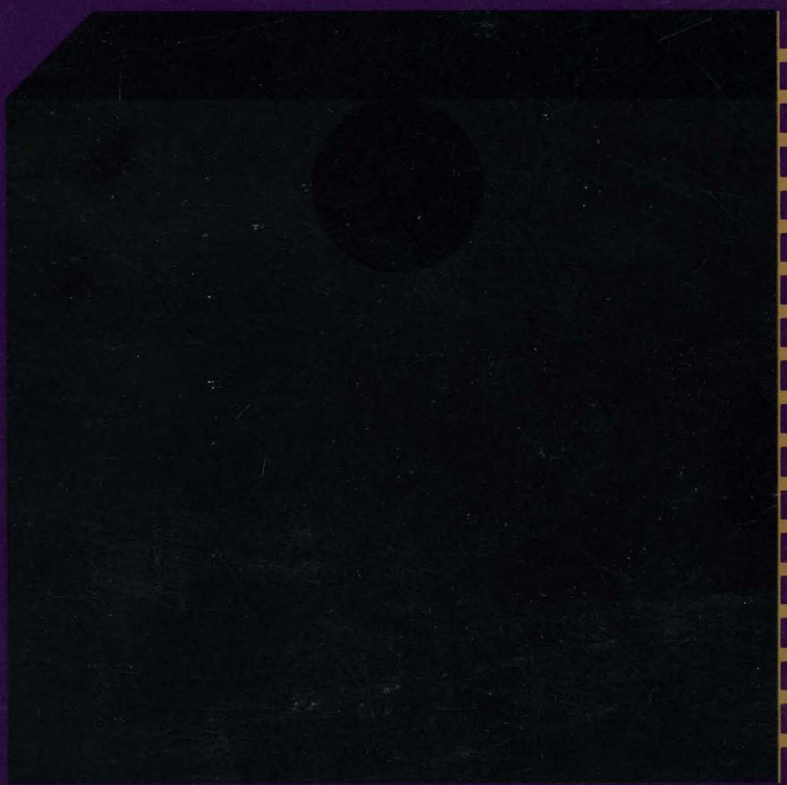
ANALOG DEVICES



ADSP-2101

Cross-Software Manual

Programming Reference

You may contact the Digital Signal Processing Division in the following ways:

- By contacting your local Analog Devices Sales Representative
- For Marketing information, call (617) 461-3881 in Norwood, Massachusetts, USA
- For Applications Engineering information, call (617) 461-3672 in Norwood, Massachusetts, USA
- The Norwood office Fax number is (617) 461-3010
- The Norwood office may also be reached by
  Telex: 924491
  TWX: 710/394-6577
  Cables: ANALOGNORWOODMASS
- The DSP Division runs a Bulletin Board Service that can be reached at 300, 1200, or 2400 baud, no parity, 8 bits data, 1 stop bit by dialing:
  (617) 461-4258
- By writing to:
  Analog Devices
  DSP Division
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA

ADSP-2101 Cross-Software Manual

PRINTED IN USA FIRST EDITION

Literature

ADSP-2101 MANUALS

ADSP-2101 User's Manual/Architecture (preliminary)
Complete description of architecture and system interface.

ADSP-2101 Cross-Software Manual
Complete programmer's reference including C compiler.

APPLICATIONS INFORMATION

ADSP-2100 Family Applications Handbook, Volume 1
Topics include arithmetic, filters, FFTs, LPC, modem algorithms.

ADSP-2100 Family Applications Handbook, Volume 2
Topics include graphics, pulse-code modulation, multirate filters, DTMF.

ADSP-2100 Family Applications Handbook, Volume 3
Topics include optimized and 2D FFTs, memory interface, multiprocessing, host interface, sonar beamforming.

SPECIFICATIONS INFORMATION

ADSP-2101 Data Sheet (preliminary)

Contents

Contents

CHAPTER 3 ASSEMBLER

Contents

CHAPTER 4 LINKER

CHAPTER 5 SIMULATOR FUNCTIONS

Contents

Contents

CHAPTER 6 SIMULATOR CONFIGURATIONS

Contents

CHAPTER 7 C COMPILER

Contents

Contents

Contents

Contents

APPENDIX E LINKER OPERATION

APPENDIX F ERROR MESSAGES

Contents

FIGURES

Contents

TABLES

Overview 1

1.1 INTRODUCTION

The ADSP-2101 Development System is a complete set of software and hardware development tools. The Development System includes the Cross-Software system to aid the software design and a real-time hardware Emulator to facilitate the debug cycle.

The Cross-Software system includes six separate programs: System Builder, Assembler, Linker, Simulator, PROM Splitter and C Compiler. These programs are described in the following section.

Release 2.0 and later of the Cross-Software system runs on the IBM-PC under PC-DOS and on the Sun-3 workstation under Unix (Bsd 4.2). For information on host-specific system requirements, refer to Appendix C. For information on support for other machine types and operating systems, contact Analog Devices, Digital Signal Processing, Marketing Division. (See the contact information on the copyright page.)

This manual is a complete programmer's reference. For information on the architecture and system interface of the ADSP-2101, refer to the *ADSP-2101 User's Manual*.

> *Each release of the Cross-Software is shipped with a Release Note. This note describes the current version and provides information on any updates to the software. If you return the registration card enclosed with your Cross-Software, you will receive a Release Note for each subsequent update of the software.*

1 Overview

1.1.1 ADSP-2101 Cross-Software System & Manual

This manual describes the Cross-Software system in the following chapters:

- System Builder, Chapter 2

The System Builder is a software tool to describe the target system. The System Specification source file is created, which specifies the amount of RAM and ROM available, the allocation of program and data memory and any memory mapped I/O ports for the target hardware environment. High-level constructs are used to simplify this task.

- Assembler, Chapter 3

The