your local Hewlett-Packard Sales Representative.

The HP-UX Reference

The following entries from the *HP-UX Reference* may also be of interest:

- *as*(1) — describes the assembler and its options.
- *ld*(1) — describes the link editor, which converts *as* relocatable object files to executable object files.
- *a.out*(4) and *magic*(4) —describe the format of object files.

Assembler Versions

The Series 300 HP-UX operating system supports two different versions of the *as* assembler, *as10* and *as20*. The *as10* assembler (*/bin/as10*) is compatible with the MC68010 instruction set. The *as20* assembler (*/bin/as20*) supports the MC68020/30, MC68881, and the HP 98248 Floating-Point Accelerator instruction set.

The separate driver program */bin/as*, when executed, makes a run-time check to determine the type of microprocessor that is present on the host computer, then starts the appropriate assembler: */bin/as10* on MC68010-based computers (Model 310), and */bin/as20* on MC68020/30-based machines (Models 320, 330, 350, 360, and 370). Options to the *as*(1) command are available to override the run-time default assembler selection. Use of these options (+x and +X) are explained later in this chapter under the heading "Invoking the Assembler". You also have the option of invoking *as10* or *as20* directly as a standard HP-UX command, thus bypassing the *as* driver program.

Precautions

Though for the most part *as* notation corresponds directly to notation used in the previously described processor manuals, several exceptions exist that could lead the unsuspecting user to write incorrect *as* code. These exceptions are described next. (Note that further differences are described in the chapters "Address Mode Syntax" and "Instruction Sets".)

Comparison Instructions

One difference that may initially cause problems for some programmers is the order of operands in *compare* instructions: the convention used in the *M68000 Programmer's Reference Manual* is the opposite of that used by *as*. For example, using the *M68000 Programmer's Reference Manual*, one might write:

```
CMP.W   D5,D3      Is register D3 <= register D5?
BLE     IS_LESS    Branch if less or equal.
```

Using the *as* convention, one would write:

```
cmp.w   %d3,%d5    # Is register d3 <= register d5?
ble     is_less    # Branch if less or equal.
```

This follows the convention used by other assemblers supported in the UNIX[1] operating system. This convention makes for straightforward reading of *compare*-and-*branch* instruction sequences, but does, nonetheless, lead to the peculiarity that if a *compare* instruction is replaced by a *subtract* instruction, the effect on condition codes will be entirely different.

This may be confusing to programmers who are used to thinking of a comparison as a subtraction whose result is not stored. Users of *as* who become accustomed to the convention will find that both the *compare* and *subtract* notations make sense in their respective contexts.

[1] UNIX(TM) is a trademark of AT&T Bell Laboratories, Inc.

Simplified Instructions

Another issue that may cause confusion for some programmers is that the M68000 processor family has several different instructions to do basically the same operation. For example, the *M68000 Programmer's Reference Manual* lists the instructions `SUB`, `SUBA`, `SUBI`, and `SUBQ`, which all have the effect of subtracting a source operand from a destination operand.

The *as* assembler conveniently allows all these operations to be specified by a single assembly instruction, `sub`. By looking at the operands specified with the `sub` instruction, *as* selects the appropriate M68000 opcode—i.e., either `SUB`, `SUBA`, `SUBI`, or `SUBQ`.

This could leave the misleading impression that all forms of the `SUB` operation are semantically identical, when in fact, they are not. Whereas `SUB`, `SUBI`, and `SUBQ` all affect the condition codes consistently, `SUBA` does not affect the condition codes at all. Consequently, the *as* programmer should be aware that when the destination of a `sub` instruction is an address register (which causes `sub` to be mapped to `SUBA`), the condition codes will not be affected.

Specific Forms

You are not restricted to using simplified instructions; you can use specific forms for each instruction. For example, you can use the instructions `addi`, `adda`, and `addq`, or `subi`, `suba`, or `subq`, instead of just `add` or `sub`. A specific-form instruction will *not* be overridden if the instruction doesn't agree with the type of its operand(s) or if a more efficient instruction exists. For example, the specific form `addi` is not automatically translated to another form, such as `addq`.

Invoking the Assembler

The *as*(1) assembler converts *as* assembly language programs to relocatable object code. The syntax for the *as* command is:

as [*options*] [*file*]

The *as*(1) assembler creates *relocatable* object code. To make the code *executable*, the output relocatable code file (see the "Output Object File (-o)" section below) must be linked using *ld*(1). For details on using *ld*, see the *ld*(1) page of the *HP-UX Reference.*

The *as* program (*/bin/as*) is really a driver that invokes either */bin/as10* or */bin/as20* after making a run-time check to determine the hardware processor. The `+x` and `+X` options override the run-time check and specify that */bin/as20* or */bin/as10*, respectively, be invoked regardless of the hardware. The specific assemblers can also be invoked directly, bypassing the */bin/as* driver. To bypass the */bin/as* driver, use one of these commands:

as10 [*options*] [*file*]
as20 [*options*] [*file*]

Input Source File

The *file* argument specifies the filename of the assembly language source program. Typically, assembly source files have a `.s` suffix; e.g., `my_prog.s`. If no *file* is specified or it is simply a hyphen (-), then the assembly source is read from standard input (*stdin*).

Generate Assembly Listing (-A)

Generate an assembly listing with offsets, a hexadecimal dump of the generated code, and the source text. The listing goes to standard out *(stdout)*. This option cannot be used when the input is *stdin.*

Generate Assembly Listing in Listfile (-a listfile)

Generate an assembly listing in the file *listfile.* The listing option cannot be used when the input is *stdin.* The *listfile* name cannot be of the form `*.[cs]` and cannot start with the character − or +.

Local Symbols in LST (-L)

When the `-L` option is used, entries are generated in the linker symbol table (LST) for local symbols as well as global symbols. Normally, only global and undefined symbols are entered into the LST. This is a useful option when using the *adb*(1) to debug assembly language programs.

Linker Symbol Table (-l)

Generates entries in the linker symbol table for all global and undefined symbols, and for all local symbols except those with "." or "L" as the first character. This option is useful when using tools like *prof*(1) on files generated by the *cc*(1) or *fc*(1) compilers. It generates LST entries for user-defined local names but not for compiler-generated local names.

Invoking the Macro Preprocessor (-m)

The `-m` option causes the *m4*(1) macro preprocessor to process the input file before *as* assembles it.

Short Displacement (-d)

When the `-d` flag is used with the *as20* assembler, *as20* generates short displacement forms for MC68010-compatible syntaxes, even for forward references. (For details on this option, see the "as20 Addressing Optimization" section of the "Address Mode Syntax" chapter; also see the appendix "Compatibility Issues.") The `-d` option is ignored by *as10*.

Span-dependent Optimization (-O)

Turns on span-dependent optimization. This optimization is off by default.

Output Object File (-o objfile)

When *as* is invoked with the `-o` flag, the output object code is stored in the file *objfile*. If the `-o` flag is not specified and the source is read from `stdin`, then the object file is written to *a.out*. Otherwise, if no object file is specified, output is left in the current directory in a file whose name is formed by removing the `.s` suffix (if there is one) from the assembly source base filename (*file*) and appending a `.o` suffix. To prevent accidental corruption of source files, *as* will not accept an output filename of the form `*.[cs]`. To avoid confusion with other options, *as* will not accept an output filename that starts with the character − or +.

The *as*(1) page of the HP-UX reference provides detailed information about the *as* command and its options.

Suppress Warning Messages (-w)

Warning messages are suppressed when this option is used with the assembly invocation commands (i.e., *as*, *as20*, or *as10*).

As10 Selection (+X)

This option causes */bin/as* to invoke */bin/as10*, overriding the run-time processor check. It is ignored when either *as10* or *as20* is invoked directly.

As20 Selection (+x)

This option causes */bin/as* to invoke */bin/as20*, overriding the run-time processor check. It is ignored when either *as10* or *as20* is invoked directly.

Set Version Stamp Field (-V <number>)

This option causes the *a_stamp* field in the *a.out* header (see *a.out*(4)) to be set to <number>. The `-V` option overrides any `version` pseudo-op in the assembly source. See Chapter 7.

As mentioned at the start of this section, *as* creates *relocatable* object files. Therefore, the `.o` files created by *as* use the *magic number* `RELOC_MAGIC` as defined in the */usr/include/magic.h* header file. The linker, *ld*(1), must be used to make the file *executable*. For details on the linker and magic numbers, see the following pages from the *HP-UX Reference*: *ld*(1), *a.out*(4), and *magic*(4).

Using cc(1) to Assemble

The *as* assembler can also be invoked using the C compiler, *cc*(1). Options can be passed to the assembler via the `-W a` option. For example,

```
cc -c -W a,-L file.s
```

would assemble *file.s* to generate *file.o,* with the assembler generating LST entries for local symbols.

```
cc -o cmd xyz.s abc.c
```

will compile *abc.c* and assemble *xyz.s.* The resulting `.o` files (*xyz.o* and *abc.o*) are then linked, together with *libc*, to give the executable program *cmd.*

When invoked via *cc*(1), the *cc* `+x`, `+X` options can be used to select /bin/as20 or /bin/as10. If no `+x` or `+X` is specified, *cc* will select the assembler to run based on a run-time check of the hardware.

Overview of Assembler Operation

The *as* assembler operates in two passes. Pass one parses the assembly source program. As it parses the source code, it determines operand addressing modes and assigns values to labels. The determination of the addressing mode used for each instruction is based on the information the assembler has available when the instruction is encountered. Preliminary code is generated for each instruction.

Throughout this reference, you will encounter the term **pass-one absolute**. For example, some expressions allow only pass-one absolute expressions. A pass-one absolute expression is one whose value can be determined when it is first encountered.

Pass two of *as* processes the preliminary code and label values (determined in pass one) to generate object code and relocation information. In addition, *as* generates a relocatable object file that can be linked by *ld*(1) to produce an executable object code file. If you want to know more about the format of object files generated by *ld*, see the following *HP-UX Reference* pages: *ld*(1), *a.out*(4), and *magic*(4).

Assembly Language Building Blocks 2

This chapter discusses the basic building blocks used to create assembly language programs: **identifiers**, **register identifiers**, and **constants**.

Identifiers

An *identifier* is a string of characters taken from **a-z**, **A-Z**, **0-9**, and _ (the underscore character). The first character of an identifier must be a letter (**a-z** or **A-Z**) or the underscore (_).

NOTE

Identifiers can also begin with a dot (.), although this is used primarily for certain reserved symbols used by the assembler (`.b`, `.w`, `.l`, `.s`, `.d`, `.x`, and `.p`). To avoid conflict with internal assembler symbols, you should not use identifiers that start with a dot. In addition, the names `.`, `.text`, `.data`, and `.bss` are predefined.

The dot (.) identifier is the location counter. `.text`, `.data`, and `.bss` are relocatable symbols that refer to the start of the *text*, *data*, and *bss* segments respectively. These three names are predefined for compatibility with other UNIX assemblers. (See the chapter "Segments, Location Counters, and Labels" for details on segments.)

The *as* assembler is case-sensitive; for example, `loop_35`, `Loop_35`, and `LOOP_35` are all distinct identifiers. Identifiers cannot exceed 256 characters in length.

The *as* assembler maintains two name spaces in the symbol table: one for instruction and pseudo-op mnemonics, the other for all other identifiers—user-defined symbols, special reserved symbols, and predefined assembler names. This means that a user symbol can be the same as an instruction mnemonic without conflict; for example, `addq` can be used as either a label or an instruction. However, an attempt to define a predefined identifier (e.g., using `.text` as a label) will result in a symbol redefinition error. Since all special symbols and predefined identifiers start with a dot (.), user-defined identifiers should not start with the dot.

Register Identifiers

A *register identifier* represents an MC68010, MC68020/30, or MC68881 processor register. The first character of a register identifier is the % character (the % is part of the identifier). Register identifiers are the *only* identifiers that can use the % character. In this section, register identifiers are described for the following groups of registers:

- MC68000 registers, common to the MC68000, MC68010, and MC68020/30 processors
- MC68010 registers, common to both the MC68010 and MC68020/30 processors
- MC68020/30 registers, used only by the MC68020/30 processor
- MC68881 registers, used only by the MC68881 coprocessor.
- HP 98248 Floating-Point Accelerator registers.

MC68000 Registers

Both the MC68010 and MC68020/30 processors use a common set of MC68000 registers: eight data registers; eight address registers; and condition code, program counter, stack pointer, status, user stack pointer, and frame pointer registers.

The predefined MC68000 register identifiers are:

%d0	%d4	%a0	%a4	%cc	%usp
%d1	%d5	%a1	%a5	%pc	%fp
%d2	%d6	%a2	%a6	%sp	
%d3	%d7	%a3	%a7	%sr	

Table 2-1 succinctly defines these registers.

Table 2-1. MC68000 Register Identifiers

Name	
`%d`*n*	Data Register *n*
`%a`*n*	Address Register *n*
`%cc`	Condition Code Register
`%pc`	Program Counter
`%sp`	Stack Pointer (this is `%a7`)
`%sr`	Status Register
`%usp`	User Stack Pointer
`%fp`	Frame Pointer Address Register (this is `%a6`)

MC68010 Registers

In addition to the MC68000 registers, the MC68010 processor supports the registers shown in Table 2-2.

Table 2-2. MC68010 Register Identifiers

Name	
`%sfc`	Source Function Code Register
`%dfc`	Destination Function Code Register
`%vbr`	Vector Base Register

MC68020/30 Registers

The entire register set of the MC68000 and MC68010 is included in the MC68020/30 register set. Table 2-3 shows additional control registers available on the MC68020/30 processor.

Table 2-3. MC68020/30 Control Register Identifiers

Name	
`%caar`	Cache Address Register
`%cacr`	Cache Control Register
`%isp`	Interrupt Stack Pointer
`%msp`	Master Stack Pointer

Various addressing modes of the MC68020/30 allow the registers to be **suppressed** (not used) in the address calculation. Syntactically, this can be specified either by omitting a register from the address syntax or by explicitly specifying a **suppressed register** (also known as a **zero register**) identifier in the address syntax. Table 2-4 defines the register identifiers that can be used to specify a suppressed register.

Table 2-4. Suppressed (Zero) Registers

Name	
`%zdn`	Suppressed Data Register *n*, where *n* is the register number (0 through 7)
`%zan`	Suppressed Address Register *n*, where *n* is the register number (0 through 7)
`%zpc`	Suppressed Program Counter

MC68881 Registers

Table 2-5 defines the register identifiers for the MC68881 floating-point coprocessor.

Table 2-5. MC68881 Register Identifiers

Name	Description
`%fp0`, `%fp1`, `%fp2`, `%fp3`, `%fp4`, `%fp5`, `%fp6`, `%fp7`	Floating Point Data Registers
`%fpcr`	Floating Point Control Register
`%fpsr`	Floating Point Status Register
`%fpiar`	Floating Point Instruction Address Register

Floating-Point Accelerator Registers

Table 2-6 defines the register identifiers for the floating-point accelerator.

Table 2-6. Floating-Point Accelerator Registers

Name	Description
`%fpa0` – `%fpa15`	Floating Point Data Registers
`%fpacr`	Floating Point Control Register
`%fpasr`	Floating Point Status Register

Constants

The *as* assembler allows you to use **integer**, **character**, **string**, and **floating point** constants.

Integer Constants

Integer constants can be represented as either decimal (base 10), octal (base 8), or hexadecimal (base 16) values. A **decimal** constant is a string of digits (0-9) starting with a non-zero digit (1-9). An **octal** constant is a string of digits (0-7) starting with a zero (0). A hexadecimal constant is a string of digits and letters (0-9, a-f, and A-F) starting with **0x** or **0X** (zero X). In hexadecimal constants, upper- and lower-case letters are not distinguished.

The *as* assembler stores integer constants internally as 32-bit values. When calculating the value of an integer constant, overflow is not detected.

Following are example decimal, octal, and hexadecimal constants:

`35`	Decimal 35
`035`	Octal 35 (Decimal 29)
`0X35`	Hexadecimal 35 (Decimal 53)
`0xfF`	Hexadecimal ff (Decimal 255)

Character Constants

An ordinary character constant consists of a single-quote character (') followed by an arbitrary ASCII character other than the backslash (\), which is reserved for specifying **special characters**. Character constants yield an integer value equivalent to the ASCII code for the character; because they yield an integer value, they can be used anywhere an integer constant can. The following are all valid character constants:

Constant	Value
'0	Digit Zero
'A	Upper-Case A
'a	Lower-Case a
'\'	Single-Quote Character (see following description of special characters)

A special character consists of \ followed by another character. All special characters are listed in Table 2-6.

Table 2-6. Special Characters

Constant	Value	Meaning
`\b`	0x08	Backspace
`\t`	0x09	Horizontal Tab
`\n`	0x0a	Newline (Line Feed)
`\v`	0x0b	Vertical Tab
`\f`	0x0c	Form Feed
`\r`	0x0d	Carriage Return
`\\`	0x5c	Backslash
`\'`	0x27	Single Quote
`\"`	0x22	Double Quote

Note: If the backslash precedes a character other than the special characters shown in Table 2-6, then the character is produced. For example, `\A` is equivalent to `A`; `\=` is equivalent to `=`; and so on.

In addition to the special characters shown in Table 2-6, you can optionally represent any character by following the backslash with an octal number containing up to three digits:

ddd

For example, `\11` represents the horizontal tab (`\t`); `\0` represents the NULL character; and `\377` represents the value 255, whatever character it may be.

String Constants

A **string** consists of a sequence of characters enclosed in double quotes. String constants can be used only with the **byte** and **asciz** pseudo-ops, described in the "Pseudo-Ops" chapter.

Special characters (see Table 2-6) can be imbedded anywhere in a string. A double-quote character *within* a string must be preceded by the \ character.

Strings may contain no more than 256 characters.

String constants can be continued across lines by ending nonterminating line(s) with the \ character. Spaces at the start of a continued line are significant and will be included in the string. For example,

```
#
# The following lines start in the first column.
#
byte "This\
 string \
contains a double-quote (\") character."
```

produces the string:

```
This string contains a double-quote (") character.
```

Floating-Point Constants

Floating-point constants can only be used as either:

- immediate operands to MC68881 floating-point instructions, or
- as the operand of one of the following data-allocation pseudo-ops: `float`, `double`, `extend`, and `packed`.

A floating-point constant starts with **0f** (zero f) or **0F** and is followed by a string of digits containing an optional decimal point and followed by an optional exponent. The floating-point data formats are described in the *MC68881 User's Manual.* The following are examples of floating-point constants:

```
        fadd.d      &0f1.2e+02,%fp1   # the constant is "double"¹
                                      # inferred from instr. size (.d)
        float       0f-1.2e3
```

The type of a floating-point constant (`float`, `double`, `extend`, or `packed`) is determined by the pseudo-op used or, for immediate operands, by the operation size (`.s`, `.d`, `.x`, or `.p`). When a floating-point constant is used as an immediate operand to an instruction, an operation size *must* be specified in order to define the type of the constant.

Floating-point constants are converted to IEEE floating-point formats using the *cvtnum(3C)* routine. (See the *cvtnum(3C)* page in the *HP-UX Reference* for details.) The rounding modes can be set with the `fpmode` pseudo-op. Also, *special* IEEE numbers can be specified with the NAN (Not A Number) and INF (INFinity) syntaxes:

```
0finf
0fNan(abcdeeo)
```

[1] The "&" operator in the floating-point constant example specifies to *as* that the floating-point constant is an immediate operand. For details, see the chapter "Addressing Mode Syntax."

Notes

Assembly Language Syntax 3

This chapter discusses the syntax of *as* assembly language programs.

Syntax of the Assembly Language Line

Assembly language source lines conform to the following syntax:

[*label*]... [*statement*] [*comment*]

An assembly language line is comprised of up to three main parts: **label**, **statement**, and **comment**. Each part is optional (as denoted by the brackets [ ]). Therefore, a line can be entirely blank (no parts present), or it may contain any combination of the parts in the specified order. A line can also have more than one label.

Labels, statements, and comments are separated by white space (i.e., any number of spaces or tabs), and there can also be white space before labels.

Note: We recommend that you use tabs to align the columns of your assembly language programs. This is not required by the assembler. However, it does make your programs easier to read.

Labels

A label is an identifier followed by a colon (although the colon is not considered to be part of the label). A label can be preceded by white space. There can be more than one label per line. (This feature is used primarily by compilers.)

Labels can precede any instruction or pseudo-op, except the `text`, `data`, and `bss` pseudo-ops. For details on label types and label values, see the chapter "Segments, Location Counters, and Labels."

Statements

A *statement* consists of an MC68010/20/30 opcode or an *as* pseudo-op and its operand(s), if any:

statement == *opcode* [*operand* [,*operand*]...]

Several *statements* can appear on the same line, but they must be separated by semicolons:

statement [; *statement*]...

Comments

The # character signifies the start of a comment. Comments are ignored by the assembler. Comments start at the # character and continue to the end of the line. A # character within a string or character constant does **not** start a comment.

NOTE

Some users invoke *cpp(1)* to make use of macro capabilities. In such cases, care should be taken not to start comments with the # in column one because the # in column one has special meaning to *cpp*.

Segments, Location Counters, and Labels 4

This chapter discusses **segments**, **location counters**, and their relation to **labels**.

Segments

An *as* assembly language program may be divided into separate sections known as **segments**. Three segments exist in *as* assembly language: **text**, **data**, and **bss**. The resulting object code from *as* assembly is the concatenation of the *text*, *data*, and *bss* segments.

By convention, instructions are placed in the *text* segment; initialized data is placed in the *data* segment; and storage for uninitialized data is allocated in the *bss* segment. By default, *as* begins assembly in the text segment.

Instructions and data can be intermixed in either the *text* or *data* segment, but **only uninitialized data can be allocated in the bss segment.**

The pseudo-ops **text**, **data**, and **bss** cause *as* to switch to the specified segment. You can switch between different segments as often as needed. These pseudo-ops are discussed in the "Pseudo-Ops" chapter.

NOTE

The *as* assembler also maintains **dntt**, **slt**, and **vt** segments for support of the symbolic debugger, *cdb(1)*. These are generated, for example, when the *cc(1)* compiler is invoked with the **-g** option. These segments are mainly for compiler use and are not generally of interest to *as* programmers.

Location Counters

The assembler maintains separate **location counters** for the *text*, *data*, and *bss* segments. The location counter for a given segment is incremented by one for each byte generated in that segment.

The symbol dot (.) is a predefined identifier which represents the value of the location counter in the current segment. It can be used as an operand for an instruction or a data-allocation pseudo-op. For example:

```
        text
        jmp     .       # this is an infinite loop
```

Or,

```
        data
x:      long    ., ., .
```

When allocating data, as in the second example, the location counter is updated after every data item. So the second example is equivalent to:

```
        data
x:      long    x, x+4, x+8    # "long" data items consume 4 bytes each
```

Labels

A label has an associated segment and value. A label's segment is equivalent to the segment in which the label is defined. A label's value is taken from the location counter for the segment. Thus, a label represents a memory location relative to the beginning of a particular segment.

A label is associated with the next assembly instruction or pseudo-op that follows it, even if it is separated by comments or newlines. If the instruction or pseudo-op which follows a label causes any implicit alignment to certain memory boundaries (e.g., instructions are always aligned to even addresses), the *location counter is updated before the label's value is assigned.* Explicit assignments using the `lalign` pseuo-op occur *after* the label value is set.

The following example should help clarify what a label's segment and value are:

```
#
# Switch to the data segment and enter the first initialized
#    data into it:
#
        data
x:      long      0x1234      # allocate 4 bytes for this number
        byte      2           # allocate 1 byte for this number
y:                            # now initialize the variable "y"
z:      long      0xabcd
```

Assuming these lines are the first statements in the *data* segment, then label `x` is in the *data* segment and has value 0; labels `y` and `z` are also in the *data* segment and each has value 6 (because the `long` pseudo-op causes implicit alignment to even addresses, i.e., word boundaries). Note that both `y` and `z` are labels to the `long` pseudo-op.

Padding or filler bytes generated by implicit alignment are initialized to zeroes.

Notes

Expressions 5

This chapter discusses *as* assembly language **expressions**. An expression can be extremely simple; for example, it can be a single constant value. Expressions can also be complex, comprised of many operators (e.g., +, −, *, /) and operands (constants and identifiers).

Expression Types

All identifiers and expressions in an *as* program have an associated **type**, which can be one of the following:

- **absolute**
- **relocatable**
- **external**.

Absolute

In the simplest case, an expression or identifier may have an **absolute** value, such as 56, −9000, or 256 318. All constants are absolute expressions. Identifiers used as labels cannot have an absolute value because they are relative to a segment. However, other identifiers (e.g., those whose values are assigned via the `set` pseudo-op) can have absolute values.

Relocatable

Any expression or identifier may have a value relative to the start of a segment. Such a value is known as a **relocatable** value. The memory location represented by such an expression cannot be known at assembly time, but the relative values of two such expressions (i.e., the difference between them) can be known if they are in the same segment.

Identifiers used as labels have *relocatable* values.

External

If an identifier is never assigned a value, it is assumed to be an **undefined external**. Such identifiers may be used with the expectation that their values will be defined in another program, and therefore known at link time; but the relative value of *undefined externals* cannot be known.

Expression Rules

The basic building blocks of expressions are *operators*, *constants*, and *identifiers*. Table 5-1 shows all the operators supported by *as*.

Table 5-1. Expression Operators

Unary Operators

Op	Description
+	Unary Plus (no-op)
–	Negation
~	1's Complement (Bitwise Negate)

Binary Operators

<table>
<tr><th>Op</th><th>Description</th></tr>
<tr><td>+</td><td>Addition</td></tr>
<tr><td>–</td><td>Subtraction</td></tr>
<tr><td>*</td><td>Multiplication</td></tr>
<tr><td>/[1]</td><td>Division</td></tr>
<tr><td>@[1]</td><td>Modulo</td></tr>
<tr><td>></td><td>Bit Shift Right</td></tr>
<tr><td><</td><td>Bit Shift Left</td></tr>
<tr><td>&</td><td>Bitwise AND</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td>^</td><td>Bitwise Exclusive-OR</td></tr>
</table>

[1] If the result of a division is a non-integer, truncation is performed so that the sign of the remainder is the same as the sign of the quotient.

Using the following abbreviations:

- *expr* — expression
- *unop* — unary operator
- *binop* — binary operator
- *const* — constant
- *id* — identifier

expressions can be constructed from the following rule:

expr == *const*
id
unop expr
expr binop expr
(*expr*)

Note that the definition is recursive; that is, expressions can be built from other expressions. All of the following are valid expressions:

```
0x7ffa091c
125
Default_X_Col
- 1
BitMask & 0x3fc9                    # BitMask must be absolute.
(0)
(MinValue + X_offset) * ArraySize   # MinValue, X_offset, and ArraySize all
                                    #   must be absolute.
```

Precedence and Associativity Rules

To resolve the ambiguity of the evaluation of expressions, the following precedence rules are used:

unary + − ~	HIGHEST
* / @	
+ −	
< >	
&	
^	
|	LOWEST

Parentheses ( ) are used to override the precedence of operators. Unary operators group (associate) right-to-left; binary operators group left-to-right. Note that the precedence rules agree with those of the C programming language.

Determining Expression Type

The resulting type of an expression depends on the type of its operand(s). Using the following notation:

- **abs** — integer absolute expression
- **rel** — relocatable expression
- **ext** — undefined external
- **dabs** — double floating point constant
- **fabs** — floating point constant (**float**, **extend**, or **packed**).

The resulting expression type is determined as follows:

abs *binop* **abs** = **abs**
unop **abs** = **abs**

dabs *binop* **dabs** = **dabs** (where *binop* can be +, –, *, /)
unop **dabs** = **dabs** (where *unop* can be +, –)

fabs (**fabs** expressions are limited to single constants)

abs + **rel** = **rel**
rel + **abs** = **rel**
rel – **abs** = **rel**

abs + **ext** = **ext**
ext + **abs** = **ext**
ext – **abs** = **ext**

rel – **rel** = **abs** (provided both **rel** expressions are relative to the same segment)

Absolute integer constants are stored internally as 32-bit signed integer values. Evaluation of absolute integer expressions uses 32-bit signed integer arithmetic. Integer overflow is not detected.

Note: The value of a **rel** – **rel** expression can be computed **only when** the values of both **rel** expressions are known. Therefore, a **rel** – **rel** expression can appear in a larger expression (e.g., **rel** – **rel** + **abs**) **only if** both **rel**s are defined before the expression occurs; this is so that the assembler can do the subtraction during pass one. If either of the **rel**s is not defined prior to a **rel** – **rel** subtraction, the calculation is delayed until pass two; then the expression can be no more complex than **identifier** – **identifier**.

When the `-O` option is used to turn on span-dependent optimization, all subtraction calculations of *text* symbols (labels defined in the *text* segment) are normally delayed until pass two since the final segment relative offset of a *text* symbol cannot be determined in pass one. This means that expressions involving subtraction of *text* symbols are limited to *identifier* - *identifier*. This default can be overridden with the **`allow_p1sub`** pseudo-op which directs the assembler to compute subtractions in pass one even if the symbols are *text* symbols. The difference will be calculated using the (preliminary) pass one values of the symbols; the two labels in such a subtraction (*label1* - *label2*) should not be separated by any code operations that will be modified by span-dependent optimization (see Span-Dependent Optimization in Chapter 6 and a description of *allow_p1sub* pseudo-op in Chapter 7).

Expressions must evaluate to absolute numbers or simple relocatable quantities; that is, *identifier* [± *absolute*]. Complex relocation (i.e., expressions with more than one non-absolute symbol other than the *identifier* – *identifier* form) is not permitted, even in intermediate results. Thus, even though expressions like `(rel1 – rel2) + (rel3 – rel4)` are legal (if all `rel`*i* are in the same segment and defined prior to the expression), expressions such as `(rel1 + rel2) – (rel3 + rel4)` are not.

Since expression evaluation is done during pass one, an expression (and every intermediate result of the expression) must be reducible to an absolute number or simple relocatable form (i.e., *identifier* [± *offset*] or *identifier* – *identifier*) at pass one. This means that other than the special form *identifier* – *identifier*, an expression can contain at most one forward-referenced symbol.

For example, the following code stores a NULL-terminated string in the *data* segment and stores the length of the string in the memory location `login_prompt_length`. The string length (not including the terminating NULL) is computed by subtracting the relative values of two labels (`login_prompt_end` – `login_prompt`) and subtracting 1 (for the terminating NULL). This is valid because both labels are defined *prior* to the subtraction in which they are used.

```
                        data
login_prompt:           byte    "Login Name: ",0
login_prompt_end:       space   0

     # The "space" pseudo-op above causes the label "login_prompt_end"
     # to have the value of the location counter.  If this was not included,
     # the label would be associated with the following "short" pseudo-op,
     # which has implicit word-alignment, and which might cause an invalid
     # value in the "login_prompt_length" calculation.

login_prompt_length:    short   login_prompt_end - login_prompt - 1
```

The next code example contains an invalid expression, because:

1. The expression uses two as-yet-unencountered relative expressions, `exit_prompt` and `exit_prompt_len`.
2. Secondly, the computed expression (`exit_prompt_end` – `exit_prompt` – 1) is too complex because of the "– 1". Expressions that use as-yet-unencountered relative expressions cannot be any more complex than **identifier – identifier**.

```
                        data
exit_prompt_len:        short   exit_prompt_end - exit_prompt - 1
exit_prompt:            byte    "Good-Bye\n",0
exit_prompt_end:        space   0
```

To solve this problem, you could rewrite the above code as:

```
                        data
exit_prompt_len:        short   exit_prompt_end - exit_prompt - 1
exit_prompt:            byte    "Good-Bye\n",0
exit_prompt_end:        byte    0
```

Notice that the `exit_prompt_len` expression has been reduced to a **rel** − **rel** expression, `exit_prompt_end − exit_prompt`.

Pass-One Absolute Expressions

Throughout this reference you will encounter the term **pass-one absolute expression**. For example, some pseudo-op and instruction arguments must be pass-one absolute expressions. A pass-one absolute expression is one which can be reduced to an absolute number in pass one of the assembly. A pass-one absolute expression cannot contain any forward references.

Pass-One Absolute Expressions and Span-Dependent Optimization

A pass-one expression cannot contain any forward references. When the `-O` option is used, a symbol subtraction of two *text* symbols (*identifier* - *identifier*) is not pass-one absolute because all subtraction calculations for *text* symbols are delayed until pass two. This can cause problems in a program segment like the following:

```
    text
Lstart:  long 100, 101
         .
         .
Lend:    lalign 1               # no effect except to define the
                                # label Lend.
Lsize:   long (Lend - Lstart)/4 # number of table entries
```

Tegment would assemble correctly if the `-O` option is not used, but the calculation (Lend - Lstart)/4 would give a syntax error if the `-O` option is used because the expression would be too complex.

This can be remedied by either moving the table declarations to the *data* segment, or by using the *allow_p1sub* pseudo-op. The *allow_p1sub* pseudo-op directs the assembler to perform pass one subtractions where possible even for *text* symbols. The subtractions are performed using pass one values; the labels should not be separated by any code that will be modified by span-dependent optimization (see Span-Dependent Optimization in Chapter 6 and a description of *allow_p1sub* pseudo-op in Chapter 7).

Floating-Point Expressions

Floating-point constants can be *float* (single-precision), *double*, *extended*, or *packed.* The particular kind of floating-point constant generated by *as* is determined by the context in which the constant occurs. (See the **float**, **double**, **extend**, and **packed** pseudo-ops in the "Pseudo-Ops" chapter.)

When used with the **float**, **extend**, or **packed** pseudo-ops, floating-point expressions are restricted to a single constant; for example:

```
        float        0f1.23e10
```

Double floating-point expressions can be built using the unary operators + and −, and the binary operators +, −, /, and *. Double expressions are evaluated using C-like double arithmetic. The following shows a double expression:

```
        double       0f1.2 * 0f3.4 + 0f.6
```

Span-Dependent Optimization 6

The MC680xx branching instructions (`bra`, `bsr`, `bCC`) have a PC-relative address operand. The size of the operand needed depends on the distance between the instruction and its target. Choosing the smallest form is called span-dependent optimization.

Using the -O Option

The assembler `-O` option enables span-dependent optimization in the assembler. By default, span-dependent optimization is not enabled.[1] When the `-O` option is enabled, the *as10* and *as20* assemblers will attempt to optimize the PC-relative offset for the instructions shown in Table 6-1.

Table 6-1.

as10	as20
bCC	bCC
bra	bra
bsr	bsr
	fbFPCC (68881)
	fpbCC (FPA)

Span-dependent optimizations are performed only within the text segment and affect only instructions that do not have an explicit size suffix. Any instruction with an explicit size suffix is assembled according to the specified size suffix and is not optimized.

[1] When compiling C or Fortran programs with the *cc(1)* or *f77(1)* compilers using the `-O` compiler option, the peephole optimizer (/lib/c2) does the span-dependent optimization rather than the assembler. A C or Fortran program should not be compiled with the `-Wa,-O` option.

The *as20* assembler chooses between `.b`, `.w`, and `.l` operations. The *as10* assembler chooses between `.b` and `.w` operations; when a `.w` offset is not sufficient, the *as10* assembler uses equivalent instructions to provide the effect of a long offset. This means that a program that fails to assemble with the *as10* because of branch offsets that are longer than a word may assemble when *as10* `-O` is used.[2]

Tables 6-2 and 6-3 show the span-dependent optimizations performed by the *as10* and *as20* assemblers, respectively.

Table 6-2. as10 Span-Dependent Optimizations

Instruction	Byte Form	Word Form	Long Form
`br, bra, bsr`	byte offset	word offset	jmp or jsr with absolute long address
`bCC`	byte offset	word offset	byte offset conditional branch with reversed condition around jmp with absolute long address

Note

A byte branch offset cannot be zero (i.e., branch to the following address). A `br`, `bra`, or `bCC` to the following address is optimized to a `nop`. A `bsr` to the following address uses a word offset.

[2] When a branch is too long to fit in the given offset, you will get an error message similar to **`as error: "x.s" line 120: branch displacement too large: try -O assembler option (compiler option -Wa,-O) (with no size on branch statement)`**. If you are using *as10* and the offset is already word sized, then try using the `-O` option and remove the `.w` suffix from the branch instruction.

Table 6-3. as20 Span-Dependent Optimizations

Instruction	Byte Form	Word Form	Long Form
`br, bra, bsr`	byte offset	word offset	long offset
`bCC`	byte offset	word offset	long offset
`fbCC`	–	word offset	long offset
`fpbCC`	byte offset	word offset	long offset

Note

A byte branch offset cannot be zero (i.e., branch to the following address). A `br`, `bra`, or `bCC` to the following address is optimized to a `nop`. A `bsr` to the following address uses a word offset. The FPA `fpbCC` optimization refers to optimizing the implied 68020 branch (see *FPA Manual*).

The following programs show original assembly source and the corresponding code produced by span-dependent optimization. The first program shows the optimizations performed by *as20*:

```
                 Effective Code
               after optimization
   Original Code              with as20

     bcs     L1                     nop
L1:  add     %d0,%d1          L1:  add     %d0,%d1

     bne     L2                    bne.b   L2
     bra     L2                    bra.b   L2
     bsr     L2                    bsr.b   L2
     space   80                    space   80
L2:  add     %d0,%d1          L2:  add     %d0,%d1

     beq     L3                    beq.w   L3
     bra     L3                    bra.w   L3
     bsr     L3                    bsr.w   L3
     space   2000                  space   2000
L3:  add     %d0,%d1          L3:  add     %d0,%d1

     bgt     L4                    bgt.l   L4
     bra     L4                    bra.l   L4
     bsr     L4                    bsr.l   L4
     space   40000                 space   40000
L4:  add     %d0,%d1          L4:  add     %d0,%d1
```

The second program illustrates the optimizations performed by *as10*:

```
         Effective Code
             after optimization
    Original Code                  with as10

     bcs     L1                           nop
 L1: add     %d0,%d1          L1: add     %d0,%d1

     bne     L2                   bne.b   L2
     bra     L2                   bra.b   L2
     bsr     L2                   bsr.b   L2
     space   80                   space   80
 L2: add     %d0,%d1          L2: add     %d0,%d1

     beq     L3                   beq.w   L3
     bra     L3                   bra.w   L3
     bsr     L3                   bsr.w   L3
     space   2000                 space   2000
 L3: add     %d0,%d1          L3: add     %d0,%d1

     bgt     L4                   ble.l   L4x
                                  jmp     L4 #absolute.l addressing
                              L4x:
     bra     L4                   jmp     L4 #absolute.l addressing
     bsr     L4                   jsr     L4 #absolute.l addressing
            space   40000                space   40000
        L4: add     %d0,%d               L4: add     %d0,%d1
```

Restrictions When Using the -O Option

Several caveats should be followed when using the span-dependent optimization option. These are good programming practices to follow in general when programming in assembly.

When the span-dependent optimization option is enabled, branch targets should be restricted to simple labels, such as `L1`. More complex targets, such as `L1+10`, are ambiguous since the span-dependent optimizations can modify instruction sizes. A branch with a nonsimple target may not assemble as expected.

Absolute (rather than symbolic) offsets in PC-relative addressing modes should be used only where the programmer can calculate the PC offset and the offset cannot be changed by potential span-dependent optimization.

Important Recommendation

When using span-dependent optimization, limit *text* segment targets to simple labels, such as *L1*. Nonsimple targets, such as *L1+10* or PC-relative addressing with a nonsymbolic offset field should be used only when the programmer knows that the code between label *L1* and *L1+10* will always assemble to a fixed size and cannot be modified by span-dependent optimization.

Span-Dependent Optimization and Lalign

When span-dependent optimization is enabled, the assembler will preserve any even-sized *laligns* relative to the start of the *text* segment. This may result in some branch optimizations being suboptimal.

Only *laligns* of 1, 2, and 4, however, are guaranteed to be preserved by the linker (*ld*(1)). (See "A Note about lalign" in Pseudo-Op section.)

Symbol Subtractions

In normal mode, the assembler calculates symbol subtractions in pass one if both symbols are already defined. This allows more complex expressions involving symbol differences to be used.

```
Table:  long 123
        long 234
        ...
        long 231
Tend:   lalign 1                # no effect except to define Tend
Tsize:  long (Tend-Table)/4     # number of elements in Table
```

When span-dependent optimization is enabled, the assembler normally saves all symbols subtractions involving *text* segment symbols until pass two because the symbol values (*text* relative offset) will not be known until after pass one is complete and span-dependent optimization is performed. This restricts expressions involving *text* symbol differences to *identifier - identifier*. In the example program above, the line defining `Tsize` would assemble correctly if the `-O` option is not used but will generate a syntax error (`"illegal divide"`) if the `-O` option is enabled.

There are two solutions to this problem. In the above example, the code lines could be put into the *data* segment; span-dependent optimization does not affect the rules for calculating symbol differences of *data* or *bss* symbols.

The second alternative is to use the *allow_p1sub* and *end_p1sub* pseudo-ops. The *allow_p1sub* and *end_p1sub* pseudo-ops bracket areas where the assembler is directed to calculate *text* symbol subtractions in pass one (provided both symbols are already defined), even though the `-O` option is enabled. The two *text* symbols in a difference *label1 - label2* should not be separated by any code that could be modified by span-dependent optimization. If the two symbols are separated by code that is optimized, the subtraction result will be wrong since it is calculated using pass one offsets.

The following code segment is similar to the code generated by the C compiler for a switch statement. It has been modified to calculate a *Lswitch_limit* for the size of the switch table (the compiler generates an in-line constant instead). The line defining *Lswitch_limit* is bracketed by *allow_p1sub* and *end_p1sub* so that the subtraction will be done in pass one and the complex expression will be accepted by the assembler. The pass one subtraction is valid since labels *L22* and *Lswitch_end* are separated only by long pseudo-ops which cannot change in size during span-dependent optimization.

```
         subq.l  &0x1,%d0
         cmp.l   %d0,Lswitch_limit
         bhi.l   L21
         mov.l   (L22,%za0,%d0.l*4),%d0
         jmp     2(%pc,%d0.l)
L23:
         lalign  4
L22:
         long    L15-L23
         long    L16-L23
         long    L17-L23
         long    L18-L23
         long    L19-L23
         long    L20-L23
Lswitch_end:     lalign 1
         allow_p1sub
Lswitch_limit:   (Lswitch_end-L22)/4 - 1
         end_p1sub
L13:
```

Pseudo-Ops

7

The *as* assembler supports a number of **pseudo-ops**. A *psuedo-op* is a special instruction that directs the assembler to do one of the following:

- select segments
- initialize data
- define symbols
- align within the current segment.
- floating-point directives
- span-dependent directives for expression calculation
- set the `a_stamp` field in the `a.out` header

Segment Selection Pseudo-Ops

You can control in which segment code and/or data is generated via **segment selection** pseudo-ops. Table 7-1 describes the three segment selection pseudo-ops.

Table 7-1. Segment Selection Pseudo-Ops

Pseudo-Op	Description
text	Causes the *text* segment to be the current segment—i.e., all subsequent assembly output (until the next segment selection pseudo-op) is generated in the *text* segment. By default, assembly begins in the *text* segment.
data	Causes the *data* segment to be the current segment—i.e., any subsequent assembly is placed in the *data* segment.
bss	Causes the *bss* segment to be the current segment. The *bss* segment is reserved for uninitialized data only. Attempting to assemble code or data definition pseudo-ops (e.g., `long`, `byte`, etc) results in an error. The only data-allocation pseudo-ops that should be used in the *bss* segment are **space** and **lcomm**.

An assembly program can switch between different segments any number of times. In other words, you can have a program that switches back and forth between different segments, such as:

```
text
   ⋮
   assembly code for the text segment
   ⋮
data
   ⋮
   put some initialized data here in the data segment
   ⋮
bss
   ⋮
   allocate some space for an array in the bss segment
   ⋮
text
   ⋮
   more assembly code in the text segment
   ⋮
data
   ⋮
   more initialized data in the data segment
   ⋮
```

Data Initialization Pseudo-Ops

Table 7-2 lists all *data initialization pseudo-ops.* Data initialization pseudo-ops allocate the appropriate space and assign values for data to be used by the assembly language program. Data is allocated in the current segment.

Table 7-2. Data Initialization Pseudo-Ops

Pseudo-Op	Description
byte *iexpr*|*string*[,...]	The **byte** pseudo-op allocates successive bytes of data in the assembly output from a specified list of integer expressions (*iexpr*) and/or string constants (*string*). The *iexpr* can be absolute, relocatable, or external. However, only the low-order byte of each relocatable or external *iexpr* is stored. A *string* operand generates successive bytes of data for each character in the *string*; *as* does **not** append the string with a terminating NULL character.
short *iexpr*[,...]	The **short** psuedo-op generates 16-bit data aligned on word (16-bit) boundaries from a list of integer expressions (*iexpr*). The *iexpr* can be absolute, relocatable, or external. However, only the low-order 16-bit word of each relocatable or external *iexpr* is stored.
long *iexpr*[,...]	The **long** pseudo-op generates 32-bit data from a list of one or more integer expressions (*iexpr*) separated by commas. Data is generated on word (16-bit) boundaries. An *iexpr* can be absolute, relocatable, or external.
asciz *string*	The **asciz** pseudo-op puts a null-terminated *string* into the assembly output: one byte is generated for each character, and the string is appended with a zero byte.
float *fexpr*[,...]	Generates single-precision (32-bit) floating point values[1] from the specified list of one or more absolute floating point expressions (*fexpr*). Data is stored on word (16-bit) boundaries. Only simple floating point constants are allowed.

Table 7-2. Data Initialization Pseudo-Ops (continued)

Pseudo-Op	Description
double *fexpr*[,...]	Generates double-precision (64-bit) floating point values[1] from the specified list of one or more absolute floating point expressions (*fexpr*). Data is stored on word (16-bit) boundaries.
packed *fexpr*[,...]	Generates word-aligned, packed floating point values[1] (12 bytes each) from the list of floating point expressions. Only simple floating point constants are allowed for *fexpr*.
extend *fexpr*[,...]	Generates word-aligned, extended floating point values[1] (12 bytes each) from the list of floating point expressions. Only simple floating point constants are allowed for *fexpr*.
space *abs*	When used within the *data* or *text* segment, this pseudo-op generates *abs* bytes of zeroes in the assembly output, where *abs* is a pass-one absolute integer expression ≥ 0. When used in the *bss* segment, it allocates *abs* number of bytes for uninitialized data. This data space is not actually allocated until the program is loaded.
lcomm *identifier,size,align*	Allocate *size* bytes within *bss*, after aligning to *align* within the *bss* assembly segment. Both *size* and *align* must be absolute integer values computable on the first pass. *Size* must be ≥ 0; *align* must be > 0. **lcomm** always allocates space within *bss*, regardless of the current assembly segment; however, it does not change the current assembly segment.

[1] For **float**, **double**, **packed**, and **extend**, conversions are performed according to the IEEE floating point standard using the *cvtnum(3C)* routine. (See the *cvtnum(3C)* page of the *HP-UX Reference* for details on this routine.) The current value of *fpmode* defines the rounding mode to be used.

Symbol Definition Pseudo-Ops

Symbol definition pseudo-ops allow you to assign values to symbols (*identifiers*), define common areas, and specify symbols as global. Table 7-3 describes the symbol definition pseudo-ops.

Table 7-3. Symbol Definition Pseudo-Ops

Pseudo-Op	Description
set *id,iexpr*	Sets the value of the identifier *id* to *iexpr* which may be pass-one integer absolute or pass-one relocatable. A pass-one relocatable expression is defined as: `sym` [± *abs*] where *sym* has been defined prior to encountering the expression in pass one, and *abs* is pass-one absolute.
comm *id,abs*	Allocates a common area named *id* of size *abs* bytes. The *abs* parameter must be pass-one absolute. The linker will allocate space for it. The symbol *id* is marked as global.
global *id*[,*id*]	Declares the list of identifiers to be global symbols. The names will be placed in the linker symbol table and will be available to separately assembled **.o** files. This allows the linker (*ld*(1)) to resolve references to *id* in other programs.

Alignment Pseudo-Ops

Table 7-4 defines the two **alignment** pseudo-ops provided by *as*.

Table 7-4. Alignment Pseudo-Ops

Pseudo-Op	Description

lalign *abs*	Align modulo *abs* in the current segment. *abs* must be a pass-one absolute integer expression. The most useful forms are:
	`lalign    2`
	`lalign    4`
	within the *data* or *bss* segments. These force 16-bit (word) and 32-bit alignment, respectively, in the current segment. When used in the *data* or *text* segment, the "filler" bytes generated by the alignment are initialized to zeroes. If the statement is labeled, the label's value is assigned before the "filler" bytes are added. (See "A Note about lalign" below for details on how this pseudo-op is used.)

even	Same as *lalign 2*.
align *name,abs*	This pseudo-op creates a global symbol of type *align*. When the linker sees this symbol, it will create a hole beginning at symbol *name* whose size will be such that the next symbol will be aligned on a *abs* modulo boundary. *abs* must be a pass-one absolute integer expression. (See "A Note about align" below for details on this pseudo-op.)

A Note about lalign

The assembler concatenates *text*, *data*, and *bss* segments when forming its output (object) file. The assembler rounds each segment size up to the next multiple of four bytes, which may or may not leave unused space at the end of each segment.

When multiple object (**.o**) files are linked, *ld*(1) concatenates all *text* segments into one contiguous *text* segment, all *data* segments into one contiguous *data* segment, and all *bss* segments into one contiguous *bss* segment. Because of this, only *lalign* values of 1, 2, and 4 can be guaranteed to be preserved; any other *lalign* values cannot be guaranteed. This also applies to the `lcomm` pseudo-op.

A Note about align

The **align** pseudo-op should be used with care. Consider the following example:

```
                bss
                align   gap, 1024
Table:          space   4096
```

The **align** pseudo-op causes **Table** to be aligned on a 1Kb boundary in memory. The symbol **gap** is the address of the hole created before the start of **Table**. Because the actual alignment of **gap** is performed by the linker and **not** the assembler (the assembler assigns addresses as though the hole size were zero), any expression calculation which spans the alignment hole will yield incorrect results. For example:

```
                bss
x:              space   10
                align   gap, 1024
Table:          space   4096
Table_end:      space   0
                data
bss_size:       Table_end - x    # The assembler assumes the size of
                                 # "gap" to be zero, so this expression
                                 # will yield incorrect results.
```

Pseudo-Ops to Control Expression Calculation with Span-Dependent Optimization

Table 7-5 describes pseudo-ops provided to control pass one symbol subtraction calculations when the `-O` (span-dependent optimization) option is used. These pseudo-ops have no effect and are ignored if the `-O` option is not in effect.

Table 7-5. Symbol Subtraction

Pseudo-Op	Description
`allow_p1sub`	Directs the assembler to perform symbol subtractions in pass one when both symbols are known, even if the symbols are *text* symbols. Two *text* symbols in a difference (*identifier1 - identifier2*) should not be separated by any code that could be modified by span-dependent optimization.
`end_p1sub`	Directs the assembler to revert to the default for subtractions when the `-O` option is used; subtractions involving *text* symbols will be delayed until pass two.

When the `-O` option is used, all subtraction calculations of *text* symbols are normally delayed until pass two since the final segment relative offset of a *text* symbol cannot be determined in pass one. This limits expressions involving the subtraction of *text* symbols to *identifier - identifier*. The *allow_p1sub* and *end_p1sub* pseudo-ops bracket areas where the assembler is directed to calculate *text* symbol subtractions in pass one provided the symbols are already defined. Two *text* symbols in a difference (*label1 - label2*) should not be separated by any code that could be modified by span-dependent optimization since the subtraction is calculated using pass one offsets.

Floating-Point Pseudo-Ops

Table 7-6 describes the floating-point pseudo-ops.

Table 7-6. Floating-Point Pseudo-Ops

Pseudo-Op	Description

fpmode *abs*	Sets the floating point mode for the conversion of floating point constants used with the **float**, **double**, **extend**, and **packed** pseudo-ops or as immediate operands to MC68881 or FPA instructions. Valid modes are defined by *cvtnum(3C)*. (See the *cvtnum(3C)* page of the *HP-UX Reference* for details on modes.) By default, the *fpmode* is initially 0 (`C_NEAR`).
	
	Valid values for *fpmode*, as defined on the *cvtnum(3C)* page of the *HP-UX Reference*, are:
	
	`0 (C_NEAR)`
	`1 (C_POS_INF)`
	`2 (C_NEG_INF)`
	`3 (C_TOZERO)`

fpid *abs*	Sets the co-processor *id-number* for the MC68881 floating point processor. By default, the *id-number* is initially 1. **This pseudo-op is available with the as20 assembler only.**
fpareg *%an*	Sets the FPA base register to be used in translating FPA pseudo instructions to memory-mapped move instructions. By default, register *%a2* is used. Note that this does **not** generate code to load the FPA base address into *%a2*. The user must explicitly load the register (see HP 98248A *Floating-Point Accelerator Reference*).

Version Pseudo-Ops

Table 7-7 describes the `version` pseudo-op. Beginning with the HP-UX 6.5 release, the assembler supports a `version` pseudo-op for setting the `a_stamp` field in the `a.out` header (see *a.out*(4)). Prior to release 6.5, this field was always set to 0 by the assembler.

Table 7-7. Version Pseudo-Ops

Pseudo-Op	Description
version *abs*	where *abs* must be a pass-one absolute integer expression. Multiple version pseudo-op's will generate a warning from the assembler and the last occurrence will be used. The `-V` <number> command line option can also be used to set the `a_stamp` field. If the `-V` command line option is used, that overrides any `version` pseudo-op in the source file.

The 68020 HP-UX compilers save and restore the non-scratch floating point registers that they use (`%fp2` through `%fp7` and `%fpa3` through `%fpa15`), and will assume that called functions will do the same. The 68010 compilers do not allocate floating point registers (there is no 68881 on the Model 310). This incompatibility with the pre-6.5 compiler conventions can cause a problem if new code allocates a floating point register and calls old code which uses that register as a scratch register.

The 6.5 compilers use the `a_stamp` field to mark the type of code being generated so that the linker (*ld*(1)) can give warning messages about possible incompatibilities with pre-6.5 object files. The `a_stamp` field is set by the compilers according to the following conventions:

0	pre-6.5 or unknown 6.5 floating point usage
1	68010 code
2	code which does not depend on new save/restore assumptions
3	68020 code which depends on called-routine save/restore of floating point registers

You should set an appropriate version value using either the `version` pseudo-op or the `-V` option.

The linker (*ld*(1)) issues a warning if an attempt is made to link a combination of `version 0` and `version 3` files. The linker warning is:

```
warning:  possible floating point incompatibility in object files
          -- recompile with +O1 (see appropriate language reference manual)
```

The assembler issues a warning if no `version` is set and floating point opcodes are used. The assembler warning is:

```
as: warning: "x.s" line  2: no version specified and floating point ops
    present; version may not be properly set (set Assembler Reference Manual)
```

Set the `a_stamp` field using `version` to an appropriate value (using `version` or `-V`) to eliminate these warnings.

If you use permanent floating point registers but do not call any routines that could corrupt those registers, you can safely include a `version 2` directive to avoid any warning messages when linking.

If no `version` pseudo-op or `-V` option is specified, the assembler sets the `a_stamp` field according to the following rules:

0	as20 invoked, floating point operations are present, and a warning message is generated
1	as10 invoked
2	as20 invoked and no floating point operations are present

CDB Support Pseudo-Ops

The *as* assembler also supports pseudo-ops for use by the C debugger, *cdb*(1). These are not of much use to *as* programmers and are shown here merely for completeness:

dntt
dnt_*TYPE*
sltnormal
sltspecial
vt

Address Mode Syntax 8

Table 8-1 summarizes the *as* syntax for MC68000, MC68010, and MC68020/30 addressing modes. Addressing modes specific to the MC68020/30 processor (and, therefore, to the *as20* assembler) are appropriately noted. All other modes can be used on all three processors.

The following conventions are used in Table 8-1:

%a*n* Address register ***n***, where ***n*** is any digit from 0 through 7.

%d*n* Data register ***n***, where ***n*** is any digit from 0 through 7.

%ri Index register ***ri*** may be any address or data register with an optional size designation (i.e., ***ri*.w** for 16 bits or ***ri*.l** for 32 bits); default size is **.w**.

scl Optional scale factor. An index register may be multiplied by the scaling factor in some addressing modes. Values for *scl* are 1, 2, 4, or 8; default is 1. For the MC68010, only the default scale factor 1 is allowed.

bd Two's complement base displacement added before indirection takes place; its size can be 16 or 32 bits. (This addressing mode is available on the MC68020/30 only.)

od Two's-complement outer displacement added as part of effective address calculation after memory indirection; its size can be 16 or 32 bits. (This addressing mode is available on the MC68020/30 only.)

d Two's complement (sign-extended) displacement added as part of the effective address calculation; its size may be 8 or 16 bits; when omitted, the assembler uses a value of zero.

%pc Program counter.

[ ] Square brackets are used to enclose an indirect expression; these characters are required where shown. (MC68020/30 Only.)

() Parentheses are used to enclose an entire effective address; these characters are required where shown.

{} Braces indicate that a scaling factor (*scl*) is optional; these characters should **not** appear where shown.

Table 8-1. Effective Address Modes

M68000 Family Notation	as Notation	Effective Address Mode	Register Encoding as20	Register Encoding as10
Dn	%dn	Data Register Direct	000/n	000/n
An	%an	Address Register Direct	001/n	001/n
(An)	(%an)	Address Register Indirect	010/n	010/n
(An)+	(%an)+	Address Register Indirect with Post-Increment	011/n	011/n
-(An)	-(%an)	Address Register Indirect with Pre-Decrement	100/n	100/n
d(An)[1]	d(%an)	Address Register Indirect *or* (d,%an) with Displacement	101/n[1] 110/n full fmt	101/n
d(An,Ri)[2]	d(%an,%ri)	Address Register Indirect *or* (d,%an, %ri) with Index Plus Displacement	110/n[2] brief fmt 110/n full fmt	110/n
(bd,An,Ri{*scl}) (MC68020/30 Only)	(bd,%an,%ri{*scl})	Address Register Direct with Index Plus Base Displacement	110/n full fmt	–
([bd,An,Ri{*scl}],od) (MC68020/30 Only)	([bd,%an,%ri{*scl}],od)	Memory indirect with Pre-Indexing plus Base and Outer Displacement	110/n full fmt	–
([bd,An],Ri{*scl},od) (MC68020/30 Only)	([bd,%an],%ri{*scl},od)	Memory indirect with Post-Indexing plus Base and Outer Displacement	110/n full fmt	–

1 If *d* is pass-one, 16-bit absolute and the base register (**%an** or **%pc** is not suppressed), then the MC68010-compatible mode is chosen; otherwise, the more general MC68020/30 full form is assumed.

2 If *d* is not pass-one 8-bit absolute, or the base register (**%an** or **%pc**) is suppressed, the more general MC68020/30 full-format form is assumed.

Table 8-1. Effective Address Modes (continued)

M68000 Family Notation	as Notation	Effective Address Mode	Register Encoding as20	Register Encoding as10
d(PC)	d(%pc)	Program Counter Indirect *or* (d,%pc) with Displacement	111/010[3] 111/011 full fmt	111/010
d(PC,Ri)	d(%pc,%ri.l)	Program Counter Direct *or* (d,%pc,%ri) with Index and Displacement	111/011[4] brief fmt 111/011 full fmt	111/011
(bd,PC,Ri{*scl})[5] (MC68020/30 Only)	(bd,%pc,%ri{*scl})	Program Counter Direct with Index and Base Displacement	111/011 full fmt	–
([bd,PC],Ri{*scl},od)[5] (MC68020/30 Only)	([bd,%pc],%ri{*scl},od)	Program Counter Memory Indirect with Post-Indexing Plus Base and Outer Displacement	111/011 full fmt	–
([bd,PC,Ri{*scl}],od)[5] (MC68020/30 Only)	([bd,%pc,%ri{*scl}],od)	Program Counter Memory Indirect with Pre-Indexing Plus Base and Outer Displacement	111/011 full fmt	–

[3] If *d* is pass-one, 16-bit absolute and the base register (**%an** or **%pc** is not suppressed), then the MC68010-compatible mode is chosen; otherwise, the more general MC68020/30 full form is assumed.

[4] If *d* is not pass-one 8-bit absolute, or the base register (**%an** or **%pc**) is suppressed, the more general MC68020/30 full-format form is assumed.

[5] The size of the *bd* and *od* displacement fields is 16 bits if the displacement is pass-one 16-bit absolute; otherwise, a 32-bit displacement is used. (For details, see the section below entitled "as20 Addressing Mode Optimization.")

Table 8-1. Effective Address Modes (continued)

M68000 Family Notation	as Notation	Effective Address Mode	Register Encoding as20	Register Encoding as10
xxx.W	xxx *or* xxx.w[6]	Absolute Short Address (*xxx* signifies an expression yielding a 16-bit memory address)	111/000	111/000
xxx.L	xxx *or* xxx.l[6]	Absolute Long Address (*xxx* signifies an expression yielding a 32-bit memory address)	111/001	111/001
#xxx	&xxx	Immediate data (*xxx* signifies a constant expression)	111/100	111/100

[6] If no size suffix is specified for an absolute address, the assembler will use absolute-word if *xxx* is pass-one absolute and fits in 16 bits; otherwise, absolute-long is chosen.

Notes on Addressing Modes

The components of each addressing syntax must appear in the order shown in Table 8-1.

It is important to note that expressions used for **absolute** addressing modes need not be *absolute expressions*, as described in the "Expressions" chapter. Although the addresses used in those addressing modes must ultimately be filled-in with constants, that can be done later by the linker, *ld(1)*. There is no need for the assembler to be able to compute them. Indeed, the **Absolute Long** addressing mode is commonly used for accessing *undefined external* addresses.

Address components which are expressions (**bd**, **od**, **d**, absolute, and immediate) can, in general, be absolute, relocatable, or external expressions. Relocatable or external expressions generate relocation information with the final value set by the linker, *ld(1)*. It should be noted that relocation of byte- or word-sized expressions will result in truncation. The base displacement (**bd** or **d**) of a PC-relative addressing mode can be an absolute or relocatable expression, but *not* an external expression.

In Table 8-1, the index register notation should be understood as **ri.size*scale**, where both size and scale are optional. For the MC68010 processor, only the default scale factor ***1** is allowed.

Refer to Section 2 of the *M68000 Programmer's Reference Manual* for additional information about effective address modes. Section 2 of the *MC68020 32-Bit Microprocessor User's Manual* also provides information about generating effective addresses and assembler syntax.

Note that suppressed address register **%zan** can be used in place of address register **%an**; suppressed PC register **%zpc** can be used in place of **%pc**; and suppressed data register **%zdn** can be used in place of **%dn**, if suppression is desired. (This applies to MC68020/30 full-format forms only.)

Note also that an expression used as an address always generates an absolute addressing mode, even if the expression represents a location in the current assembly segment. If the expression represents a location in the current assembly segment and PC-relative addressing is desired, this must be explicitly specified as **xxx(%pc)**.

The new address modes for the MC68020/30 use two different formats of extension. The brief format provides fast indexed addressing, while the full format provides a number of options in size of displacement and indirection. The assembler will generate the brief format if the following conditions are met:

- the effective address expression is not memory indirect
- the value of displacement is within a byte and this can be determined at pass one
- no base or index suppression is specified.

Otherwise, the assembler will generate the full format.

In the MC68020/30 full-format addressing syntaxes, all the address components are optional, except that "empty" syntaxes, such as `()` or `([],10)`, are not legal. Omitted displacements are assumed to be 0; an omitted base register defaults to `%za0`; an omitted index register defaults to `%zd0`. To specify a PC-relative addressing mode with the base register (PC) suppressed, `%zpc` must be explicitly specified since an omitted base register defaults to `%za0`.

Some source code variations of the new modes may be redundant with the MC68000 address register indirect, address register indirect with displacement, and program counter with displacement modes. The assembler will select the more efficient mode when redundancy occurs. For example, when the assembler sees the form **(An)**, it will generate address register indirect mode (mode 2). The assembler will generate address register indirect with displacement (mode 5) when seeing either of the following forms (as long as **bd** is pass-1 absolute and will fit in 16 bits or less):

```
bd(An)
(bd,An)
```

For the PC-addressing modes

```
bd(PC)
bd(PC,Ri)
([bd,PC],Ri,od)
([bd,PC,Ri],od)
```

bd can either be relocatable in the current segment or absolute. If *bd* is absolute, it is taken to be the displacement value; the value is never adjusted by the assembler. If *bd* is relocatable and in the current segment, it is taken to be a target; the assembler calculates the appropriate displacement. ***bd* cannot be an external symbol or a relocatable symbol in a different segment.**

as20 Addressing Mode Optimization

As mentioned in the "Introduction" chapter, there are actually two HP-UX assemblers: *as10* for the MC68010 processor (Model 310 computers), and *as20* for the MC68020/30 and MC68881 processors (Model 320 computers). For the *as20* assembler, there are several addressing mode syntaxes that could produce either 8-, 16-, or 32-bit offsets. The *as20* assembler attempts to select the smallest displacement, based on the information it has available at pass one when an instruction is assembled.

Examples

The addressing mode syntax

(bd, %an, %ri)

will be translated to the most efficient form possible (i.e., the shortest form of the instruction possible), based on the information the assembler has available at pass one—when the assembler first encounters it.

If **bd** is pass-one absolute and fits in 8 bits (-127..128), and neither the base (**%a*n***) nor index (**%r*i***) register is suppressed, then the MC68020/30 brief format "Address Register Indirect with Index and **8-bit Displacement**" mode is chosen. (Note that if the scale factor is the default (*1), then this is a MC68010-compatible addressing mode.)

Otherwise, the MC68020/30 full format "Address Register Indirect with Index and **Base Displacement**" mode is used. The size of the Base Displacement (16- or 32-bit) is based on whether or not `bd` is pass-one absolute and if it fits in 16 bits. The following examples should help clarify:

```
#
# Example One:
#
        set     offset,10
        tst.w   (offset,%a6,%d2)        # Brief format with 8-bit
                                        # displacement is chosen.
```

In the above example, brief format with 8-bit displacement was chosen by the assembler because the value of the base displacement (in this case, `offset`) was known prior to the `tst.w` instruction (it was pass-one absolute) and neither `%a6` nor `%d2` is a suppressed register.

```
#
# Example Two:
#
        tst.w   (offset,%a6,%d2)        # Full format is used and 32 bits
                                        # are reserved for the offset.
           ⋮
        set     offset,10
```

In this example, full format is used for the instruction and a 32-bit displacement is generated, even though only 8 bits are required for the base displacement (`offset`). This is because the assembler does not know the value of `offset` before encountering the `tst.w` instruction; therefore, it cannot assume that the base displacement will fit in 8 bits.

Similarly, the addressing mode syntax

(bd, %an)

is converted to "Address Register Indirect with **16-bit Displacement**" (Mode 5) if the base displacement (**bd**) is pass-one absolute and fits in 16 bits, and if ***%an*** is not a suppressed register. Otherwise, the assembler uses a 32-bit base displacement with the equivalent form

(bd, %an, %zd0)

A similar situation holds for the displacements in PC addressing modes.

Forcing Small Displacements (-d)

Invoking *as* (*as20*) with the `-d` option forces the assembler to use the shortest form and smallest base displacement possible for all MC68010-compatible addressing modes.

For example, the addressing mode syntax

(bd, %an, %ri)

always assumes an 8-bit displacement. And,

(bd, %an)

always assumes a 16-bit displacement. In both cases the registers cannot be suppressed, and the only index scale allowed is the default *1.

Note: Refer to the "Compatibility Issues" appendix for details on using this option.

Instruction Sets 9

This chapter describes the instructions available for the MC680x0 family of processors and the MC68881 floating point coprocessor.

MC68000/10/20 Instruction Sets

Table 9-1 shows how MC68000, MC68010, and MC68020/30 instructions should be written if they are to be interpreted correctly by the *as* assembler. For details on each instruction, see the appropriate processor manual.

The entire instruction set can be used on the MC68020/30. Instructions that are MC68010/MC68020/30-only or MC68020/30-only are noted appropriately in the **Operation** column of Table 9-1. (For further details on portability, see the "Compatibility Issues" appendix.)

The following abbreviations are used in Table 9-1:

S — The letter S, as in **add.S**, stands for one of the operation size attribute letters: **b** (byte), **w** (16-bit word), or **l** (32-bit word).

A — The letter A, as in **add.A**, stands for one of the address operation size attribute letters: **w** (16-bit word), or **l** (32-bit word).

CC In the contexts **bCC**, **dbCC**, **sCC**, **tCC** and **tpCC**, the letters **CC** represent any of the following condition code designations (except that the **f** and **t** conditions may not be used in the **bCC** instruction):

cc	carry clear	lo	low (=cs)
cs	carry set	ls	low or same
eq	equal	lt	less than
f	false	mi	minus
ge	greater or equal	ne	not equal
gt	greater than	pl	plus
hi	high	t	true
hs	high or same (=cc)	vc	overflow clear
le	less or equal	vs	overflow set

EA This represents an arbitrary effective address. You should consult the appropriate reference manual for details on the addressing modes permitted for a given instruction.

I An expression used as an immediate operand.

Q A pass-one absolute expression evaluating to a number from 1 to 8.

L A label reference, or any expression, representing a memory address in the current segment.

d Two's complement or sign-extended displacement added as part of effective address calculation; size may be 8 or 16 bits; when omitted, the assembler uses a value of zero.

%dx, **%d*y***, **%d*n*** Data registers.

%ax, **%a*y***, **%a*n*** Address registers.

%*rx*, **%*ry***, **%*rn*** Represent either data or address registers.

%*rc* Represents a control register (**%sfc**, **%dfc**, **%cacr**, **%usp**, **%vbr**, **%caar**, **%msp**, **%isp**).

reglist Specifies a set of registers for the `movm` instruction. A *reglist* is a set of components (register identifiers) separated by slashes. Ranges of registers can be specified as **%a***m*-**%a***n* and/or **%d***m*-**%d***n* (where ***m*** < ***n***). For example, the following are valid *reglists*:

```
%d0/%d3
%a1/%a2/%d3-%d6
```

offset Either an immediate operand or a data register. An immediate operand must be pass-one absolute.

width Either an immediate operand or a data register. An immediate operand must be pass-one absolute.

When **I** represents a standard immediate mode effective address (i.e., MC68020/30 Mode 7, Register 4), as for the `addi` instruction, the expression can be absolute, relocatable, or external. However, when **I** represents a special immediate operand that is a field in the instruction word (e.g., for the `bkpt` instruction), then the expression must be pass-one absolute.

Table 9-1. MC680x0 Instruction Formats

Mnemonic	Assembler Syntax	Operation	Default Operation Size When None Specified
ABCD	abcd.b %dy,%dx abcd.b -(%ay),-(%ax)	Add Decimal with Extend	.b
ADD	add.S EA,%dn add.S %dn,EA	Add Binary	.w
ADDA	add.A EA,%an adda.A EA,%an	Add Address	.w
ADDI	add.S &I,EA addi.S &I,EA	Add Immediate	.w
ADDQ	add.S &Q,EA addq.S &Q,EA	Add Quick	.w
ADDX	addx.S %dy,%dxA addx.S -(%ay),-(%ax)	Add Extend	.w
AND	and.S EA,%dn and.S %dn,EA	AND Logical	.w
ANDI	and.S &I,EA andi.S &I,EA	AND Immediate	.w
ANDI to CCR	and.b &I,%cc andi.b &I,%cc	AND Immediate to Condition Codes	.b
ANDI to SR	and.w &I,%sr andi.w &I,%sr	AND Immediate to the Status Register	.w
ASL	asl.S %dx,%dy asl.S &Q,%dy asl.w &1,EA asl.w EA	Arithmetic Shift Left	.w .w
ASR	asr.S %dx,%dy asr.S &Q,%dy asr.w &1,EA asr.w EA	Arithmetic Shift Right	.w .w

Table 9-1. MC680x0 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size When None Specified
Bcc	bCC.w L	Branch Conditionally (16-Bit Displacement)	.w required
	bCC.b L	Branch Conditionally Short (8-Bit Displacement)	.b required
	bCC.l L	Branch Conditionally Long (32-Bit Displacement) (MC68020/30 Only)	.l required
	bCC L	Same as **bCC.w**[1]	.w
BCHG	bchg %dn,EA bchg &I,EA	Test a Bit and Change	.l if second operand is data register, else .b
BCLR	bclr %dn,EA bclr &I,EA	Test a Bit and Clear	.l if second operand is data register, else .b
BFCHG	bfchg EA{offset:width}	Complement Bit Field (MC68020/30 Only)	No suffix allowed
BFCLR	bfclr EA{offset:width}	Clear Bit Field (MC68020/30 Only)	No suffix allowed
BFEXTS	bfexts EA{offset:width},%dn	Extract Bit Field (Signed) (MC68020/30 Only)	No suffix allowed
BFEXTU	bfextu EA{offset:width},%dn	Extract Bit Field (Unsigned) (MC68020/30 Only)	No suffix allowed
BFFFO	bfffo EA{offset:width},%dn	Find First One in Bit Field (MC68020/30 Only)	No suffix allowed
BFINS	bfins %dn,EA{offset:width}	Insert Bit Field (MC68020/30 Only)	No suffix allowed
BFSET	bfset EA{offset:width}	Set Bit Field (MC68020/30 Only)	No suffix allowed

[1] Defaults to .w if -O option not used. When -O option is used, assembler sets the size based on the distance to the target L.

Table 9-1. MC680x0 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size When None Specified
BFTST	bftst EA{offset:width}	Test Bit Field (MC68020/30 Only)	No suffix allowed
BKPT	bkpt &I[2]	Breakpoint (MC68020/30 Only)	No suffix allowed
BRA	bra.w L br.w L	Branch Always (16-Bit Displacement)	.w required
	bra.b L br.b L	Branch Always (Short) (8-Bit Displacement)	.b required
	bra.l L br.l L	Branch Always (Long) (32-Bit Displacement) (MC68020/30 Only)	.l required
	br L	Defaults to **br.w**[3]	.w
BSET	bset %dn,EA bset &I,EA	Test a Bit and Set	.l if second operand is data register, else .b
BSR	bsr.w L	Branch to Subroutine (16-bit Displacement)	.w required
	bsr.b L	Branch to Subroutine (Short) (8-bit Displacement)	.b required
	bsr.l L	Branch to Subroutine (Long) (32-bit Displacement) (MC68020/30 Only)	.l required
	bsr L	Same as **bsr.w**[3]	.w
BTST	btst %dn,EA btst &I,EA	Test a Bit	.l if second operand is data register, else .b
CALLM	callm &I,EA	Call Module (MC68020/30 Only)	No suffix allowed

[2] The immediate operand must be a pass-one absolute expression.
[3] Defaults to .w when -O is not used. When -O option is used, the assembler sets the size based on the distance to the target L.

Table 9-1. MC680x0 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size When None Specified
CAS	cas.S %dx,%dy,EA	Compare and Swap Operands (MC68020/30 Only)	.w
CAS2	cas2.A %dx:%dy, %dx:%dy,%rx:%ry	Compare and Swap Dual Operands (MC68020/30 Only)	.w
CHK	chk.w EA,%dn	Check Register Against Bounds	.w
	chk.l EA,%dn	Check Register Against Bounds (Long) (MC68020/30 Only)	.l
CHK2	chk2.S EA,%rn	Check Register Against Bounds (MC68020/30 Only)	.w
CLR	clr.S EA	Clear an Operand	.w
CMP	cmp.S %dn,EA[4]	Compare	.w
CMPA	cmp.A %an,EA[4] cmpa.A %an,EA[4]	Compare Address	.w
CMPI	cmp.S EA,&I[4] cmpi.S EA,&I[4]	Compare Immediate	.w
CMPM	cmp.S (%ax)+,(%ay)+[4] cmpm.S (%ax)+,(%ay)+[4]	Compare Memory	.w
CMP2	cmp2.S %rn,EA[4]	Compare Register Against Bounds (MC68020/30 Only)	.w
DBcc	dbCC.w %dn,L	Test Condition, Decrement, and Branch	.w
	dbra.w %dn,L	Decrement and Branch Always	.w
	dbr.w %dn,L	Same as **dbra.w**	.w

[4] The order of the operands for this instruction is reversed from that in the *MC68000 Programmer's Reference Manual.*

Table 9-1. MC680x0 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size When None Specified
DIVS	divs.w EA,%dx	Signed Divide 32-bit ÷ 16-bit ⇒ 32-bit	.w
	tdivs.l EA,%dx divs.l EA,%dx	Signed Divide (Long) 32-bit ÷ 32-bit ⇒ 32-bit (MC68020/30 only)	.l .l required
	tdivs.l EA,%dx:%dy divsl.l EA,%dx:%dy	Signed Divide (Long) 32-bit ÷ 32-bit ⇒ 32r:32q (MC68020/30 only)	.l .l
	divs.l EA,%dx:%dy	Signed Divide (Long) 64-bit ÷ 32-bit ⇒ 32r:32q (MC68020/30 only)	.l
DIVU	divu.w EA,%dn	Unsigned Divide 32-bit ÷ 16-bit ⇒ 32-bit	.w
	tdivu.l EA,%dx divu.l EA,%dx	Unsigned Divide (Long) 32-bit ÷ 32-bit ⇒ 32-bit (MC68020/30 only)	.l .l required
	tdivu.l EA,%dx:%dy divul.l EA,%dx:%dy	Unsigned Divide (Long) 32-bit ÷ 32-bit ⇒ 32r:32q (MC68020/30 only)	.l .l
	divu.l EA,%dx:%dy	Unsigned Divide (Long) 64-bit ÷ 32-bit ⇒ 32r:32q (MC68020/30 only)	.l
EOR	eor.S %dn,EA	Exclusive OR Logical	.w
EORI	eor.S &I,EA eori.S &I,EA	Exclusive OR Logical	.w
EORI to CCR	eor.b &I,%cc eori.b &I,%cc	Exclusive OR Immediate to Condition Code Register	.b
EORI to SR	eor.w &I,%sr eori.w &I,%sr	Exclusive OR Immediate to Status Register	.w
EXG	exg.l %rx,%ry	Exchange Registers	.l

Table 9-1. MC680x0 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size When None Specified
EXT	ext.w %dn	Sign-Extend Low-Order Byte of Data to Word	.w
	ext.l %dn	Sign-Extend Low-Order Word of Data to Long	.l required
	extb.l %dn	Sign-Extend Low-Order Byte of Data to Long (MC68020/30 Only)	.l
	extw.l %dn	Same as **ext.l** (MC68020/30 Only)	.l
ILLEGAL	illegal	Take Illegal Instruction Trap	No suffix allowed
JMP	jmp EA	Jump	No suffix allowed
JSR	jsr EA	Jump to Subroutine	No suffix allowed
LEA	lea.l EA,%an	Load Effective Address	.l
LINK	link.w %an,&I	Link and Allocate	.w
	link.l %an,&I	Link and Allocate (MC68020/30 Only)	.l required
LSL	lsl.S %dx,%dy lsl.S &Q,%dy	Logical Shift Left	.w
	lsl.w &1,EA lsl.w EA		.w
LSR	lsr.S %dx,%dy lsr.S &Q,%dy	Logical Shift Right	.w
	lsr.w &1,EA lsr.w EA		.w
MOVE	mov.S EA,EA	Move Data from Source to Destination	.w
MOV to CCR	mov.w EA,%cc	Move to Condition Codes	.w
MOVE from CCR	mov.w %cc,EA	Move from Condition Codes (MC68010 and MC68020/30 Only)	.w

Table 9-1. MC680x0 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size When None Specified
MOVE to SR	mov.w EA,%sr	Move to Status Register	.w
MOVE from SR	mov.w %sr,EA	Move from Status Register	.w
MOVE USP	mov.l %usp,%an mov.l %an,%usp	Move User Stack Pointer	.l
MOVEA	mov.A EA,%an mova.A EA,%an	Move Address	.w
MOVEC to CR	mov.l %rn,%rc	Move to Control Register (MC68010 and MC68020/30 Only)	.l
MOVEC from CR	mov.l %rc,%rn	Move from Control Register (MC68010 and MC68020/30 Only)	.l
MOVEM	movm.A &I,EA movm.A EA,&I	Move Multiple Registers	.w
	movm.A reglist,EA movm.A EA,reglist	Same as above, but using the **reglist** notation.	.w
MOVEP	movp.A %dx,d(%ay) movp.A d(%ay),%dx	Move Peripheral Data	.w
MOVEQ	mov.l &I,%dn movq.l &I,%dn	Move Quick	.l
MOVES	movs.S %rn,EA movs.S EA,%rn	Move to/from Address Space (MC68010 and MC68020/30 Only)	.w
MULS	muls.w EA,%dw	Signed Multiply 16-bit $\times$ 16-bit $\Rightarrow$ 32-bit	.w
	tmuls.l EA,%dx muls.l EA,%dx	Signed Multiply (Long) 32-bit $\times$ 32-bit $\Rightarrow$ 32-bit (MC68020/30 Only)	.l .l required
	muls.l EA,%dx:%dy	Signed Multiply (Long) 32-bit $\times$ 32-bit $\Rightarrow$ 64-bit (MC68020/30 Only)	.l

Table 9-1. MC680x0 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size When None Specified
MULU	mulu.w EA,%dx	Unsigned Multiply 16-bit $\times$ 16-bit $\Rightarrow$ 32-bit	.w
	tmulu.l EA,%dx mulu.l EA,%dx	Unsigned Multiply (Long) 32-bit $\times$ 32-bit $\Rightarrow$ 32-bit (MC68020/30 Only)	.l .l required
	mulu.l EA,%dx:%dy	Unsigned Multiply (Long) 32-bit $\times$ 32-bit $\Rightarrow$ 64-bit (MC68020/30 Only)	.l
NBCD	nbcd.b EA	Negate Decimal with Extend	.b
NEG	neg.S EA	Negate	.w
NEGX	negx.S EA	Negate with Extend	.w
NOP	nop	No Operation	No suffix allowed
NOT	not.S EA	Logical Complement	.w
OR	or.S EA,%dn or.S %dn,EA	Inclusive OR Logical	.w
ORI	or.S &I,EA ori.S &I,EA	Inclusive OR Immediate	.w
ORI to CCR	or.b &I,%cc ori.b &I,%cc	Inclusive OR Immediate to Condition Codes	.b
ORI to SR	or.w &I,%sr ori.w &I,%sr	Inclusive OR Immediate to Status Register	.w
PACK	pack -(%ax),-(%ay),&I pack %dx,%dy,&I	Pack BCD (MC68020/30 Only)	No suffix allowed
PEA	pea.l EA	Push Effective Address	.l
RESET	reset	Reset External Devices	No suffix allowed
ROL	rol.S %dx,%dy rol.S &Q,%dy	Rotate (without Extend) Left	.w
	rol.w &1,EA rol.w EA		.w

Table 9-1. MC680x0 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size When None Specified
ROR	ror.S %dx,%dy ror.S &Q,%dy ror.w &1,EA ror.w EA	Rotate (without Extend) Right	.w .w
ROXL	roxl.S %dx,%dy roxl.S &Q,%dy roxl.w &1,EA roxl.w EA	Rotate with Extend Left	.w .w
ROXR	roxr.S %dx,%dy roxr.S &Q,%dy roxr.w &1,EA roxr.w EA	Rotate with Extend Right	.w .w
RTD	rtd &I	Return and Deallocate Parameters (MC68010 and MC68020/30 Only)	No suffix allowed
RTE	rte	Return from Exception	No suffix allowed
RTM	rtm %rn	Return from Module (MC68020/30 Only)	No suffix allowed
RTR	rtr	Return and Restore Condition Codes	No suffix allowed
RTS	rts	Return from Subroutine	No suffix allowed
SBCD	sbcd.b %dy,%dx sbcd.b -(%ay),-(%ax)	Subtract Decimal with Extend	.b
Scc	sCC.b EA	Set According to Condition	.b
STOP	stop &I	Load Status Register and Stop	No suffix allowed
SUB	sub.S EA,%dn sub.S %dn,EA	Subtract Binary	.w
SUBA	sub.A EA,%an suba.A EA,%an	Subtract Address	.w
SUBI	sub.S &I,EA subi.S &I,EA	Subtract Immediate	.w

Table 9-1. MC680x0 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size When None Specified
SUBQ	sub.S &Q,EA subq.S &Q,EA	Subtract Quick	.w
SUBX	subx.S %dy,%dx subx.S -(%ay),-(%ax)	Subtract with Extend	.w
SWAP	swap.w %dn	Swap Register Halves	.w
TAS	tas.b EA	Test and Set an Operand	.b
TRAP	trap &I[5]	Trap	No suffix allowed
TRAPV	trapv	Trap on Overflow	No suffix allowed
TRAPcc	tCC tpCC.A &I	Trap on Condition (MC68020/30 Only)	No suffix allowed .w
TST	tst.S EA	Test an Operand	.w
UNLK	unlk %an	Unlink	No suffix allowed
UNPK	unpk -(%ay),-(%ay), &I unpk %dx,%dy,&I	Unpack BCD (MC68020/30 Only)	No suffix allowed

5 The immediate operand must be a pass-one absolute expression.

MC68881 Instructions

Table 9-4 ("MC68881 Instruction Formats"), found at the end of this chapter, shows how the floating-point coprocessor (MC68881) instructions should be written to be understood by the *as* assembler. In Table 9-4, *FPCC* represents any of the floating-point condition code designations shown in Table 9-2.

Table 9-2. Floating-Point Condition Code Designations

Trap on Unordered

FPCC	Meaning
ge	greater than or equal
gl	greater or less than
gle	greater or less than or equal
gt	greater than
le	less than or equal
lt	less than
nge	not greater than or equal
nlt	not less than
ngl	not greater or less than
nle	not less than or equal to
ngle	not greater or less than or equal
sneq	not equal
sne	not equal
sf	never
seq	equal
st	always

No Trap on Unordered

FPCC	Meaning
eq	equal
oge	greater than or equal
ogl	greater or less than
ogt	greater than
ole	less than or equal
olt	less than
or	ordered
t	always
ule	unordered or less or equal
ult	unordered less than
uge	unordered greater than or equal
ueq	unordered equal
ugt	unordered greater than
un	unordered
neq	unordered or greater or less
ne	unordered or greater or less
f	never

In Table 9-4, the designation *ccc* represents a group of constants in MC68881 constant ROM. The values of these constants are defined in Table 9-3. (The description of the FMOVECR instruction in the *MC68881 User's Manual* provides detailed information on these constants.)

Table 9-3. MC68881 Constant ROM Values

ccc	Value
00	pi
0B	log10(2)
0C	e
0D	log2(e)
0E	log10(e)
0F	0.0
30	logn(2)
31	logn(10)
32	10**0
33	10**1
34	10**2
35	10**4
36	10**8
37	10**16
38	10**32
39	10**64
3A	10**128
3B	10**256
3C	10**512
3D	10**1024
3E	10**2048
3F	10**4096

Other abbreviations used in Table 9-4 are:

EA	Represents an effective address. See the *MC68881 User's Manual* for details on the addressing modes permitted for each instruction.
L	A label reference or any expression representing a memory address in the current segment.
I	Represents an absolute expression used as an immediate operand.
%dn	Represents a data register.
%fpm, **%fpn**, **%fpq**	Represent floating point data registers.
fpreglist	A list of floating point data registers for an `fmovm` instruction. (See description of **reglist** in the description for Table 9-1.)
%fpcr	Represents floating point control register.
%fpsr	Represents floating point status register.
%fpiar	Represents floating point instruction address register.
fpcrlist	A list of one to three floating point control register identifiers, separated by slashes (e.g., `%fpcr/%fpiar`).
&ccc	An immediate operand for the `fmover` instruction. Must be pass-one absolute.
SF	Represents source format letters; consult the *MC68881 User's Manual* for restrictions on SF in combination with the EA (effective address) mode used: b ⇒ byte integer (8 bits) w ⇒ word integer (16 bits) l ⇒ long word integer (32 bits) s ⇒ single precision d ⇒ double precision x ⇒ extend precision p ⇒ packed binary coded decimal
A	represents source format letters **w** or **l**

Note: When **.SF** is shown, a size suffix **must** be specified; there is no default size. In forms where **.x** is shown, size defaults to **.x**.

An effective address for a packed-format operation has the form

<EA>{&k}

or

<EA>{&dn}

The first form requires *k* to be a pass-one absolute value.

Table 9-4. MC68881 Instruction Formats

Mnemonic	Assembler Syntax	Operation	Default Operation Size
FABS	fabs.SF EA,%fpn fabs.x %fpm,%fpn fabs.x %fpn	Absolute Value Function	No default; give size .x .x
FACOS	facos.SF EA,%fpn facos.x %fpm,%fpn facos.x %fpn	Arcosine Function	No default; give size .x .x
FADD	fadd.SF EA,%fpn fadd.x %fpm,%fpn	Floating Point Add	No default; give size .w
FASIN	fasin.SF EA,%fpn fasin.x %fpm,%fpn fasin.x %fpn	Arcsine Function	No default; give size .x .x
FATAN	fatan.SF EA,%fpn fatan.x %fpm,%fpn fatan.x %fpn	Arctangent Function	No default; give size .x .x
FATANH	fatanh.SF EA,%fpn fatanh.x %fpm,%fpn fatanh.x %fpn	Hyperbolic Arctangent Function	No default; give size .x .x
FBfpcc	fbFPCC.A L fbr.A L fbra.A L	Co-Processor Branch Conditionally Same as **fbt**.	.w[1] .w .w
FCMP	fcmp.SF %fpn,EA[2]	Floating Point Compare	No default; give size
FCOS	fcos.SF EA,%fpn fcos.x %fpm,%fpn fcos.x %fpn	Cosine Function	No default; give size .x .x
FCOSH	fcosh.SF EA,%fpn fcosh.x %fpm,%fpn fcosh.x %fpn	Hyperbolic Cosine Function	No default; give size .x .x

[1] Defaults to `.w` if `-O` is not used. When `-O` option is used, assembler sets the size based on the distance to the target *L*.

[2] The order of the operands for the FCMP instruction is reversed from that in the *MC68881 Programmer's Reference Manual.*

Table 9-4. MC68881 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size
FDBfpcc[3]	fdbFPCC.w %dn,L fdbr.w L fdbra.w L	Decrement and Branch on Condition Same as **fdbf**.	.w .w .w
FDIV	fdiv.SF EA,%fpn fdiv.x %fpm,%fpn	Floating Point Divide	No default; give size .x
FETOX	fetox.SF EA,%fpn fetox.x %fpm,%fpn fetox.x %fpn	e**x Function	No default; give size .x .x
FETOXM1	fetoxm1.SF EA,%fpn fetoxm1.x %fpm,%fpn fetoxm1.x %fpn	e**x - 1 Function	No default; give size .x .x
FGETEXP	fgetexp.SF EA,%fpn fgetexp.x %fpm,%fpn fgetexp.x %fpn	Get the Exponent Function	No default; give a size .x .x
FGETMAN	fgetman.SF EA,%fpn fgetman.x %fpm,%fpn fgetman.x %fpn	Get the Mantissa Function	No default; give size .x .x
FINT	fint.SF EA,%fpn fint.x %fpm,%fpn fint.x %fpn	Integer Part Function	No default; give size .x .x
FINTRZ	fintrz.SF EA,%fpn fintrz.x %fpm,%fpn fintrz.x %fpn	Integer Part, Round to Zero Function	No default; give size .x .x
FLOG2	flog2.SF EA,%fpn flog2.x %fpm,%fpn flog2.x %fpn	Binary Log Function	No default; give size .x .x
FLOG10	flog10.SF EA,%fpn flog10.x %fpm,%fpn flog10.x %fpn	Common Log Function	No defualt, give size .x .x

[3] The description of the **FDB***fpcc* instruction found in the First Edition of the *MC68881 User's Manual* incorrectly states that "The value of the PC used in the branch address calculation is the address of the FDBcc instruction plus two." It should say "the address of the FDBcc instruction plus **four**." If you always reference this instruction using a label, then it should not cause any problems, as the assembler will automatically generate the correct offset.

Table 9-4. MC68881 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size
FLOGN	flogn.SF EA,%fpn flogn.x %fpm,%fpn flogn.x %fpn	Natural Log Function	No default; give size .x .x
FLOGNP1	flognp1.SF EA,%fpn flognp1.x %fpm,%fpn flognp1.x %fpn	Natural Log (x+1) Function	No default; give size .x .x
FMOD	fmod.SF EA,%fpn fmod.x %fpm,%fpn	Floating Point Modulus	No default; give size .x
FMOVE	fmov.SF EA,%fpn fmov.x %fpm,%fpn	Move to Floating Point Register	No default; give size .x
	fmov.SF %fpn,EA fmov.p %fpn,EA{%dn} fmov.p %fpn,EA{&I}[4]	Move from Floating Point Register to Memory	No default; give size .p .p
	fmov.l EA,%fpcr[5] fmov.l EA,%fpsr[5] fmov.l EA,%fpiar[5]	Move from Memory to Special Register	.l .l .l
	fmov.l %fpcr,EA[5] fmov.l %fpsr,EA[5] fmov.l %fpiar,EA[5]	Move from Special Register to Memory	.l .l .l
FMOVECR	fmovcr.x &ccc,%fpn[4]	Move a ROM-Stored to a Floating Point Register	.x

[4] The immediate operand must be a pass-one absolute expression.
[5] See the *MC68881 User's Manual* for restrictions on EA (effective address) modes with this command.

Table 9-4. MC68881 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size
FMOVEM	fmovm.x EA,&I fmovm.x EA,fpreglist fmovm.x EA,%dn	Move to Multiple Floating Point Registers	.x .x .x
	fmovm.x &I,EA fmovm.x fpreglist,EA fmovm.x %dn,EA	Move from Multiple to MC68881 Control Registers	.x .x .x
	fmovm.l EA,fpcrlist[6]	Move Multiple to MC68881 Control Registers	.l
	fmovm.l fpcrlist,EA[6]	Move from Multiple Registers Registers to Memory	.l
FMUL	fmul.SF EA,%fpn fmul.x %fpm,%fpn	Floating Point Multiply	No default; give size .x
FNEG	fneg.SF EA,%fpn fneg.x %fpm,%fpn fneg.x %fpn	Negate Function	No default; give size .x .x
FNOP	fnop	Floating Point No-Op	No suffix allowed
FREM	frem.SF EA,%fpn frem.x %fpm,%fpn	Floating Point Remainder	No default; give size .x
FRESTORE	frestore EA	Restore Internal State of Co-Processor	No suffix allowed
FSAVE	fsave EA	Save Internal State of Co-Processor	No suffix allowed
FSCALE	fscale.SF EA,%fpn fscale.x %fpm,%fpn	Floating Point Scale Exponent	No default; give size .x
FSfpcc	fsFPCC.b EA	Set on Condition	.b
FSGLDIV	fsgldiv.SF EA,%fpn fsgldiv.x %fpm,%fpn	Floating-Point Single-Precision Divide	No default; give size .x
FSGLMUL	fsglmul.SF EA,%fpn fsglmul.x %fpm,%fpn	Floating-Point Single-Precision Multiply	No default; give size .x
FSIN	fsin.SF EA,%fpn fsin.x %fpm,%fpn fsin.x %fpn	Sine Function	No default; give size .x .x

[6] See the *MC68881 User's Manual* for restrictions on EA (effective address) modes with this command.

Table 9-4. MC68881 Instruction Formats (continued)

Mnemonic	Assembler Syntax	Operation	Default Operation Size
FSINCOS	fsincos.SF EA,%fpn:%fpq fsincos.x %fpm,%fpn:%fpq	Sine/Cosine Function	No default; give size .x
FSINH	fsinh.SF EA,%fpn fsinh.x %fpm,%fpn fsinh.x %fpn	Hyperbolic Sine Function	No default; give size .x .x
FSQRT	fsqrt.SF EA,%fpn fsqrt.x %fpm,%fpn fsqrt.x %fpn	Square Root Function	No default; give size .x .x
FSUB	fsub.SF EA,%fpn fsub.x %fpm,%fpn	Floating Point Subtract	No default; give size .x
FTAN	ftan.SF EA,%fpn ftan.x %fpm,%fpn ftan.x %fpn	Tangent Function	No default; give size .x .x
FTANH	ftanh.SF EA,%fpn ftanh.x %fpm,%fpn ftanh.x %fpn	Hyperbolic Tangent Function	No default; give size .x .x
FTENTOX	ftentox.SF %fpn ftentox.x %fpm,%fpn ftentox.x %fpn	10**x Function	No default; give size .x .x
FTfpcc	ftFPCC	Trap on Condition without a Parameter	No suffix allowed
FTPfpcc	ftpFPCC.A &I	Trap on Condition with a Parameter	.w
FTEST	ftest.SF EA ftest.x %fpm	Floating Point Test an Operand	No default; give size .x
FTWOTOX	ftwotox.SF EA,%fpn ftwotox.x %fpm,%fpn ftwotox.x %fpn	2**x Function	No default; give size .x .x

FPA Macros

The table in this section entitled "FPA-Macro Formats" shows how floating-point accelerator macros are written for use with the *as* assembler.

To help you interpret the Assembler Syntax column of the following table, here is a list of notations used:

`%fpaS`	is the floating-point accelerator source.
`%fpaD`	is the floating-point accelerator destination.
`<ea>`	is the non-floating-point accelerator source.
`%fpacr`	is the floating-point accelerator control register.
`%fpasr`	is the floating-point accelerator status register.
[ ]	indicates that the text between these square brackets is optional.
SF	is a floating-point size suffix that is required where shown. `s` ⇒ single precision `d` ⇒ double precision
SB	is an MC68020/30 size suffix for a branch instruction that is optional. If this suffix is omitted and the `-O` option for span-dependent optimization was not used, the default is `.w`. However, if the `-O` option is used span-dependent optimization selects the size. `b` ⇒ byte integer (8 bits) `w` ⇒ word integer (16 bits) `l` ⇒ long word integer (32 bits)

Table 9-5. FPA-Macro Formats

Mnemonic	Assembler Syntax	Operation
FPABS	`fpabs.SF %fpaS[,%fpaD]`	absolute value of operand
FPADD	`fpadd.SF %fpaS,%fpaD`	addition
FPAREG	`fpareg %an`	resets the address register to be used as the base register
FPBEQ	`fpbeq.SB <label>`	branch if equal
FPBF	`fpbf.SB <label>`	branch if false
FPBGE	`fpbge.SB <label>`	branch if greater than or equal
FPBGL	`fpbgl.SB <label>`	branch if greater than or less than
FPBGLE	`fpbgle.SB <label>`	branch if greater than, less than, or equal
FPBGT	`fpbgt.SB <label>`	branch if greater than
FPBLE	`fpble.SB <label>`	branch if less than or equal
FPBLT	`fpblt.SB <label>`	branch if less than
FPBNE	`fpbne.SB <label>`	branch if not equal
FPBNGE	`fpbnge.SB <label>`	branch if not greater than or equal
FPBNGL	`fpbngl.SB <label>`	branch if not greater than or less than
FPBNGLE	`fpbngle.SB <label>`	branch if not greater than, less than, or equal
FPBNGT	`fpbngt.SB <label>`	branch if not greater than
FPBNLE	`fpbnle.SB <label>`	branch if not less than or equal
FPBNLT	`fpbnlt.SB <label>`	branch if not less than
FPBOGE	`fpboge.SB <label>`	branch if ordered greater than or equal
FPBOGL	`fpbogl.SB <label>`	branch if ordered greater than or less than
FPBOGT	`fpbogt.SB <label>`	branch if ordered greater than

Table 9-5. FPA-Macro Formats (continued)

Mnemonic	Assembler Syntax	Operation
FPBOLE	fpbole.SB <label>	branch if ordered less than or equal
FPBOLT	fpbolt.SB <label>	branch if ordered less than
FPBOR	fpbor.SB <label>	branch if ordered
FPBSEQ	fpbseq.SB <label>	branch if signalling equal
FPBSF	fpbsf.SB <label>	branch if signalling false
FPBSNE	fpbsne.SB <label>	branch if signalling not equal
FPBST	fpbst.SB <label>	branch if signalling true
FPBT	fpbt.SB <label>	branch if true
FPBUEQ	fpbueq.SB <label>	branch if unordered or equal
FPBUGE	fpbuge.SB <label>	branch if unordered or greater than or equal
FPBUGT	fpbugt.SB <label>	branch if unordered or greater than
FPBULE	fpbule.SB <label>	branch if unordered or less than or equal
FPBULT	fpbult.SB <label>	branch if unordered or less than
FPBUN	fbpun.SB <label>	branch if unordered
FPCMP	fpcmp.SF %fpaS,%fpaD	compare
FPCVD	fpcvd.l %fpaS[,%fpaD]	converts long word integer to double precision
FPCVD	fpcvd.s %fpaS[,%fpaD]	converts single precision to double precision
FPCVL	fpcvl.d %fpaS[,%fpaD]	converts double precision to a long word integer

Table 9-5. FPA-Macro Formats (continued)

Mnemonic	Assembler Syntax	Operation
FPCVL	`fpcvl.s %fpaS[,%fpaD]`	converts single precision to a long word integer
FPCVS	`fpcvs.d %fpaS[,%fpaD]`	converts double precision to single precision
FPCVS	`fpcvs.l %fpaS[,%fpaD]`	converts long word integer to single precision
FPDIV	`fpdiv.SF %fpaS,%fpaD`	division
FPINTRZ	`fpintrz.SF %fpaS[,%fpaD]`	rounds to integer using the round-to-zero mode
FPM2ADD	`fpm2add.SF <ea>,%fpaS,%fpaD`	combination move to destination and addition
FPM2CMP	`fpm2cmp.SF <ea>,%fpaS,%fpaD`	combination move to destination and compare
FPM2DIV	`fpm2div.SF <ea>,%fpaS,%fpaD`	combination move to destination and division
FPM2MUL	`fpm2mul.SF <ea>,%fpaS,%fpaD`	combination move to destination and multiplication
FPM2RDIV	`fpm2rdiv.SF <ea>,%fpaS,%fpaD`	combination move to destination and reverse division (i.e. source ÷ destination)
FPM2RSUB	`fpm2rsub.SF <ea>,%fpaS,%fpaD`	combination move to destination and reverse subtraction (i.e. source – destination)
FPM2SUB	`fpm2sub.SF <ea>,%fpaS,%fpaD`	combination move to destination and subtraction
FPMABS	`fpmabs.SF <ea>,%fpaS[,%fpaD]`	combination move and taking absolute value of operand
FPMADD	`fpmadd.SF <ea>,%fpaS,%fpaD`	combination move and addition
FPMCVD	`fpmcvd.l <ea>,%fpaS[,%fpaD]`	combination move and convert long word integer to double precision

Table 9-5. FPA-Macro Formats (continued)

Mnemonic	Assembler Syntax	Operation
FPMCVD	`fpmcvd.s <ea>,%fpaS[,%fpaD]`	combination move and convert single precision to double precision
FPMCVL	`fpmcvl.d <ea>,%fpaS[,%fpaD]`	combination move and convert double precision to long word integer
FPMCVL	`fpmcvl.s <ea>,%fpaS[,%fpaD]`	combination move and convert single precision to long word integer
FPMCVS	`fpmcvs.d <ea>,%fpaS[,%fpaD]`	combination move and convert double precision to single precision
FPMCVS	`fpmcvs.l <ea>,%fpaS[,%fpaD]`	combination move and convert long word integer to single precision
FPMDIV	`fpmdiv.SF <ea>,%fpaS[,%fpaD]`	combination move and division
FPMINTRZ	`fpmintrz.SF <ea>,%fpaS[,%fpaD]`	combination move and rounding to integer using round-to-zero mode
FPMMOV	`fpmmov.SF <ea>,%fpaS,%fpaD`	combined move
FPMMUL	`fpmmul.SF <ea>,%fpaS,%fpaD`	combination move and multiplication
FPMNEG	`fpmneg.SF <ea>,%fpaS[,%fpaD]`	combination move and negation

FPMOV	`fpmov.SF <ea>,%fpaD`
	`fpmov.SF %fpaS,<ea>`
	`fpmov.SF %fpaS,%fpaD`
	`fpmov.SF <ea>,%fpasr`
	`fpmov.SF %fpasr,<ea>`
	`fpmov.SF <ea>,%fpacr`
	`fpmov.SF %fpacr,<ea>`
		move from an external location
		move to an external location
		move between two FPA registers
		move to the status register
		move from the status register
		move to the control register
		move from the control register

FPMRDIV	`fpmrdiv.SF <ea>,%fpaS,%fpaD`	combination move and reverse division (i.e. source ÷ destination)
FPMRSUB	`fpmrsub.SF <ea>,%fpaS,%fpaD`	combination move and reverse subtraction (i.e. source − destination)
FPMSUB	`fpmsub.SF <ea>,%fpaS,%fpaD`	combination move and subtraction
FPMTEST	`fpmtest.SF <ea>,%fpaS`	combination move and test of operand

Table 9-5. FPA-Macro Formats (continued)

Mnemonic	Assembler Syntax	Operation
FPMUL	`fpmul.SF %fpaS,%fpaD`	multiplication
FPNEG	`fpneg.SF %fpaS[,%fpaD]`	negates the sign of an operand
FPRDIV	`fprdiv.SF %fpaS,%fpaD`	reverse division (i.e. source ÷ destination)
FPRSUB	`fprsub.SF %fpaS,%fpaD`	reverse subtraction (i.e. source − destination)
FPSUB	`fpsub.SF %fpaS,%fpaD`	subtraction
FPTEST	`fptest.SF %fpaS`	compares the operand with zero
FPWAIT	`fpwait`	generates a loop to wait for the completion of a previously executed instruction

Notes

Assembler Listing Options 10

As supports two options for generating assembling listings. The `-A` option causes a listing to be printed to *stdout*. The `-a` *listfile* option writes a listing to *listfile*. In general, listing lines have the form:

<lineno> <offset> <codebytes> <source>

The <offset> is in hexadecimal, and offsets for *data* and *bss* locations are adjusted to be relative to the beginning of *text* in the *a.out* file. The <codebytes> are listed in hexadecimal. A maximum of 24 code bytes are displayed per source line (8 bytes per listing line, up to 3 listing lines per source line); excess bytes are not listed. Implicit alignment bytes are not listed. The <source> field is truncated to 40 characters.

The lister options cannot be used when the assembly source is *stdin*.

The following example shows a listing generating by assembling a small program using the `-A` option.

```
 1 0034      data
 2 0034      lalign 4
 3 0034      global _x
 4 0034     _x:
 5 0034 0000 0064    long 100
 6 0038      lalign 4
 7 0038      global _y
 8 0038     _y:
 9 0038 0000 0000    long 0
10 0000      text
11 0000      global _main
12 0000     _main:
13 0000 2F0E      mov.l %a6,-(%sp)
14 0002 2C4F      mov.l %sp,%a6
15 0004 DFFC FFFF FFF8    adda.l &LF1,%sp
16 000A 48D7 00C0    movm.l &LS1,(%sp)
17 000E 7C00      movq &0,%d6
18 0010 7E00      movq &0,%d7
19 0012     L16:
20 0012 BEB9 0000 0034    cmp.l %d7,_x
21 0018 6C00 000A    bge L15
22 001C DC87      add.l %d7,%d6
23 001E     L14:
24 001E 5287      addq.l &1,%d7
```

```
25 0020 6000 FFF0    bra L16
26 0024    L15:
27 0024 23C6 0000 0038    mov.l %d6,_y
28 002A    L13:
29 002A 4CD7 00C0    movm.l (%sp),&192
30 002E 4E5E     unlk %a6
31 0030 4E75     rts
32 0032      set LF1,-8
33 0032      set LS1,192
34 003C      data
```

Compatibility Issues A

When writing *as* assembly language code, you should be aware that each processor has a different register set. Because of this, it is possible to write assembly code that works on a Model 320 computer but doesn't work on a Model 310. Therefore, if your goal is to write portable code, keep the following in mind:

- Instructions that use the MC68020/30's additional registers will not work on either the MC68000 or MC68010.
- Likewise, instructions that use the MC68010's special registers will not work on the MC68000. However, such instructions *will* work on the MC68020/30 because the MC68010 register set is a subset of the MC68020/30 register set.
- The MC68010 instruction set is a subset of the MC68020/30 instruction set. Therefore, some MC68020/30 instructions will not work on the MC68010.
- Model 320, 330, 350, 360, and 370 computers use the MC68881 floating point co-processor. Therefore, if you have a Model 310 computer, you cannot write assembly language code to use the MC68881.

Using the -d Option

The `-d` option to *as* is used under special circumstances. It is typically used when you wish to write code that meets the following conditions:

- The code is intended to run on either a Model 310 or Model 320, 330, 350, 360, or 370 computers.
- There are actually two versions of the code: one for the MC68010 processor; the other for the MC68020/30 and MC68881 processors.
- The program makes a run-time decision on which code to execute.

For example, suppose you write some code to perform floating point operations. You want the code to run on either a Model 310 or Model 320 computer. When the code runs on a Model 310, all floating point operations must be performed in software; when the code runs on a Model 320, you want the code to use the MC68881 floating point co-processor so that it will run faster. The following pseudo-code illustrates this concept:

```
      :
if this code is running on a computer with MC68020/30 and MC68881 then
      :
   perform floating point operations using MC68881
      :
else      /* code is running on a Model 310 computer */
      :
   perform floating point operations using library routines
      :
endif
      :
```

If you write code that meets these conditions, then you should use the */bin/as20* assembler with the `-d` option. The `-d` option ensures that only MC68010-compatible address displacements will be generated. Therefore, the MC68010 code generated by *as20* will run on a Model 310.

Determining Processor at Run Time

The type of code discussed in the previous section is special in that it must determine which processor it is running on at run time. One way to make this run-time determination on current Series 300 computers is to look at the `flag_68010` flag in *crt0.o*. If this word is non-zero, then the processor is a MC68010; otherwise, it is a MC68020/30.

Another method would be to write a routine that sets up signal-catching for the signal `SIGILL`. (The `SIGILL` interrupt is generated if an illegal instruction is executed.) Then the routine would execute an MC68020/30-only instruction. If the illegal instruction interrupt occurs, then the code is not running on an MC68020/30 processor. (See *signal(2)* in the *HP-UX Reference* for details on setting up a signal handler.)

Two additional flag words are defined in *crt0.o* beginning with the 5.5 HP-UX release. These words are as follows:

flag_fpa is non-zero if there is a HP 98248 Floating-Point Accelerator in the system; otherwise, the word is zero (0).

flag_68881 is non-zero if there is an M68881 Floating-Point Coprocessor in the system; otherwise, the word is zero (0).

Notes

Diagnostics B

Whenever *as* detects a syntactic or semantic error, a single-line diagnostic message is written to standard error output (**stderr**). The message provides descriptive information along with the line number and filename in which the error occurred.

Most of the error messages generated by *as* are descriptive and self-explanatory. Two general messages require further comment:

- "**syntax error**": *as* generates this message when a line's syntax is illegal. If you encounter this error, check the overall format of the line and the format of each operand.
- "**syntax error (opcode/operand mismatch)**": The overall syntax of the line is legal, and the format of each operand is also legal; however, the combination of opcode, operation size, and operand types is not legal. Check the addressing modes for each operand and the operation sizes that are legal for the given opcode.

Notes

Interfacing Assembly Routines to Other Languages

C

This appendix describes information necessary to interface assembly language routines to procedures written in C, FORTRAN, or Pascal.

Linking

In order for a symbol defined in an assembly language source file (such as the name of an assembly language routine) to be known externally, it must be declared with the **global** pseudo-op. (The **comm** pseudo-op also marks identifiers as global.) (For details on these pseudo-ops, see the "Pseudo-Ops" chapter.)

It is not necessary for an externally defined symbol, used in an assembly program, to be declared in a global statement: if a symbol is used but not defined, it is assumed to be defined externally. However, to avoid possible name confusion with local symbols, it is recommended that you use the **global** pseudo-op to declare all external symbols.

Register Conventions

Several registers are reserved for run-time stack use and other purposes.

Frame and Stack Pointers

Register A6 is designated as a pointer to the current stack frame; its value remains constant during the execution of a routine; all local variables are addressed from it. Register A7 is designated the run-time stack pointer. Its value changes during the execution of the routine.

Scratch Registers

Registers D0, D1, A0, and A1 are "scratch registers" which are reserved to contain intermediate results or temporary values which do not survive through a call to a function. That is, a called routine is free to alter these registers without saving and restoring previous values, and a calling routine must save the value (in memory or a non-scratch register) before making a call if it wants the value preserved. All float registers, if they are present, are considered to be scratch registers by the C and F77 compilers; Pascal preserves their values across procedure and function calls.

Function Result Registers

All functions return their result in register D0 except when the result is a 64-bit real number in which case the result is returned in the D0-D1 register pair. Register A1 is used to pass to the called routine the address in the runtime stack of temporary storage where a C structure-valued function is to write its value. That address is passed back to the calling routine in D0 in the same way as any other address valued function.

Temporary Registers and Register Variables

Registers which are not reserved as described above (D2-D7, A2-A5) are available for two uses: First, they may be used as temporary value storage. Unlike the scratch registers, though, their integrity is guaranteed across function calls because their values are saved and restored. Second, they may be reserved by the user in C and by the F77 and Pascal compilers as "register variable" locations. If the FPA option is selected, A2 is reserved as the floating-point accelerator base register and only registers A3—A5 are available as address registers for scratch registers and register variables.

Calling Sequence Overview

This section describes the procedure calling conventions as they are *currently* implemented by the Series 300 C, FORTRAN, and Pascal compilers. These conventions must be followed in order to interface an assembly language routine to one of these higher level languages.

Calling Sequence Conventions

The following calling conventions are used whenever a routine is called:

- The calling routine pushes function arguments onto the runtime stack in reverse order. The called routine can always access a given parameter at a fixed offset from %a6 (the stack frame pointer).
- The calling routine pops the parameters from the stack upon return.
- The called routine must save any registers that it uses except the scratch registers D0, D1, A0, A1. The float registers can be treated as scratch registers, except when interfacing to Pascal.
- The called routine stores its return value in D0. A 64-bit real return value is stored in the register pair D0, D1.
- The called routine uses the `link` instruction in its prologue code to allocate local data space and to set up A6 and A7 for referencing local variables and parameters. (The `link` instruction modifies the values of A6 and A7. The extension of stack space is done by the HP-UX operating system when a %a7-relative reference would extend beyond the current stack space.)
- The called routine epilogue code uses the `unlk` and `rts` instructions to deallocate local data space and return to the calling procedure, respectively.

Example

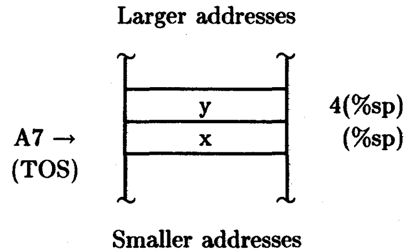
For example, consider the following simple C program.

```
int z;

main()
{
        int x,y;
        z = test(x,y);
}

test(i,j)
int i;
register int j;
{
        int k;
        k = i + j;
        return(k);
}
```

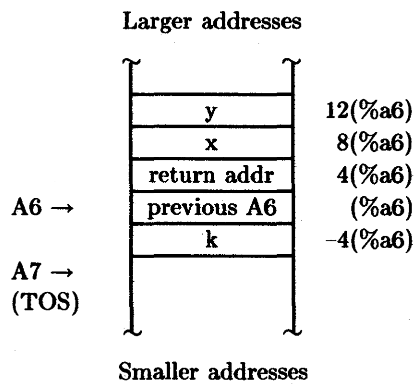
When compiled (but not optimized), it will generate assembly code like the following. (Comments have been added to point out features of the calling conventions.)

```
        comm    _z,4
        global  _main
_main:
        link.l  %a6,&LF1              # Allocate local data space
        movm.l  &LS1,(%sp)            # Save non-scratch registers
        mov.l   -8(%a6),-(%sp)        # Push argument "y"
        mov.l   -4(%a6),-(%sp)        # Push argument "x"
        jsr     _test                 # Call "test"
        addq    &8,%sp                # Pop arguments
        mov.l   %d0,_z                # Save function result
        movm.l  (%sp),&LS1            # Restore registers
        unlk    %a6                   # Deallocate local space
        rts                           # and return
        set     LF1,-8                # Gives size for local data
        set     LS1,0                 # Register mask of affected
                                      # non-scratch registers.

        global  _test
_test:
        link.l  %a6,&LF2              # Allocate local data space
        movm.l  &LS2,(%sp)            # Save non-scratch registers

        mov.l   12(%a6),%d7           # Parameter "j". Parameters
                                      # are at positive offsets off
                                      # %a6 (moved to %d7 because
                                      # of the "register" declaration.)
        mov.l   8(%a6),%d0
        add.l   %d7,%d0
        mov.l   %d0,-4(%a6)           # Local vars are at negative
                                      # offsets off %a6
        mov.l   -4(%a6),%d0           # Put return value in %d0
        bra.l   L15
L15:
        movm.l  (%sp),&LS2            # Restore registers
        unlk    %a6                   # Deallocate and return
        rts
        set     LF2,-8                # Displacement for link to
                                      # allocate local data space
        set     LS2,128
        data
```

Immediately before execution of the `jsr _test` instruction (line 8), the user stack looks like:



Following the `link` instruction in function `test`, the stack looks like:



C and FORTRAN

This section describes some of the language-specific dependencies of C and Fortran. You should consult the manual pages for these compilers for further information.

Assembly files can be generated by C and Fortran. You can examine the generated assembly files for additional information. (The only current means for looking at the code generated by the Pascal compiler is through the debugger adb.)

NOTE

All stack pictures in the remainder of this document depict the state of the stack immediately preceding execution of the `jsr sub_name` instruction. Larger addresses are always at the top; the stack grows from top to bottom.

C and FORTRAN Functions

In C and FORTRAN, all global-level variables and functions declared by the user are prefixed with an underscore. Thus, a variable name `xyz` in C would be known as `_xyz` at the assembly language level. All global variables can be accessed through this name using a long absolute mode of addressing.

C and FORTRAN push their arguments on the stack in right-to-left order. C always uses *call-by-value*, so actual argument *values* are placed on the stack. The current definition of C requires that argument values be extended to int's before pushing them on the stack; float's are extended to double's.

FORTRAN's parameter-passing mechanism is always *call-by-reference*, unless forced to call-by-value via the `$ALIAS` directive. In this document, all examples are call-by-reference. For each argument, the address of the most significant byte of the actual value is pushed on the stack.

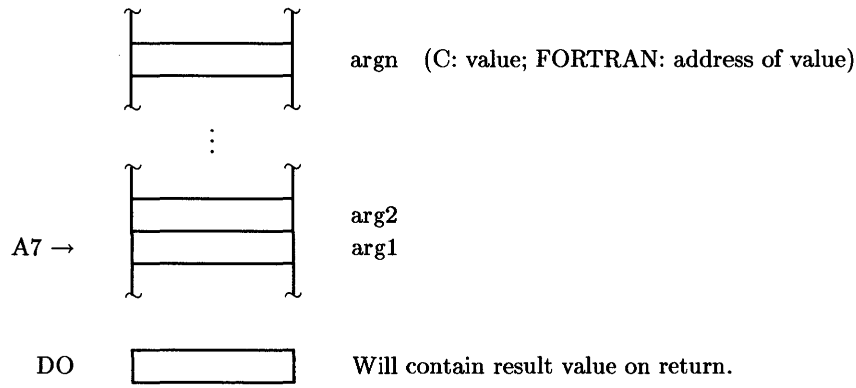
Function results are returned in register D0, or register pair D0, D1 for a 64-bit real result.

Note: For exceptions to FORTRAN's parameter-passing and return-value conventions, see the subsequent sections "FORTRAN CHARACTER Parameters," "FORTRAN CHARACTER Functions," and "FORTRAN COMPLEX Functions."

When a C structure-valued function is called, temporary storage for the return result is allocated on the runtime stack by the calling routine. The beginning address of this temporary storage space is passed to the called function through register A1.

The following shows the state of the stack after a routine with *n* arguments is called.

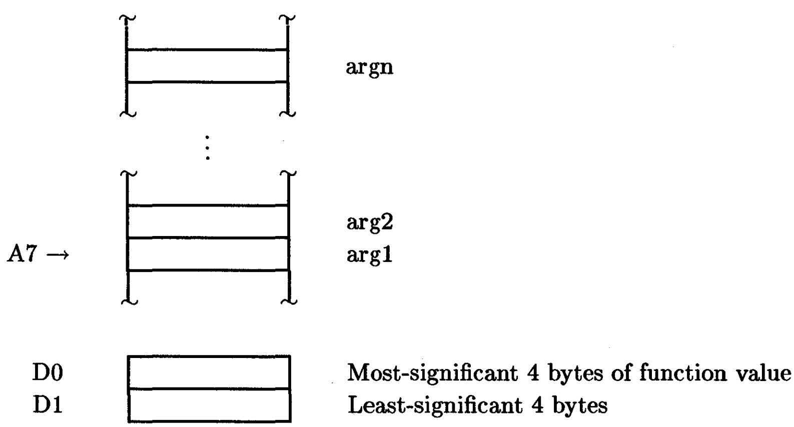
```
      C:  long func (arg1, arg2, ..., argn)
FORTRAN:  INTEGER FUNCTION func (arg1, arg2, ..., argn)
```



C and FORTRAN Functions Returning 64-Bit Double Values

For C and FORTRAN functions which return a 64-bit double value, the stack looks like:

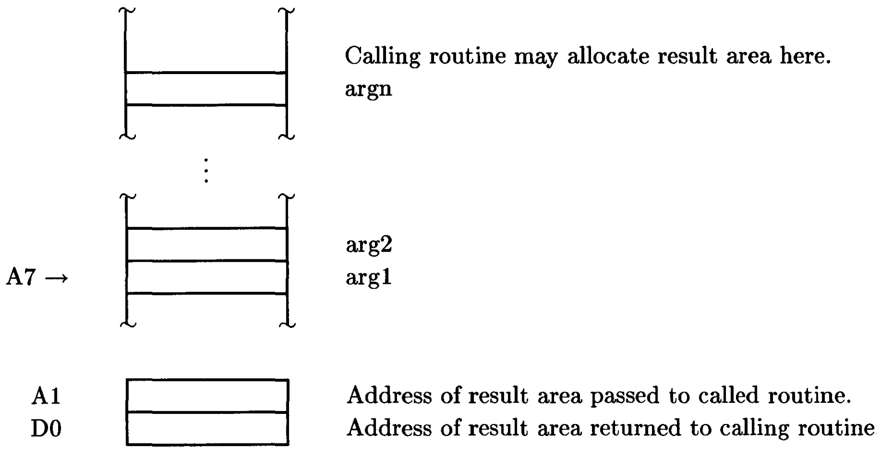
```
      C: double func (arg1, arg2, ..., argn)
FORTRAN: REAL*8 FUNCTION func (arg1, arg2, ..., argn)
```



C Structure-Valued Functions

The calling routine is responsible for allocating a result area of the proper size and alignment. It may be anywhere on the stack above the arguments, or it may be in static space. The address of the result area is passed to the called routine in register A1.

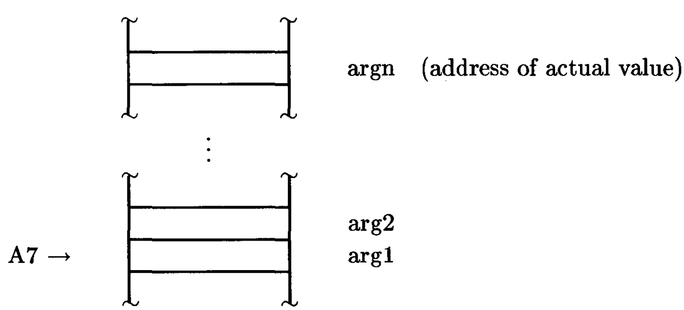
```
(struct s) func (arg1, arg2, ..., argn)
```



FORTRAN Subroutines

FORTRAN subroutines have the same calling sequences as FORTRAN functions described above, except that no results or result areas are dealt with.

```
SUBROUTINE sub (arg1, arg2, ..., argn)
```



FORTRAN CHARACTER Parameters

Each argument of type CHARACTER*n causes two items to be pushed on the stack. The first is a "hidden parameter" which gives the length of the CHARACTER argument. The second is the pointer to the argument value.

FORTRAN CHARACTER Functions

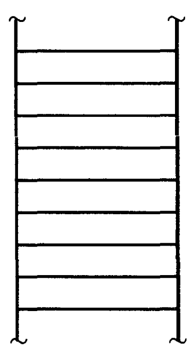
CHARACTER-valued FUNCTIONs are implemented differently from other FORTRAN functions. The calling routine is responsible for allocating the result area. However, the address of the result area is neither passed to nor returned from the called routine in registers. Instead, after all parameters are pushed on the stack, the length of the return value is pushed, followed by the address of the return area.

For example, suppose you call a character function as:

```
INTEGER int1, int3
CHARACTER*7 str1
CHARACTER*8 str2
CHARACTER*15 func, result
result = func (int1, str1, str2, int3)
```

Then the resulting stack is:

```
CHARACTER*15 FUNCTION func (arg1, arg2, arg3, arg4)
```

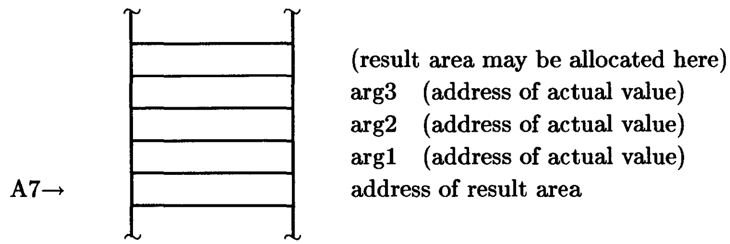


8	(size of str2)
7	(size of str1)
int3	(address of actual value)
str2	(address of actual value)
str1	(address of actual value)
int1	(address of actual value)
15	(size of result)
A7 →	address of result area

FORTRAN COMPLEX*8 and COMPLEX*16 Functions

All FORTRAN COMPLEX functions return their results through a result area.

```
COMPLEX*16 FUNCTION func (arg1, arg2, arg3)
```



Pascal

In Pascal, any exported user-defined function is prefixed by the module name surrounded by underscores. A function named `funk` in module `test` would be known as `_test_funk` to an assembly language programmer. If a procedure is declared to be external, as in

```
procedure proc; external;
```

then all calls to **proc** will be represented by `_proc` in assembly language.

Pascal uses both the call-by-value and call-by-reference mechanisms discussed for C and FORTRAN. Pascal also pushes its parameters on the stack in right-to-left order. All parameter information is stored in the parameter stack in multiples of four bytes (e.g., an argument of type *char* will occupy 4 bytes on the stack, not 1). No parameter or result area information is communicated to the called routine through registers. Pascal has a number of conventions not found in either C or FORTRAN. They are described below.

Static Links

All procedures and functions declared at level 2 or greater (main program is at level 0; contained procedures and functions are at level 1; routines inside these routines are at level 2, ...) expect a **static link** word on the stack below all parameter information. This word contains the address of the enclosing routine's stack frame (i.e., the value in register A6 when the routines immediately surrounding the called routine is executing). The called routine needs this information to access *intermediate* (i.e., non-local, non-global) variables on the stack.

Passing Large Value Parameters

Large *value* parameters are passed via a **copyvalue** mechanism. Calling routines pass copyvalue parameters by pushing the address of the value on the stack (i.e., treat them the same as call-by-reference parameters). Then the called routine makes a local copy of the parameter by dereferencing the pointer.

Parameter-Passing Rules

The rules used by the Pascal compiler for passing parameters are described here.

Call-By-Reference ("var" Parameters)

For all **var** parameters, push the address of the most significant byte.

Call-By-Value (Copyvalue Parameters)

If a value parameter meets either of the following criteria:

- it is a string
- it is larger than four bytes but is not a *longreal* or a *procedure/function* variable

then the address of the variable is pushed (as if by call-by-reference). Then the called routine uses the *copyvalue* mechanisim to make a local copy of the parameter.

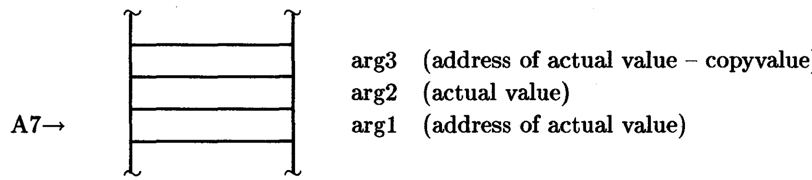
Call-By-Value (Non-Copyvalue Parameters)

For all *longreal*, *procedure/function* variables, and for all items that use four or less bytes (except strings), the value of the variable is pushed.

Example of Parameter Passing

The following Pascal procedure definition produces the stack below:

```
procedure proc (var arg1: real; arg2: integer; arg3: string[3]);
/* proc is declared at level 1 ==> no static link in calling sequence */
```



Pascal Functions Return Values

Like C and FORTRAN functions, Pascal functions return small results in registers D0 and D1. Larger function values are passed through a result area. The address of the result area is pushed before the argument values. The result area address is **not** communicated through any registers.

The following Pascal function types return values in D0 and possibly D1:

- scalar (includes char, boolean, enum, and integer)
- subrange
- real
- longreal
- pointer

The following Pascal function types return values through a result area:

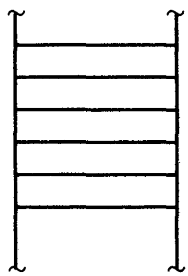
- procedure-valued
- set
- array
- string
- record
- file

Example with Static Link

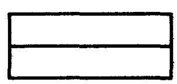
Suppose you've declared a Pascal procedure as:

```
function func (     arg1: longreal;
                var arg2: type1;
                    arg3: arraytype)   (* assume sizeof(arraytype) > 4 *)
               : longreal;
(* func is declared at level 2 ==> static link required *)
```

Then the arguments and static link would be placed on the stack as follows:



arg3 (address of actual value – copyvalue)
arg2 (address of actual value)
arg1 (actual value, 4 LSB's)
arg1 (actual value, 4 MSB's)
A7 → static link (stack frame address of level 1 routine containing "func")



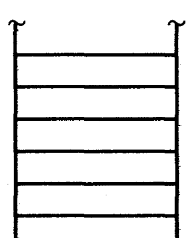
D0 4 MSB's of longreal result
D1 4 LSB's of longreal result

Example with Result Area

Suppose you've declared a Pascal function of a *set* type, which returns the result in a result area:

```
function func (     arg1: longreal;
                var arg2: type1;
                    arg3: arraytype)   (* assume sizeof(arraytype) > 4 *)
               : settype;
(* "func" is declared at level 1 ==> no static link expected *)
```

Then the resulting stack would be:



address of result area
arg3 (address of actual value – copyvalue)
arg2 (address of actual value)
arg1 (actual value, 4 LSB's)
A7 → arg1 (actual value, 4 MSB's)

Pascal Conformant Arrays

Several words of information are passed for conformant arrays. For every dimension, the length (including padding bytes), upper, and lower bounds are pushed. Last of all, the address of the array is placed on the stack.

Example Using Conformant Arrays

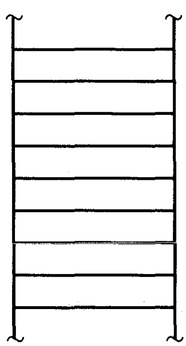
Consider the following Pascal code which calls a subroutine, **sub**, which performs operations on a conformant array.

```
var ary: array [1..3, 2..5] of integer;
    :
sub (ary);
```

The called routine is declared as:

```
procedure sub( ary[ lb1..ub1: integer; lb2..ub2: integer ] of integer );
(* sub declared at level 3 ==> static link required *)
```

The resulting stack will be:



16 – length of dimension 1
1 – lower bound of dim 1 (identifier "lb1")
3 – upper bound of dim 1 (identifier "ub1")
4 – length of dimension 2
3 – lower bound of dim 2 (identifier "lb2")
5 – upper bound of dim 2 (identifier "ub2")
address of "ary"
A7 → static link

Pascal “var string” Parameters.

var string parameters without a declared length have the maximum length passed as a *hidden* parameter. The subroutine must have this information to avoid writing past the end of string storage. The maximum size is pushed on the stack before the string address.

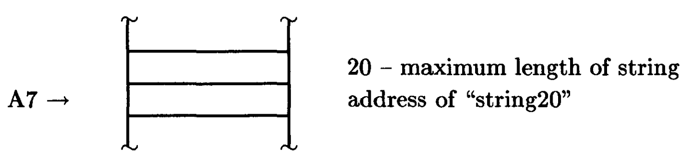
For example, suppose you've written the following Pascal code:

```
var string20: string[20];
    :
sub (string20);
```

The routine `sub` is declared as:

```
procedure sub (var s: string);
(* "sub" declared at level 1 ==> no static link expected *)
```

The resulting stack looks like:



Notes

Example Programs

D

This appendix provides sample assembly language programs. The intent of the programs is to show as many features of the *as* assembler as possible.

Interfacing to C

The following example illustrates a complete assembly example, and the interface of assembly and C code. The assemly source file *count1.s* contains an assembly language routine, `_count_chars`, which counts all the characters in an input string, incrementing counters in a global array (`count`). It checks for certain errors and uses the *fprintf(3C)* routine to issue error messages.

The example illustrates calling conventions between C and assembly code, including access to parameters, and the sharing of global variables between C and assembly routines. The variable `Stderr` is defined in *count1.s* but accessed in *prog.c*; the array `count` is defined in *prog.c* and accessed from *count1.s.*

The *cc(1)* command can be used to build a complete command from these sources:

```
cc -o ccount prog.c count1.s
```

The C Source File (prog.c)

```

/* Main driver for a program to count all occurences of each (7-bit)
 * ascii character in a sequence of input lines, and then dump the
 * results.
 * The loop to do the counting is done by a routine written in
 * assembly.
 */

# include <stdio.h>
# define SMAX 100       /* maximum string size */
char input_string[SMAX];

# define NCHAR 128
unsigned short count[NCHAR];

extern int count_chars();        /* Routine to do the count. It returns
                                  * a count of the total number of
                                  * characters it counted.
                                  */

unsigned int totalcount;         /* Total letter count */
extern FILE * Stderr;

main() {
   Stderr = stderr;     /* Set up error descriptor required by
                         * count_chars.
                         */
   while (fgets(input_string, SMAX, stdin) != NULL )
        totalcount += count_chars(input_string);

   dump_counts();
}

dump_counts() {
  register int i;

  printf("Char Value    Count\n");
  printf("=========     =====\n");
  for (i =0; i<NCHAR; i++)
        printf("\t%02X\t%4u\n", i, count[i]);

  printf("\nTotal Letters Counted = %d\n", totalcount);
}
```

The Assembly Source File (count1.s)

```
# count_chars (s)
# Routine to count characters in input string
# Called as
#       count_chars(s)
#  from C.
#       Count the occurrences of each (7-bit) ascii character in the
#       input line pointed to by "s".
#       The input lines are guaranteed to be null-terminated.
#       The counts are stored in external array
#               unsigned short count[NCHAR]
#       where NCHAR in 128.
#       Give an error (using fprintf from libc) if
#               * an input char in not in the 7-bit ascii range.
#               * the count overflows for a given character.
#       The return value is the total number of characters counted.
#       Illegal characters are not included in the total character
#       count.
# Calling routine must set global variable Stderr to file descriptor
# for error messages. We make this require because a C program
# can more portably calculate the necessary address.

        global  _count         # Array of unsigned short for storing cnts
                               # is defined externally
        global  _fprintf       # External function
        global  _count_chars   # Make _count_characters visible
                               # externally

# Register usage:
#  NOTE: We don't use scratch registers for variables we would want
#  preserved across calls to _printf. An alternative strategy would be
#  to use all scratch registers and save them around any calls to
#  _printf, on the assumption that such calls will be rare.
#       %a2 :  address of count[] array
#       %a3 :  step through input string
#       %d2 :  total character count
#       %d1 :  value of current character  (scratch register)
```

```

        global  _Stderr         # Stderr file descriptor - must be
                                # externally set.
        bss
_Stderr:        space 4

        text
_count_chars:
        link.l  %a6,&-12        # No local vars.  3 registers to save
        movm.l  %a2-%a3/%d2,(%sp)
        mov.l   &_count,%a2     # Count array
        mov.l   8(%a6),%a3      # Input string pointer
        clr.l   %d2             # Total character count
Loop:
        mov.b   (%a3)+,%d1      # Next character
        beq.b   Ldone           # Null character terminates string
        bmi.b   Lneg            # Illegal character
        addq.l  &1,%d2          # Increment total count
        ext.w   %d1             # Make %d1 usuable as an index
        addq.w  &1,(%a2,%d1.w*2)    # Increment the appropriate counter
        bcs.b   Lovflw
        bra.b   Loop            # Go back for next character

Lneg:   # illegal character seen -- give an error
        # push args for fprintf, in reverse order
        and.l   &0xff,%d1       # Only want low 2 bytes in arg passed.
        mov.l   %d1,-(%sp)
        mov.l   &Err1,-(%sp)
        mov.l   _Stderr,-(%sp)
        jsr     _fprintf
        add.l   &12,%sp         # Pop the 3 arguments
        bra.b   Loop            # Go back for next character

Lovflw: # count overflowed -- give an error
        # push args for fprintf, in reverse order
        and.l   &0xff,%d1       # Only want low 2 bytes in arg passed.
        mov.l   %d1,-(%sp)
        mov.l   &Err2,-(%sp)
        mov.l   _Stderr,-(%sp)
        jsr     _fprintf
        add.l   &12,%sp         # Pop the 3 arguments
        bra.b   Loop            # Go back for next character

```

```
Ldone:
        mov.l   %d2,%d0           # return value
        movm.l  (%sp),%a2-%a3/%d2 # restore registers
        unlk    %a6
        rts


        data
Err1:   asciz   "Illegal character (%02X) in input\n"
Err2:   asciz   "Count overflowed for character (%02X)\n"
```

Using MC68881 Instructions

The following assembly language program uses MC68881 instructions to approximate a *fresnel* integral.

```
#
# double fresnel(z) double z;
#
# Approximate fresnel integral by calculating first hundred terms of
# series expansion. For n=0 to n=99, each term is:
#
#             (-1)^n * (PI/2)^(2*n) * z^(4*n+1)
#            -----------------------------------
#                      (2*n)! * (4*n+1)

        set     PI,0
        text
        global  _fresnel
_fresnel:
        link    %a6,&-8
        mov.l   %d2,-4(%a6)           # save d2
        fmov    %fpcr,-8(%a6)         # save control register
        fmov    &0,%fpcr              # disable traps; round to
                                      # nearest extended format
        movq    &0,%d0                # n
        movq    &1,%d1                # 4*n+1
        fmov.w  &0,%fp0               # initialize sum
        fmov.b  &1,%fp1               # (pi/2)^(2*n)
        fmov.d  8(%a6),%fp3           # z
        fmov    %fp3,%fp2             # initialize z^(4*n+1)
        fmul    %fp3,%fp3             # z^2
        fmul    %fp3,%fp3             # z^4
        fmov.b  &1,%fp4               # initialize (2*n)!
```

```
          fmovcr  &PI,%fp5          # pi
          fdiv.b  &2,%fp5           # pi/2
          fmul    %fp5,%fp5         # (pi/2)^2
loop:
          fmov    %fp1,%fp6         # (pi/2)^(2*n)
          fdiv    %fp4,%fp6         # divide by (2*n)!
          fdiv.l  %d1,%fp6          # divide by 4*n+1
          fmul    %fp2,%fp6         # multiply by z^(4n+1)
          movq    &1,%d2
          and.b   %d0,%d2           # odd or even term?
          bne.b   L1
          fadd    %fp6,%fp0         # add term
          bra.b   L2
L1:       fsub    %fp6,%fp0         # subtract term
L2:       addq.l  &1,%d0            # n=n+1
          cmp.l   %d0,&100          # end of loop?
          beq.b   L3
          mov.l   %d0,%d2           # new n
          asl.l   &1,%d2            # n*2
          fmul.l  %d2,%fp4          # update (2*n)!
          subq.l  &1,%d2
          fmul.l  %d2,%fp4
          addq.l  &4,%d1            # update 4*n+1
          fmul    %fp3,%fp2         # update z^(4*n+1)
          fmul    %fp5,%fp1         # update (pi/2)^(2*n)
          bra.b   loop
L3:       fmov.d  %fp0,-(%sp)       # get result
          movm.l  (%sp)+,%d0-%d1
          mov.l   -4(%a6),%d2       # restore d2
          fmov    -8(%a6),%fpcr     # restore control register
          unlk    %a6
          rts
```

Notes

Translators

E

Two assembly source translators are provided to assist in converting assembly code from other HP systems to *as* assembly language for Series 300 computers.

atrans(1)

The *atrans* translator converts Pascal Language System (PLS) assembly language to *as* assembly language format. You should consult the *atrans(1)* page of the *HP-UX Reference* for details on using the *atrans* command.

astrn(1)

The *as* assembler uses a UNIX-like assembly syntax which differs in several ways from the syntax of previous HP-UX assemblers. The *astrn* translator translates old HP-UX Series 200/300 assembly language to the new *as* assembly language for Series 200/300 HP-UX systems. Consult the *astrn(1)* page of the *HP-UX Reference* for details on the *astrn* command.

NOTE

The translators are able to perform most of the translation to *as* assembly language format. However, some translation is beyond the capabilities of the translators. Lines that require human intervention to change will generate warning messages. See the appropriate page—*atrans(1)* or *astrn(1)*—of the *HP-UX Reference* for details on warning messages.

Notes

Unsupported Instructions for Series 300's

F

HP-UX Series 300 assemblers support the complete MC68010 and MC68020/30 instruction sets. However, some instructions are not fully supported by the HP-UX hardware. These instructions are as follows:

- *tas*
- *cas*
- *cas2*
- *bkpt*

The assembler generates code for these instructions, but gives warning messages that the instructions are not fully supported by the HP-UX hardware.

Notes Regarding Unsupported Instructions

This section provides detailed notes regarding the previously mentioned unsupported assembler instructions for Series 300 computers. Topics covered are as follows:

- Instructions Not Supported by the Model 310
- Instructions Not Supported by the Model 320
- Instructions Not Supported by the Model 330
- Instructions Not Supported by the Model 350, 360 or 370

Instructions Not Supported by the Model 310

The *tas* instruction is not supported by the Model 310. Executing a *tas* instruction will either generate a bus error or corrupt memory.

The instructions *cas* and *cas2* are illegal instructions. These instruction will cause normal exception processing for an illegal instruction.

The *bkpt* instruction is not illegal, but it will end up in illegal instruction processing.

Instructions Not Supported by the Model 320

The instructions *tas*, *cas*, and *cas2* will execute; however, they may cause cache consistency problems. These instructions completely bypass the cache, so if you reference the same memory locations with a different instruction you will get the old data stored in the cache instead of the new data written to memory.

The *bkpt* instruction will cause illegal instruction exception processing.

Instructions Not Supported by the Model 330

The instructions *tas*, *cas*, and *cas2* execute properly because there is no cache to be inconsistent.

The *bkpt* instruction causes illegal instruction exception processing.

Instructions Not Supported by the Model 350, 360 or 370

The instructions *tas*, *cas*, and *cas2* execute properly. The cache consistency is maintained.

The instruction *bkpt* will cause illegal instruction exception processing.

Index

a

b

c

d

e

f

g

h

i

l

m

n

o

p

r

S

t

U

V

W

Z

Table of Contents

ADB Tutorial

ADB is a debugging program that is available on HP-UX. It provides capabilities to look at "core" files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB.

Invocation

To use ADB, you must execute (invoke) the *adb(1)* command; its syntax is:

adb [-w] [*objfile* [*corefile*]]

where *objfile* is an executable HP-UX file and *corefile* is a core image file. Often times, *adb* is invoked as:

```
adb a.out core
```

or more simply:

```
adb
```

where the defaults are *a.out* and *core* respectively. The filename minus (-) means "ignore this argument," as in:

```
adb - core
```

The *objfile* can be written to if *adb* is invoked with the `-w` flag as in:

```
adb -w a.out -
```

ADB catches signals; therefore, a user cannot use a quit signal to exit from ADB. The request **$q** or **$Q** (or CTRL-D) must be used to exit from ADB.

For details on invoking the *adb* command, see the *adb(1)* page in the *HP-UX Reference*.

ADB Command Format

You interact with ADB by entering (typing) requests. The general format of a request is:

[*address*] [,*count*] [*command*] [*modifier*]

ADB maintains a current address, called *dot*, similar in function to the current pointer in the HP-UX editor, *vi(1)*. When *address* is entered, dot is set to that location. The *command* is then executed *count* times.

Address and *count* are represented by expressions. You can create expressions from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the following operators:

+	addition
–	subtraction or negation (when used as a unary operator)
*	multiplication
%	integer division
&	bitwise AND
|	bitwise inclusive OR
#	round up to the next multiple
~	unary not.

All arithmetic within ADB is 32 bits.

When typing a symbolic address for a C program, the user can type *name* or *_name*; ADB will recognize both forms. The default base for integer input is initialized to hexadecimal, but can be changed.

Table 1 illustrates some commonly used ADB commands and their meanings:

Table 1. Commonly Used ADB Commands

Command	Description
?	Print contents from `a.out` file
/	Print contents from `core` file
=	Print value of "dot"
:	Breakpoint control
$	Miscellaneous requests
;	Request separator
!	Escape to shell

CTRL-C terminates execution of any ADB command.

Displaying Information

ADB has requests for examining locations in either the *objfile* or the *corefile.* The `?` request examines the contents of *objfile*, the `/` request examines the *corefile.*

Following the `?` or `/` command the user specifies a format.

The following are some commonly used format letters:

`c`	one byte as a character
`x`	two bytes in hexadecimal
`X`	four bytes in hexadecimal
`d`	two bytes in decimal
`F`	eight bytes in double floating point
`i`	MC68xxx instruction
`s`	a null-terminated character string
`a`	print in symbolic form
`n`	print a newline
`r`	print a blank space
`^`	backup dot.

A command to print the first hexadecimal element of an array of long integers named `ints` in C would look like:

```
ints/X
```

This instruction would set the value of **dot** to the symbol table value of `_ints`. It would also set the value of the dot increment to four. The dot increment is the number of bytes printed by the format.

Let us say that we wanted to print the first four bytes as a hexadecimal number and the next four as a decimal one. We could do this by:

```
ints/XD
```

In this case, **dot** would still be set to `_ints` and the dot increment would be eight bytes. The dot increment is the value which is used by the `newline` command. `Newline` is a special command which repeats the previous command. It does not always have meaning. In this context, it means to repeat the previous command using a count of one and an address of dot plus dot increment. In this case, `newline` would set dot to `ints+0x8` and type the two long integers it found there, the first in hex and the second in decimal. The `newline` command can be repeated as often as desired and this can be used to scroll through sections of memory.

Using the above example to illustrate another point, let us say that we wanted to print the first four bytes in long hex format and the next four bytes in byte hex format. We could do this by:

```
ints/X4b
```

Any format character can be preceded by a decimal repeat character.

The count field can be used to repeat the entire format as many times as desired. In order to print three lines using the above format we would type

```
ints,3/X4bn
```

The `n` on the end of the format is used to output a carriage return and make the output much easier to read.

In this case the value of dot will not be `_ints`. It will rather be `_ints+0x10`. Each time the format was re-executed dot would have been set to dot plus dot increment. Thus the value of dot would be the value that dot had at the beginning of the last execution of the format. Dot increment would be the size of the format: eight bytes. A `newline` command at this time would set dot to `ints+0x18` and print only one repetition of the format, since the count would have been reset to one.

In order to see what the value of dot is at this point the command

```
.=a
```

could be typed. `=` is a command which can be used to print the value of **address** in any format. It is also possible to use this command to convert from one base to another:

```
0x32=oxd
```

This will print the value `0x32` in octal, hexadecimal and decimal.

Complicated formats are remembered by ADB. One format is remembered for each of the `?` , `/` and `=` commands. This means that it is possible to type

```
0x64=
```

and have the value `0x64` printed out in octal, hex and decimal. And after that, type

```
ints/
```

and have ADB print out four bytes in long hex format and four bytes in byte hex format.

To an observant individual it might seem that the two commands

```
main,10?i
```

and

```
main?10i
```

would be the same.

There are two differences. The first is that the numbers are in a different base. The repeat factor can only be a decimal constant, while the count can be an expression and is therefore, by default, in a hex base.

The second difference is that a `newline` after the first command would print one line, while a `newline` after the second command would print another ten lines.

Debugging C Programs

The following examples illustrate various features of ADB. Certain parts of the output (such as machine addresses) may depend on the hardware being used, as well as how the program was linked (unshared, shared, or demand loaded).

Debugging a Core Image

Consider the C program in Figure 1. The program is used to illustrate some of the useful information that can be obtained from a core file. The object of the program is to calculate the square of the variable `ival` by calling the function `sqr` with the address of the integer. The error is that the value of the integer is being passed rather than the address of the integer. Executing the program produces a core file because of a bus error.

Figure 1. C Program with a Pointer Bug

```
int ints[]=     {1,2,3,4,5,6,7,8,9,0,
                 1,2,3,4,5,6,7,8,9,0,
                 1,2,3,4,5,6,7,8,9,0,
                 1,2,3,4,5,6,7,8,9,0};
int ival;
main()
{
        register int i;
        for(i=0;i<10;i++)
        {       ival = ints[i];
                sqr(ival);
                printf("sqr of %d is %d\n",ints[i],ival);
        }
}

sqr(x)
int *x;
{
        *x *= *x;
}
```

ADB is invoked by:

```
adb
```

The first debugging request:

```
$c
```

is used to give a C backtrace through the subroutines called. This request can be used to check the validity of the parameters passed. As shown in Figure 2 we can see that the value passed on the stack to the routine `sqr` is a 1, which is not what we are expecting.

Figure 2. ADB Output from Program of Figure 1

```
$c
_main+0x30:      _sqr    (0x1)
start+0x58:      _main   (0x1, 0xFFFF7DAC)
$r
ps       0x0
pc       0x11C   _sqr+0x42:       unlk     %a6

sp   0xFFFF7D84

d0   0x1AE9              a0   0x1
d1   0x53                a1   0xFFFF7DAC
d2   0xFFC01             a2   0xFFC8A004
d3   0xFFC8F405          a3   0x1F626
d4   0xFFC8F401          a4   0x1F66C
d5   0x700               a5   0x1F3AC
d6   0x0                 a6   0xFFFF7D88
sqr+0x38,5?ia
_sqr+0x38:                 mov.w    (%a7)+,%d0
_sqr+0x3A:                 mulu.w   %d1,%d0
_sqr+0x3C:                 mov.l    0x8(%a6),%a0
_sqr+0x40:                 mov.l    %d0,(%a0)
_sqr+0x42:                 unlk     %a6
_sqr+0x44:
$e
flag_68881:      0x10000
_environ:        0xFFFF7DB4
_argc_value:     0x1
float_soft:      0xFFFF0001
_argv_value:     0xFFFF7DAC
_ints:   0x1
_ival:   0x1
__iob:   0x0
__ctype:         0x202020
__bufendtab:     0x0
__smbuf:         0x0
__lastbuf:       0x39D4
_errno: 0x0
__stdbuf:        0x40DC
__sobuf:         0x0
__sibuf:         0x0
_asm_mhfl:       0x0
_end:    0x0
_errnet:         0x0
_edata: 0x1
```

The next request:

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location. The instruction printed for the pc does not always make sense. This is because the pc has been advanced and is either pointing at the next instruction, or is left at a point part way through the instruction that failed. In this case the pc points to the next instruction. In order to find the instruction that failed we could list the instructions and their offsets by the following command.

```
sqr+0x38,5?ia
```

This would show us that the instruction that failed was

```
_sqr+0x40:move.l %d0, (%a0)
```

This is the first instruction before the value of the pc. The value printed out for register a0 also indicates that a write to location 0x1, which is in the text part of the user space, would fail in a shared *a.out* file. The text segment is write-protected in files that are shared or demand-loaded.

The request:

```
$e
```

prints out the values of all external variables at the time the program crashed.

Setting Breakpoints

Consider the C program in Figure 3. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

Figure 3. C Program to Decode Tabs

```
#include <stdio.h>
#define MAXLINE 80
#define YES            1
#define NO             0
#define TABSP          8

char    input[] = "data";
FILE    *stream;
int     tabs[MAXLINE];
char    ibuf[BUFSIZ];

main()
{
        int col, *ptab;
        char c;

        setbuf(stdout,ibuf);
        ptab = tabs;
        settab(ptab);   /*Set initial tab stops */
        col = 1;
        if((stream = fopen(input,"r")) == NULL) {
                printf("%s : not found\\n",input);
                exit(8);
        }
        while((c = getc(stream)) != EOF) {
                switch(c) {
                        case '\t':          /* TAB */
                                while(tabpos(col) != YES) {
                                        putchar(' ');   /* put BLANK */
                                        col++ ;
                                }
                                break;
                        case '\n':          /*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
                        default:
                                putchar(c);
                                col++ ;
                }
        }
}
```

```
/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

We will run this program under the control of ADB (see Figure 4) by:

```
adb a.out -
```

Breakpoints are set in the program as:

```
address:b [request]
```

The requests:

```
settab+e:b
fopen+4:b
tabpos+e:b
```

set breakpoints at the starts of these functions. The addresses for user-defined functions (**settab** and **tabpos**) are entered as **symbol+e** so that they will appear in any C backtrace; this is because the first few instructions of each function are instructions which link in the new function. Note that one of the functions, **fopen**, is from the C library; for this routine, **fopen+4** is appropriately used.

Figure 4. ADB Output from C Program of Figure 3

```
adb a.out -
executable file = a.out
ready
settab+e:b
fopen+4:b
tabpos+e:b
$b
breakpoints
count bkpt  command
0x1     _tabpos+0xE
0x1     _fopen+0x4
0x1     _settab+0xE
:r
process 5139 created
a.out: running
breakpoint      _settab+0xE:    clr.l   -0x4(%a6)
settab+e:d
:c
a.out: running
breakpoint      _fopen+0x4:     jsr     __findiop
$c
_main+0x48:     _fopen  (0x4000, 0x4006)
start+0x58:     _main   (0x1, 0xFFFF7DAC)
tabs/24X
_tabs:          0x1             0x0             0x0             0x0
                0x0             0x0             0x0             0x0
                0x1             0x0             0x0             0x0
                0x0             0x0             0x0             0x0
                0x1             0x0             0x0             0x0
                0x0             0x0             0x0             0x0
:c
a.out: running
breakpoint      _tabpos+0xE:    movq    &0x50,%d0
:s
a.out: running
stopped at      _tabpos+0x10:   cmp.l   %d0,0x8(%a6)
 <newline>
a.out: running
stopped at      _tabpos+0x14:   bge.w   _tabpos+0x1E
 <newline>
```

```
a.out: running
stopped at      _tabpos+0x1E:   mov.l   0x8(%a6),%d0
 <newline>
a.out: running
stopped at      _tabpos+0x22:   asl.l   &0x2,%d0
 <newline>
a.out: running
stopped at      _tabpos+0x24:   addi.l  &0x4A50,%d0
 <newline>
a.out: running
stopped at      _tabpos+0x2A:   mov.l   %d0,%a0
 <newline>
a.out: running
stopped at      _tabpos+0x2C:   mov.l   (%a0),%d0
:d*
:c
a.out: running
This   is it
process terminated
settab+e:b settab,5?ia
tabpos+e,3:b ibuf/20c
:r
process 5248 created
a.out: running
settab,5?ia
_settab:                mov.l   %a6,-(%a7)
_settab+0x2:            mov.l   %a7,%a6
_settab+0x4:            add.l   &0xFFFFFFFC,%a7
_settab+0xA:            movm.l  &<>,(%a7)
_settab+0xE:            clr.l   -0x4(%a6)
_settab+0x12:
breakpoint      _settab+0xE:    clr.l   -0x4(%a6)
:c
a.out: running
ibuf/20c
_ibuf:          This
ibuf/20c
_ibuf:          This
ibuf/20c
_ibuf:          This
breakpoint      _tabpos+0xE:    movq    &0x50,%d0
$q
process 5248 killed
```

To print the location of breakpoints type:

```
$b
```

The display indicates a *count* field. A breakpoint is bypassed *count-1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function **settab** we see that the breakpoint is set after the instruction to save the registers on the stack. We can display the instructions using the ADB request:

```
settab,5?ia
```

This request displays five instructions starting at **settab** with the addresses of each location displayed.

To run the program simply type:

```
:r
```

To delete a breakpoint, for instance the entry to the function **settab**, type:

```
settab+4:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for **fopen**), ADB requests can be used to display the contents of memory. For example:

```
$c
```

to display a stack trace, or:

```
tabs,3/8X
```

to print three lines of 8 locations each from the array called **tabs**. The format **X** is used since integers are four bytes on M680x0 processors. By this time (at location **fopen**) in the C program, **settab** has been called and should have set a one in every eighth location of **tabs**.

Advanced Breakpoint Usage

When we continue the program with:

```
:c
```

we hit our first breakpoint at **tabpos** since there is a tab following the "This" word of the data. We can execute one instruction by

```
:s
```

and can single step again by pressing the Return key. Doing this we can quickly single step through **tabpos** and get some confidence that it is working. We can look at twenty characters of the buffer of characters by typing:

```
>ibuf/20c
```

Several breakpoints of **tabpos** will occur until the program has changed the tab into equivalent blanks. Since we feel that **tabpos** is working, we can remove all the breakpoints by:

```
:d*
```

If the program is continued with:

```
:c
```

it resumes normal execution and continues to completion after ADB prints the message:

```
a.out: running
```

It is possible to add a list of commands we wish to execute as part of a breakpoint. By way of example let us reset the breakpoint at **settab** and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
settab+e:b  settab,5?ia
```

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only stop after the third occurrence by typing:

```
tabpos+e,3:b ibuf/20c
```

This request will print twenty character from the buffer of characters at each occurrence of the breakpoint.

If we wished to print the buffer every time we passed the breakpoint without actually stopping there we could type

```
tabpos+e,-1:b ibuf/20c
```

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+e:b  settab,5?ia;ptab/o
```

could be entered after typing the above requests. The semicolon is used to separate multiple ADB requests on a single line.

Now the display of breakpoints:

```
$b
```

shows the above request for the **settab** breakpoint. When the breakpoint at **settab** is encountered the ADB requests are executed.

Note

Setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

```
settab+e:b .,5?ia
fopen+4:b
```

will print the last thing dot was set to (in the example **fopen**) not the current location (**settab**) at which the program is executing.

The HP-UX *quit* and *interrupt* signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

```
:c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

```
:c 0
```

is typed.

Other Breakpoint Facilities

Arguments and change of standard input and output are passed to a program as:

```
:r arg1 arg2 ... <infile> outfile
```

This request kills any existing program under test and starts the `a.out` afresh. The process will run until a breakpoint is reached or until the program completes or crashes.

If it is desired to start the program without running it the command

```
:e arg1 arg2 ... <infile> outfile
```

can be executed. This will start the process, and leave it stopped without executing the first instruction.

If the program is stopped at a subroutine call it is possible to step around the subroutine by

```
:S
```

This sets a temporary breakpoint at the next instruction and continues. This may cause unexpected results if `:S` is executed at a branch instruction.

ADB allows a program to be entered at a specific address by typing:

```
address:r
```

The count field can be used to skip the first *n* breakpoints as:

```
,n:r
```

The request:

```
,n:c
```

may also be used for skipping the first *n* breakpoints when continuing a program.

A program can be continued at an address different from the breakpoint by:

```
address:c
```

The program being debugged runs as a separate process and can be killed by:

```
:k
```

All of the breakpoints set so far can be deleted by

```
:d*
```

A subroutine may be called by

```
:x address [parameters]
```

Maps

HP-UX supports several executable file formats. These are used to tell the loader how to load the program file. A shared text program file is the most common and is generated by a C compiler invocation such as `cc pgm.c`. A non-shared text file is produced by a C compiler command of the form `cc -N pgm.c`, while a demand-loaded a.out file is produced by a C compiler command of the form `cc -q pgm.c`. ADB interprets these different file formats and provides access to the different segments through the maps. To print the maps type:

```
$m
```

In nonshared files, both text (instructions) and data are intermixed. In shared files the instructions are separated from data and `?*` accesses the data part of the `a.out` file. The `?*` request tells ADB to use the second part of the map in the `a.out` file. Accessing data in the `core` file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution. Figure 5 shows the display of three maps for the same program linked as nonshared, shared, and demand-loaded, respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The `f1` field is the length of the header at the beginning of the file. The `f2` field is the displacement from the beginning of the file to the data. For a nonshared file with mixed text and data this is the same as the length of the header; for shared files this is the length of the header plus the size of the text portion.

Figure 5: ADB output for maps

```
$ adb a.out.unshared core.unshared
executable file = a.out.unshared
core file = core.unshared
ready
$m
? map   'a.out.unshared'
b1 = 0x0          e1 = 0x19C        f1 = 0x40
b2 = 0x0          e2 = 0x19C        f2 = 0x40
/ map   'core.unshared'
b1 = 0x0          e1 = 0x1000       f1 = 0x3000
b2 = 0xFFFF5000 e2 = 0xFFFF8000 f2 = 0x4000
$v
variables
d = 0x1000
m = 0x107
s = 0x3000
$q

$ adb a.out.shared core.shared
```

```
executable file = a.out.shared
core file = core.shared
ready
$m
? map   'a.out.shared'
b1 = 0x0        e1 = 0x18C      f1 = 0x40
b2 = 0x1000     e2 = 0x1010     f2 = 0x1CC
/ map   'core.shared'
b1 = 0x1000     e1 = 0x2000     f1 = 0x3000
b2 = 0xFFFF5000 e2 = 0xFFFF8000 f2 = 0x4000
 $v
variables
b = 0x1000
d = 0x1000
m = 0x108
s = 0x3000
t = 0x1000
 $q

$ adb a.out.demand core.demand
executable file = a.out.demand
core file = core.demand
ready
$m
? map   'a.out.demand'
b1 = 0x0        e1 = 0x18C      f1 = 0x1000
b2 = 0x1000     e2 = 0x1010     f2 = 0x2000
/ map   'core.demand'
b1 = 0x1000     e1 = 0x2000     f1 = 0x3000
b2 = 0xFFFF5000 e2 = 0xFFFF8000 f2 = 0x4000
 $v
variables
b = 0x1000
d = 0x1000
m = 0x10B
s = 0x3000
t = 0x1000
 $q
```

The `b` and `e` fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either `a.out` or `core`) is calculated as:

$$b1 \le A \le e1 \Rightarrow \text{file address} = (A\text{-}b1)+f1$$
$$b2 \le A \le e2 \Rightarrow \text{file address} = (A\text{-}b2)+f2$$

Variables and Registers

ADB provides a set of variables which are available to the user. A variable is composed of a single letter or digit. It can be set by a command such as

```
0x32>5
```

which sets the variable 5 to hex 32. It can be used by a command such as

```
<5=X
```

which will print the value of the variable 5 in hex format.

Some of these variables are set by ADB itself. These variables are:

0	last value printed
b	base address of data segment
d	length of the data segment
e	The entry point
m	execution type (0x107 (nonshared),0x108 (shared), or 0x10b (demand loaded))
s	length of the stack
t	length of the text

These variables are useful to know if the file under examination is an executable or `core` image file. ADB reads the header of the core image file to find the values for these variables. If the second file specified does not seem to be a core file, or if it is missing, the header of the executable file is used instead.

Variables can be used for such purposes as counting the number of times a routine is called. Using the example of Figure 3, if we wished to count the number of times the routine `tabpos` is called we could do that by typing the sequence

```
0>5
tabpos+4,-1:b <5+1>5
:r
<5=d
```

The first command sets the variable 5 to zero. The second command sets a breakpoint at `tabpos+4`. Since the count is -1 the process will never stop there but ADB will execute the breakpoint command every time the breakpoint is reached. This command will increment the value of the variable 5 by 1. The `:r` command will cause the process to run to termination, and the final command will print the value of the variable.

`$v` can be used to print the values of all non-zero variables.

The values of individual registers can be set and used in the same way as variables. The command

```
0x32>d0
```

will set the value of the register `d0` to hex 32. The command

```
<d0=X
```

will print the value of the register `d0` in hex format. The command `$r` will print the value of all the registers.

Formatted Dumps

It is possible to combine ADB formatting requests to provide elaborate displays. Below are some examples.

The line:

```
<b,-1/4o4^8Cn
```

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

<b	The base address of the data segment.
<b,-1	Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format 4o4^8Cn is broken down as follows:

4o	Print 4 octal locations.
4^	Backup the current address 4 locations (to the original start of the field).
8C	Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.
n	Print a newline.

The request:

```
<b,<d/4o4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, `dump`, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8ona
```

The request `120$w` sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

```
symbol + offset
```

The request `4095$s` increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request = can be used to print literal strings. Thus, headings are provided in this `dump` program with requests of the form:

```
=3n"C Stack Backtrace"
```

that spaces three lines and prints the literal string. The request `$v` prints all non-zero ADB variables. The request `0$s` sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 7 shows the results of some formatting requests on the C program of Figure 6.

Figure 6. Simple C Program That Illustrates

Formatting and Patching

```
char    str1[]  "This is a character string";
int     one     1;
int     number  456;
long    lnum    1234;
float   fpt     1.25;
char    str2[]  "This is the second character string";
main()
{
        one = 2;
}
```

Figure 7. ADB Output Showing Fancy Formats

```
 adb a.out.shared -
executable file = a.out.shared
ready
 <b,-1?8ona
_str1:          052150  064563  020151  071440  060440  061550  060562  060543

_str1+0x10:     072145  071040  071564  071151  067147  0       0       01

_number:
_number:        0       0710    0       02322   037640  0       052150  064563

_str2+0x4:      020151  071440  072150  062440  071545  061557  067144  020143

_str2+0x14:     064141  071141  061564  062562  020163  072162  064556  063400
 <b,20?4o4^8Cn
_str1:          052150  064563  020151  071440  This is
                060440  061550  060562  060543  a charac
                072145  071040  071564  071151  ter stri
                067147  0       0       01      ng@`@`@`@`@`@a

_number:        0       0710    0       02322   @`@`@aH@`@`@dR

_fpt:           037640  0       052150  064563  ? @`@`This
                020151  071440  072150  062440   is the
                071545  061557  067144  020143  second c
                064141  071141  061564  062562  haracter
                020163  072162  064556  063400
address not found in a.out file
<b,20?4o4^8t8Cna
_str1:          052150  064563  020151  071440          This is
```

```
_str1+0x8:      060440  061550  060562  060543        a charac
_str1+0x10:     072145  071040  071564  071151        ter stri
_str1+0x18:     067147  0       0       01            ng@`@`@`@`@`@a
_number:
_number:        0       0710    0       02322         @`@`@aH@`@`@dR
_fpt:
_fpt:           037640  0       052150  064563        ? @`@`This
_str2+0x4:      020151  071440  072150  062440         is the
_str2+0xC:      071545  061557  067144  020143        second c
_str2+0x14:     064141  071141  061564  062562        haracter
_str2+0x1C:     020163  072162  064556  063400
address not found in a.out file
 <b,a?2b8t^2cn
_str1:          0x54    0x68            Th
                0x69    0x73            is
                0x20    0x69             i
                0x73    0x20            s
                0x61    0x20            a
                0x63    0x68            ch
                0x61    0x72            ar
                0x61    0x63            ac
                0x74    0x65            te
                0x72    0x20            r
$q
```

Patching

Patching files with ADB is accomplished with the **write**, **w** or **W**, request (which is not like the **ed** editor write command). This is often used in conjunction with the **locate**, **l** or L request. In general, the request syntax for `l` and `w` are similar as follows:

```
?l value
```

The request l is used to match on two bytes, `L` is used for four bytes. The request `w` is used to write two bytes, whereas `W` writes four bytes. The **value** field in either `locate` or `write` requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

```
adb -w file1 file2
```

When called with this option, `file1` is created if necessary and opened for both reading and writing. `file2` can be opened for reading but not for writing.

For example, consider the C program shown in Figure 6. We can change the word "This" to "The " in the executable file for this program, `ex7`, by using the following requests:

```
adb -w ex7 -
?l 'Th'
?W 'The '
```

The request `?l` starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of `?` to write to the `a.out` file. The form `?*` would have been used for a shared text file.

More frequently the request will be typed as:

```
?l 'Th'; ?s
```

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out -
:e arg1 arg2
flag/w 1
:c
```

The `:e` request is used to start `a.out` as a subprocess with arguments `arg1` and `arg2`. If there is a subprocess running ADB writes to it rather than to the file so the `w` request causes `flag` to be changed in the memory of the subprocess.

Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the `link` instruction. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. If a `:s` command is executed at a branch instruction, and the branch is taken, the command will act as a `:c` command. This is because a breakpoint is set at the next instruction and if is not reached, the process will not stop.

Command Summary

Formatted Printing

? *format*	print from *a.out* file according to *format*
/ *format*	print from *core* file according to *format*
= *format*	print the value of **dot**
?**w** *expression*	write expression into *a.out* file
/**w** *expression*	write *expression* into *core* file
?**l** *expression*	locate *expression* in *a.out* file

Breakpoint and Program Control

:b	set breakpoint at **dot**
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run `a.out` file under ADB control
:s	single step

Miscellaneous Printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

Calling the Shell

!	call *shell* to read rest of line

Assignment to Variables

>*name*	assign **dot** to variable or register *name*

Format Summary

a	the value of dot
b	one byte in hexadecimal
c	one byte as a character
d	two bytes in decimal
f	four bytes in floating point
i	MC68xxx instruction
o	two bytes in octal
n	print a newline
r	print a blank space
s	a null terminated character string
*n***t**	move to next *n* space tab
u	two bytes as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

Expression Components

decimal integer	e.g. 0d256
octal integer	e.g. 0277
hexadecimal	e.g. 0xff
symbols	e.g. flag _main
variables	e.g. <b
registers	e.g. <pc <d0
(expression)	expression grouping

Dyadic Operators

+	add
−	subtract
*	multiply
%	integer division
&	bitwise *and*
|	bitwise *or*
#	round up to the next multiple

Monadic Operators

~	not
*	contents of location
−	integer negate

Index

t

u

v

Table of Contents

atime

This manual describes a 680x0 assembly language sequence timing utility called *atime*. After you have developed and debugged assembly language code for a 680x0 processor (Series 300 computer), you can use *atime* to:

- analyze the performance of the code (*performance analysis mode*);
- determine the number of times each instruction is hit (*execution profiling mode*); or
- assert (verify) particular values in a code sequence to assure that various algorithms produce identical results (*assertion listing mode*).

Continuing to Get Information

Now that you know what *atime* does, please read the next three brief sections which:

- describe prerequisites for using *atime*,
- mention where to get additional or related information, and
- describe the sections in this manual. The descriptions of sections include suggestions for reading them.

Prerequisites

The following items mention requirements for using *atime*:

- Your system needs `/bin/as` and `/bin/ld`.
- You have a sequence of **assembler instructions** you want to test and have developed an **input file** containing the assembler instructions and special *atime* instructions (more on this later).
- You must run *atime* on a quiescent single-user system to get valid results. (The reason is that the utility returns empirically determined performance information.)

Getting Additional Information

In the *HP-UX Reference Manual*, you might want to examine the following related commands:

as(1)	The assembler
ld(1)	The link editor
prof(1)	A program that lets you display profile data
gprof(1)	A program that lets you display call graph profile data

Manual Contents

The following paragraphs name and describe subsequent sections in the manual. They also suggest how to use the information.

Atime and Assembly Code discusses the overall picture and shows how *atime* fits into the scheme of developing assembly code. (Skip this section if you already know what to expect or do not need to see this type of information.)

The Syntax with Examples describes *atime*'s syntax and options. Then, the section shows an example of running *atime* in performance analysis mode using an example of an input-file. (Some users may find that this section is all they need. Remaining sections simply discuss the input-file, atime instructions, modes, output, and errors.)

The Input File describes the four sections in an input-file. (Read this section to get more information if the previous examples did not provide enough information.)

The atime Instructions describes the *atime* instructions, including examples. (Read this section as necessary to learn how to use the instructions.)

Performance Analysis Mode describes performance analysis mode (the default mode). (Read this and the next two sections about modes according to your needs.)

Execution Profiling Mode describes execution profiling mode (use the `-p` option).

Assertion Listing Mode describes assertion listing mode (use the `-l` option).

Recovering from Errors describes error situations and how to handle them.

Atime and Assembly Code

In most cases, you develop assembly code to obtain maximum performance from, for example, a critical routine. During development, it may frequently be unclear as to which instruction, sequence of instructions, or algorithm can be executed most efficiently by the assembly instruction set. After you have developed and debugged two or more assembler instruction sequences, you can use *atime* to determine which sequence provides optimal performance. To do this, you run *atime* on each sequence and compare the results.

This section shows how *atime* fits into the development of assembly code and describes *atime*s features. (The remaining sections describe how to use them.)

The Overall Picture

Figure 1 shows where *atime* fits into the scheme of developing assembly language. It also shows the relationships between *atime* and the input-file, modes, and output.

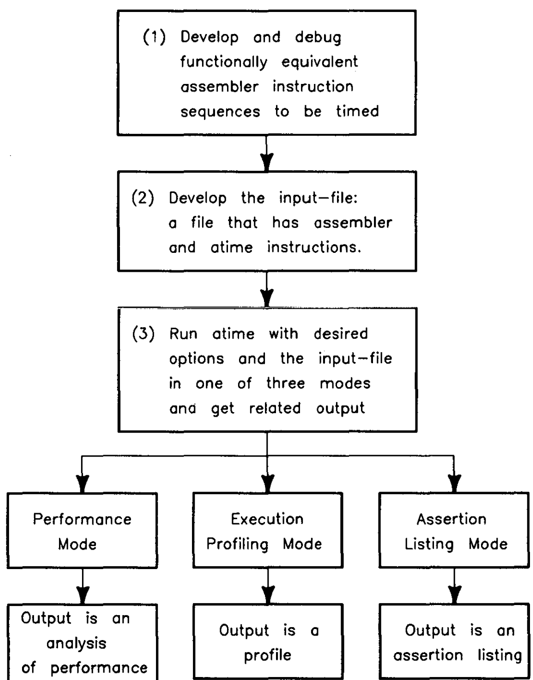


Figure 1. How atime Fits Into Developing Assembly Language

The atime Features

The *atime* utility has the following features:

- You can check the timing (speed) of functionally equivalent assembler instruction sequences (e.g. finding the most significant bit in a data register).
- You can specify sets of input data and the relative probability that each of them will occur.
- The utility runs in one of *performance analysis*, *execution profiling*, or *assertion listing* modes.
    - *Performance analysis* mode (the default) causes a code sequence to execute many times in a loop with *atime* calculating and reporting the average time per iteration.
    - Execution profiling mode (use the `-p` option) makes *atime* run all or selected data sets and reports the number of times each executable instruction is hit.
    - *Assertion listing* mode (use the `-l` option) causes *atime* to assert particular values in a code sequence for the purpose of assuring that various algorithms product identical results. You use this output to verify data for subsequent performance analyses and execution profiles.
- The utility provides output containing information you can compare with the output obtained from other runs to select the best sequence of assembler instructions.

Syntax with Examples

This section shows the general syntax. Then, it describes the command line options and shows two examples of an input-file: `bit_find` and `max_integers`.

The atime Syntax

The syntax is:

```
atime [options] input-file [output-file]
            |         |          |
            |         |          |_ Output goes to a specified file
            |         |             (if given) or to standard output
            |         |             if the name is - or is
            |         |             omitted.  Otherwise, if the mode
            |         |             is performance analysis and the
            |         |             input-file has an output
            |         |             instruction, output goes to the
            |         |             file specified there.
            |         |
            |         |_ Has four sections with assembly code source
            |            instructions and atime instructions.
            |
            |_ Use options to control such things as:
               * Specifying the mode;
               * Specifying an assertion data file;
               * Specifying a minimum number of timing iterations;
               * Turning off code sequence listing.
```

atime Options

-a*file*
: Specify an assertion data file to be used for assertion data. The file must have been created by a previous run of *atime* with the `-l` option. Only one `-a` option can be given and it will supersede any **`assert`** *file* instruction in the input-file.

-i*count*
: Specify the minimum number of timing iterations where *count* is an integer in the range 1 through $2^{32} - 1$ (you get an error otherwise). When data sets exist, the actual value used equals or exceeds the given *count* because the number of iterations must be an integral multiple of the sum of *counts* in all **`dataset`** instructions. Only one `-i` option can be used and it supersedes any **`iterate`** instruction in the input-file.

-l[*name*]
: Print asserted values. If *name* is given, the code sequence is executed using the dataset called *name* in the input-file. Multiple `-l` options are allowed. Omitting *name* prints assertions for all data sets. As each **`assert`** instruction in the input-file is executed, it prints its associated name and value. If an assertion file is specified by a `-a` option or an **`assert`** *file* instruction and there is a mismatch between the asserted value and the value in the file, that value is also printed. Also, an error is printed when a value is missing from the assertion file. Output goes to standard out unless you specify an output-file. An **`output`** instruction in the input-file is ignored. The output-file can be used as an assertion file in subsequent runs of *atime*. The `-l` option cannot be used with the `-p` option.

-n
: Turn off listing the input-file to output. It is ignored if you use `-p` or `-l`. This is equivalent to **`nolist`** in the input-file.

-p[*name*]
: Do execution profiling by printing hit counts for each timed instruction where *name* specifies the data set to analyze from the input-file. Multiple `-p` options print counts as the sums for all designated data sets. Omitting *name* profiles all data sets. The `-n` and `-i` options are ignored. Do not use the `-p` option with the `-l` option.

-t*text*
: Specify *text* as the output title (enclose multi-word titles in quotes, for example, **`"The First Sequence"`**). Leading and trailing blanks are ignored. Only one -t option can be given, and it will supersede any **`title`** instruction in the input-file.

An Example of an Input-file

This section shows two examples of input-files, which you create before running *atime*. The input-file contains assembler and *atime* instructions, and with command line options, it determines how *atime* works. **Be sure to debug the assembler instruction sequence in the input-file.**

A Rationale for Using atime

The two columns show two assembler instruction sequences that do the same thing (locate the most significant bit in the %d0 data register on a 68000 processor).

Sequence One

```
        movq    &31,%d1
L1:     btst    %d1,%d0
        dbne    %d1,L1
```

Sequence Two

```
        movq    &31,%d1
        cmp.l   %d0,&0xFFFF
        bhi.b   L1
        movq    &15,%d1
L1:     btst    %d1,%d0
        dbne    %d1,L1
```

The question is: "Which code sequence finds the bit in the least amount of time?" To get an answer, run *atime* and compare the returned information.

A Complete Input File

The following input-file named `bit_find` helps you examine code that finds the most significant bit. The example shows the four sections of an input-file. To help you differentiate instructions:

- A → precedes lines containing *atime* instructions.
- No → precedes lines having assembler instructions.

You could, for example, run *atime* in performance analysis mode (the default) and send the output to `/usr/stats/test-1` with:

```
atime bit_find /usr/stats/test-1
```

The four sections in the input-file, `bit_find`, look like this:

-------- *atime initialization section* -------

```
→       title       Example 1
→       comment     The algorithm finds the most significant bit set
→       comment     in an 8-bit number (original no. not destroyed)
```

```
→       dataname                $number
→       dataset     bit7,       0x80
→       dataset     bit6,       0x40
→       dataset     bit5,       0x20
→       dataset     bit4,       0x10
→       dataset     bit3,       0x08
→       dataset     bit2,       0x04
→       dataset     bit1,       0x02
→       dataset     bit0,       0x01
→       dataset     zero,       0x00
→       iterate     5000000
→       assert      "assertfile"
→       output      "logfile"
```

---------- *code initialization section* ---------

```
→       stack       even
          mov.l       &$number,%d0
→      code        even
```

---------- *timed section* --------

```
→     time
        mov.l       %d0,-(%sp)
        beq.b       L2
        movq        &31,%d1
     L1:
        btst        %d1,%d0
        dbne        %d1,L1
        bra.b       L3
     L2:
        movq        &-1,%d1
     L3:
        mov.l       (%sp)+,%d0
```

---------- *verify section* -------

```
→       verify
→       assert.l    original_value,%d0
→       assert.l    bit_number,%d1
```

A Second Example of an Input-file

Here is another input-file called `max_integers` (the → points to *atime* instructions).

--------- *atime initialization section* --------

```
→      title      Find the maximum of three integers
→      comment    Developed by T. R. Crew
→      comment    June 9, 1987
→      nolist
→      dataname                   $arg1,    $arg2,    $arg3
→      dataset    max1(70),         10,        4,        2
→      dataset    max2(35),          5,       11,        0
→      dataset    max3(20),          8,       13,       21
→      iterate    500000
→      assert     "assertfile"
→      output     "logfile"
→      ldopt      -lm -lc
```

--------- *code initialization section* --------

```
→      stack      even
       mov.l      &$arg1,%d0
       mov.l      &$arg2,%d1
       mov.l      &$arg3,%d2
→      code       even
```

--------- *timed section* --------

```
→      time
       cmp.l      %d0,%d1
       bge.b      L1
       exg        %d0,%d1
   L1: cmp.l      %d0,%d2
       bge.b      L2
       exg        %d0,%d2
   L2:
```

--------- *verify section* --------

```
→      verify
→      assert.l   max,%d0
```

This Page Left Blank for Notes:

The Input File

To use *atime*, you must create an *input-file*, which is specified in the *atime* command line. The *input-file* contains assembly code source instructions and special *atime* instructions, which look like assembler instructions. Together, these instructions let you obtain the timing data you need. The *input-file* has four sections, which are described next.

Section One: atime Initialization

Purpose: Set up the *atime* environment

Location: First line of file to first line of assembly code or *atime* `time`, `code`, or `stack` instruction.

Requirements: The following *atime* instructions can appear only in this section (the number in parentheses shows the maximum number of times an instruction can appear):

- `assert` *file* (1), `comment`, `dataname` (1), `dataset`, 1linclude, `iterate` (1), `ldopt` (1), `nolist` (1), `output` (1), `title` (1).
- `dataname` (if used) must precede `dataset` instructions.

Section Two: Code Initialization

Purpose: Set up environment for code to be timed

Location: Follows the *atime* initialization section and continues up to the `time` instruction.

Requirements: Note the following:

- Can contain any valid 680x0 assembler instruction.
- Can contain `code even/odd`, `stack even/odd`, or `include` instructions.
- Can contain instructions using dataname names; each possible replacement for *name* must yield a valid 680x0 instruction.
- You cannot make assumptions about the initial contents of registers. However, the stack pointer does point to a valid stack which can be used by code sequences. Be careful not to destroy data above this initial stack pointer. Registers (including stack and frame pointers) need not be saved and restored by the code sequence.

Section Three: Timed

Purpose: Time code sequence

Location: The `time` instruction up to the `verify` instruction, or to the end of the file.

Requirements: Any valid 680x0 assembler instruction or `include`.

Section Four: Verify

Purpose: Verify results

Location: From `verify` instruction to the end of the file.

Requirements: Any valid 680x0 instruction or `include` and/or:

```
assert.{b|w|l}
```

Input-file Requirements

- No branching among sections. Enter each section by falling into it from the end of the previous section. No checking occurs to report errors to the user. Trying to do this is undefined.
- Can use any valid 680x0 instruction where appropriate.
- Cannot use *m4(1)* macros or multiple instructions per line.
- Assembly code can reference external variables/routines if you provide for resolving them during linking.

The atime Instructions

The input-file contains two types of instructions: standard assembler instructions (the code you want to test for speed, code to do initialization, and code to aid in verification of results); and *atime* instructions (instructions that dictate how *atime* does its work).

Restrictions on atime Instructions

- Each instruction must be on a separate line.
- An instruction cannot be labeled.
- Comments cannot follow on the same line.
- If an instruction has a corresponding command line option, the option takes precedence.

A Quick Look at the Instructions

Table 1 lists the instructions; each instruction is described in detail following the table.

Table 1. The atime Instructions

Instruction	Function/Purpose
`assert.{b|w|l}` *name,location*	Verify a datum
`assert` *file*	Specifies a file used for assertion data
`code odd` `code even`	Changes code to odd or even word alignment.
`comment` *text*	Writes comments to the output
`dataname` *name, ..., name*	Defines names of data entries in dataset instructions
`dataset`	Defines one data set
`include "`*file*`"`	Includes text from *file*
`iterate` *count*	Specifies minimum number of timing iterations
`ldopt` *options*	Specifies link editor options
`nolist`	Turns off listing input-file contents to output-file
`output` *file*	Specifies an output-file
`stack odd` `stack even`	Adjusts stack for odd or even word alignment
`time`	Designates section of code to be timed
`title` *text*	Specifies text used as the title for output
`verify`	Designates section of code used for algorithm verification

assert

The syntax is:

`assert.{b|w|l}` *name,location*

Use **assert** to verify a datum, which enables consistency checking to verify that you get identical results when you compare two or more code sequences for performance.

assert in Performance Analysis/Execution Profiling Modes

Executing an **assert** instruction during performance analysis or execution profiling modes searches for *name* in an assertion file. The size and value associated with the *name* is compared with that of the *location* in the **assert** instruction. A mismatch gives an error. You also get an error when *name* is missing from the assertion file; or when an assertion file is not specified with either the **assert** *file* instruction or the **-a** command line option.

assert in Assertion Listing Mode

Executing **assert** in assertion listing mode prints the *name* and asserted value. If an assertion file is specified either with the **assert** *file* instruction or the **-a** command line option, the *name* is searched for there (you get an error if *name* is missing). The value in the file is printed when *name* exists and there is a size or value mismatch between it and the given *location*.

Additional Information About assert

- *name* identifies an asserted datum across *atime* executions.
  - For *name*, use an alphabetic character followed by 0 or more alphanumeric or underscore characters.
  - For location, use any data addressing mode such as `%d0` or `4(%a4,%d2.w)`
- The non-optional `b`, `w`, and `l` suffixes to `assert` indicate a size of byte, word, and long (respectively). Do **not** use the `b` suffix with the address register direct mode.
- Asserted values are treated as 2's complement signed integers.
- `assert` does not affect registers, stack, or condition codes.
- The size of this instruction in number of code bytes is not specified.
- An `assert` instruction must appear in the text segment and within the verify section of code. A given `assert` can be executed only once in a particular execution of a code sequence (ignores other attempts).

Example:

```
assert.l range,%d2
assert.w slip,-2(%a6)
tst.l    12(%a6)
smi      %d0
assert.b sign,%d0
```

assert file

Syntax is:

```
assert file
```

Lets you specify a file used for assertion data.

- Can appear only once in the *atime* initialization section of the input-file.
- For *file*, use an absolute or relative pathname.
- Having the `-a` option in the command line supersedes `assert` in the input-file.
- You can use the `-l` option to create an assertion file.

Example:

```
assert "assertdata"
```

code odd
code even

Changes the code to odd or even word alignment.

- Must appear in the text segment in the code initialization section.
- Cannot be executed in the timed section, but can be executed just before entering that section.
- Does not affect registers, stack, or condition codes.
- The actual size of these instructions in number of bytes is unspecified.

Example:

```
code    even
```

comment

Syntax is:

```
comment text
```

Lets you write any number of comments to the output.

- Must appear in the *atime* initialization section.

Example:

```
comment H. I. Que developed the code sequence
comment using a new algorithm.
```

dataname

Syntax is:

```
dataname name,name,...,name
```

Defines the names of data entries in **dataset** instructions.

- The first *name* corresponds to first *datum* in all **dataset** instructions, second *name* to second datum, and so on.
- Can have only one **dataname** instruction; it must be in the *atime* initialization section and precede all **dataset** instructions.
- Number of *names* in a **dataname** instruction must equal the number of data entries in **dataset** instructions.
- Names begin with **$** followed by one or more alphanumeric or underscore characters.
- Whitespace is ignored in the **dataname** list to allow specification of data sets in tabular form; whitespace cannot appear in a *name*.

Example:

```
dataname                  $time,    $speed,   $mass,    $part
dataset   bicycle(100),   0f120.0,  0f32.4,   0f55.2,   100
dataset   train(37),      0f24.14,  0f114.8,  0f1.5E4,  16
dataset   boat,           0f71.6,   0f37.7,   0f2500.0, -6
```

dataset

Syntax is:

```
dataset name[(count)],datum,datum,...,datum
```

Lets you define one data set. The input-file must have at least one **dataset** instruction when you include a **dataname** instruction (see dataname).

- *name* identifies the data set. It permits specifying a data set with the `-p` option for execution profiling or with the `-l` option for listing assertions.
- An optional *count* (greater than or equal to 1 and in parentheses) can follow *name* to specify the relative number of uses of the data set during timing (e.g. if one data set is 100 and another is 37, then, for each 100 executions of the first data set, the second set is executed 37 times). This lets you specify the probability of a data set being executed in a real environment. An omitted *count* defaults to 1.
- The sum of the *counts* in all **dataset** instructions (declared or defaulted) must have an integral multiple greater than or equal to the number of timing iterations and less than or equal to $2^{32} - 1$.
- You must give at least one *datum*
- The number of data items must be the same for all **dataset** instructions and must match the number of *names* in the **dataname** instruction.
- Data items must not contain commas because they are treated as strings.
- Having a *name* from a **dataname** instruction appear in an assembly instruction replaces the *name* with the corresponding string from the **dataset** instruction currently considered.
- Whitespace between items in a **dataset** list is ignored to provide for specifying data sets in a tabular format.

Example:

```
dataname                    $time,    $speed,   $mass,     $part
dataset   bicycle(100),     0f120.0,  0f32.4,   0f55.2,    100
dataset   train(37),        0f24.14,  0f114.8,  0f1.5E4,   16
dataset   boat,             0f71.6,   0f37.7,   0f2500.0,  -6
```

include

Syntax is:

`include "`*file*`"`

Includes text from *file* as follows:

- The file name can be an absolute or relative pathname.
- The `include "`*file*`"` instruction can appear anywhere in an input-file, but not in an include-file.

Example:

```
include "srcdata"
```

iterate

Syntax is:

`iterate` *count*

Specify the minimum number of timing iterations. (See *count* in `dataset` above for range.)

- With data sets, the value used for *count* is equal to or greater than the value given here because the number of iterations must be an integral multiple of the sum of the *count*s in all `dataset` instructions.
- You get an error if the calculated iteration *count* falls outside the range; *atime* terminates.
- Only one `iterate` instruction can be used and it must appear in the *atime* initialization section.
- The `-i` option supersedes an `iterate` instruction.
- The default (not specified) timing iteration value is 1000000.

Example:

```
iterate 3000000
```

ldopt

Syntax is:

```
ldopt options
```

Specifies link editor options. An **ldopt** instruction passes its options to the link editor. Only one instruction can be used and it must appear in the *atime* initialization section.

Example:

```
ldopt ext_func.o -lm
```

nolist

Syntax is:

```
nolist
```

Turns off listing the input-file contents to the output-file.

- Only one instruction can be used and it must appear in the *atime* initialization section.
- Listing is turned off for the whole file and for any include-file(s).
- A **nolist** instruction is ignored when you use the **-p** or **-l** options.

Example:

```
nolist
```

output

Syntax is:

```
output file
```

Specifies an output-file where *file* can be an absolute or relative pathname.

- Output is appended to this file.
- Only one `output` instruction can be used and it must appear in the *atime* initialization section.
- An `output` instruction is ignored when you use the `-p` or `-l` options.

Example:

```
output "/usr/stats/structmove"
```

stack odd
stack even

Adjusts the stack for odd or even word alignment by checking the current alignment and subtracting 2 (if necessary) from the stack pointer.

- Use only in the code initialization section.
- Because the stack pointer can change, memory locations referenced as offsets from the stack pointer can have their offsets changed.
- These instructions do not affect condition codes or any registers other than the stack pointer.
- The size of these instructions in terms of number of code bytes is not specified.

Example:

```
stack odd
```

time

Syntax is:

```
time
```

Designates a section of code to be timed.

- Timing of code begins with the line following the `time` instruction and continues up to a `verify` instruction or to the end of the file.
- There can be only one timed section and it must be wholly within the program's text segment.

Example:

```
mov.l     &$value,%d0
time
mov.l     %d0,%d1
swap      %d0
add.l     %d1,%d0
mov.l     %d0,(%a0)
verify
movq      &1,%d0
and.l     (%a0),%d0
```

title

Syntax is:

`title` *text*

Specifies text used as a title for output.

- Only one `title` instruction can be used and it must appear in the *atime* initialization section.
- A `-t` option supersedes a `title` instruction.

Example:

```
title  ALGORITHM 1 - values saved on stack
```

verify

Syntax is:

```
verify
```

Designates a section of code used for algorithm verification.

- The `verify` section begins with the line following the `verify` instruction and continues to the end of the file.
- This section normally contains one or more `assert` instructions.

Example:

```
mov.l      &$value,%d0
time
mov.l      %d0,%d1
swap       %d0
add.l      %d1,%d0
mov.l      %d0,(%a0)
verify
assert.l   result,%d0
```

Performance Analysis Mode

This default mode lets you analyze the performance of your assembly code.

To analyze performance, an assembly code sequence is conceptually executed many times in a loop. The total time for execution (minus overhead) divided by the number of iterations gives an average execution time, which is reported to you. For sequences of code that do the same thing, the sequence having the lowest average has the greatest speed.

Using Command Line Options

- Valid options include: `-a`, `-i`, `-n`, and `-t`.
- Do not use `-p` or `-l` because they cause *atime* to do execution profiling or assertion listing, respectively.
- Use an option only once in any order before the input-file name.

Getting and Reading Output (the analysis)

You get output as follows:

- appends to the output-file if you specified one in the command line.
- appends to the file in an `output` instruction if you specified one in the input-file.
- goes to standard out if you:
  - did not specify anything.
  - used – (minus) for the output-file in the command line.

An Example

The following example with annotations shows the order and appearance of the output.

`-----------------------------`	Separator line between sequences
`Find the Maximum of Three Integers`	Title if given by -t or title
`Developed by T. R. Crew`	Comment in comment instruction(s)
`June 9, 1987`	Comment in comment instruction(s)
`name: robert`	Login name
`machine: system1`	Computer hostname
`date: Tue Jun 9 16:33:04 1987`	Date (day, month, date, time, year)
`size: 12 bytes`	Size of timed section in bytes
`instructions: 6`	Number of executable instructions in timed section
`iterations: 50000`	Number of actual iterations
`avg. time:  780.408 nsec`	verage execution time
`(Note: The entire contents of the input-file and any include-file(s) appears here.)`	The input-file (including text from include-files) when `-n` and `nolist` are not given.

Showing the Average Time

The average time is presented according to the following format:

`0.0 sec`	for less than 1 nsec
`ddd.ddd nsec`	for 1 nsec to 999.999 nsec
`ddd.ddd usec`	for 1 μsec to 999.999 μsec
`ddd.ddd msec`	for 1 msec to 999.999 msec
`dd.ddd sec`	for 1 sec to 59.999 sec
`dd min dd.ddd sec`	for 1 min to 59 min, 59.999 sec
`dddd hr dd min dd.ddd sec`	for 1 hour or greater

Execution Profiling Mode

The execution profiling mode of *atime* gives you a profile by executing a code sequence, tallying how many times each instruction is executed. Here is the overall scheme:

- Given a list of data sets for doing execution profiling, the number of times a particular data set is executed in the process of tallying instruction hits equals the *count* associated with its particular `dataset` instruction (not specifying *count* defaults it to **1**; and if there are no data sets, the code sequence executes once).
- The mode tallies those instructions recognized as executable by the 680x0 assembler. It excludes other instructions such as data initialization (e.g. `byte`), symbol definition (e.g. `set`), and alignment (e.g. `lalign`).
- The mode aids in defining data sets. In setting up code for timing, you will usually specify at least one data set to execute a particular set of paths in the code. Having the execution printing mode **on** for that data set verifies that the set of paths **is** what is executed.
- After defining data sets, *atime* can determine if all code will be executed by running execution profiling for all data sets collectively. When you notice certain instructions not getting hit, you can add more data sets to cover those cases.

Using Command Line Options

- You must have at least one `-p` option to use the mode.
- Other options include `-a`, `-i`, `-n`, and `-t`; but `-i` and `-n` have no effect. Use at most one of each of the "other" options in any order before the input-file name. Duplicate usage of a particular option prints a warning message and ignores all but the first usage.
- Using `-l` causes an error and terminates execution.

Getting and Reading Output (the profile)

You get output as follows:

- appends to the output-file if specified in the command line.
- goes to standard out if you did not specify anything or you used – for the output-file.
- ignores an `output` instruction in the input-file

An Example

The following example shows how execution profiling mode prints information.

`-----------------------------`	Separator line between sequences
`Find the Maximum of Three Integers`	Title if given by -t or title
`Developed by T. R. Crew`	Comment in comment instruction(s)
`June 9, 1987`	Comment in comment instruction(s)
`name: robert`	Login name
`machine: system1`	Computer hostname
`date: Tue Jun 9 16:33:04 1987`	Date (day, month, date, time, year)

The remaining output has dataname and dataset lines as they appeared in the input-file and profile information in two fields: number of executions and executed assembler instructions.

```
                        $arg1,  $arg2,  $arg3
            max1(70),     10,       4,      2
            max2(35),      5,      11,      0
            max3(20),      8,      13,     21
125         cmp.l          %d0,%d1
125         bge.b          L1
 55         exg            %d0,%d1
125   L1:   cmp.l          %d0,%d2
125         bge.b          L2
 20         exg            %d0,%d2
      L2:
```

Assertion Listing Mode

The assertion listing mode of *atime* lets you determine that results are identical for every code sequence variation.

- Upon executing a code sequence for a specified data set, each **assert** instruction prints its asserted value. If an assertion file is specified, the value is checked against its corresponding value in the file; and on a mismatch, the value in the assertion file is also printed. Not having a value in the assertion file prints an error message.
- Besides printing code sequence results, output of an assertion listing can be put into a file and used as the assertion file in subsequent runs of *atime*.

Using Command Line Options

- You must specify at least one `-l` option.
- Other valid options include: `-a`, `-i`, `-n`, and `-t`, but `-i` and `-n` have no effect. Use at most one each of valid "other" options. Any order is accepted; the options must appear before the input-file. Having more than one of any particular option generates a message and *atime* ignores the extras.
- Using `-p` generates an error and terminates execution.

Getting and Using Output

You get output as follows:

- The information in the first six lines is the same as that shown for other modes.
- The assertion listing information begins with **dataset**: followed by the name of the data set (each data set requires a name).
- Then, you see each datum in the data set as its name followed by its value.
- On executing a code sequence, each asserted value is printed as its name followed by its value.
- If an assertion file is specified and it has a different corresponding value, that value is also printed.
- You get `MISSING` when a value is missing from the assertion file.
- Asserted values have a size suffix.

An Example

The following example shows how assertion listing mode prints information.

`-----------------------------`	Separator line between sequences
`Find the Maximum of Three Integers`	Title if given by -t or title
`Developed by T. R. Crew`	Comment in comment instruction(s)
`June 9, 1987`	Comment in comment instruction(s)
`name: robert`	Login name
`machine: system1`	Computer hostname
`date: Tue Jun 9 16:33:04 1987`	Date (day, month, date, time, year)

The remaining output shows the assertion information according to the above description on getting ouptut.

```
dataset: max1
        $arg1      10
        $arg2       4
        $arg3       2
        max      10.1
dataset: max2
        $arg1       5
        $arg2      11
        $arg3       0
        max      11.1
dataset: max3
        $arg1       8
        $arg2      13
        $arg3      21
        max      21.1
```

Recovering from Errors

The *atime* utility provides self-explanatory error messages. In addition, you can get error messages from the assembler or link editor. When assembly fails, an intermediate, temporary file is retained with the error message indicating its name. The file is important because it contains comments that help you correlate assembly errors with errors in the *input-file.*

Tracking Errors

Recall that `bit_find`, the input-file for finding the most significant bit, contained the line:

```
btst    %d1,%d0
```

Suppose, for example, the line had a typing mistake and read:

```
btst    %a1,%d0
```

Running *atime* on this file would return an error message similar to:

```
as error: "/usr/tmp/aaaa22982" line 37: syntax error
           (opcode/operand mismatch)
ERROR: cannot assemble file: "/usr/tmp/aaaa22982"
```

Looking at lines 36 and 37 in `/usr/tmp/aaaa22982`, you would see:

```
# "bit_find", line 25
     btst    %a1,%d0
```

This information tells you the error is in line 25 in the *input-file* called `bit_find`. Knowing this, you can locate the error in the original input-file and make necessary corrections (i.e. change `%a1` to `%d1`).

Remember to remove the temporary file when you finish using it.

Data Set Errors

Suppose you made a typing error for data set bit5 by typing:

```
dataset     bit5,   0x2X
```

which will create the erroneous instruction:

```
mov.l   &0x2X,%d0
```

You would get an error similar to:

```
as error: "/usr/tmp/aaaa22997" line 116: syntax error
           (opcode/operand mismatch)
as error: "/usr/tmp/aaaa22997" line 116: syntax error
ERROR: cannot assemble file: "/usr/tmp/aaaa22997"
```

The code in `/usr/tmp/aaaa22997` around line 116 could look like:

```
___Zcode2:      # "bit_find", line 18, dataset: bit5
                # mov.l &$number,%d0
     mov.w   %cc,__Zcodecc
     mov.l   (%sp)+,__Zcodesp
     addq.w  &4,%sp
     mov.w   __Zcodecc,%d0
     mov.l   &0x2X,%d0
     mov.w   %cc,__Zcodecc
     mov.l   __Zcodesp,-(%sp)
     mov.w   __Zcodecc,-(%sp)
     rtr
```

Backing up from line 116 and looking at the comments, you see:

- The file is `bit_find`.
- The error occurred on line 18, which is:

  ```
  mov.l   &$number,%d0
  ```

- The offending data set is called `bit5`.

Assert Instruction Errors

Suppose you made an error in one of the `assert` instructions:

```
assert.l   original_value,%d9
```

Running *atime* would return:

```
as error: "/usr/tmp/aaaa23012" line 58:
           invalid register symbol (%d9)
as error: "/usr/tmp/aaaa23012" line 58: syntax error
           (opcode/operand mismatch)
as error: "/usr/tmp/aaaa23012" line 58: syntax error
ERROR: cannot assemble file: "/usr/tmp/aaaa23012"
```

Lines 57 and 58 in `/usr/tmp/aaaa23012` look like:

```
mov.w    %cc,__Z     # "bit_find", line 33
mov.l    %d9,__ZEA     # assert.l original_value,%d9
```

Again, the comments indicate the file, offending line, and instruction in the original file.

Some Notes About Error Recovery Procedures

Looking back at the three examples of error recovery, you see a similar pattern:

- Examine the error messages, looking for clues.
- Look at the temporary file according to implied line numbers.
- Study the code and comments to find the error.
- Correct the error in the appropriate files.

Atime catches errors associated with setting up the analysis environment. With assertions, it also detects differing results between code sequences. In addition, certain types of errors are caught by the assembler or link editors. Beyond this, there are particular runtime errors that cannot be tracked down effectively except outside of using *atime*. Such errors include bad pointer dereferences and executing infinite loops. In all cases, it is best to run *atime* only on code sequences you have thoroughly tested beforehand.

Index

a

c

l

m

n

HEWLETT
PACKARD

HP Part Number
98597-90020

Microfiche No. 98597-99020
Printed in U.S.A. E1288

98597-90638
For Internal Use Only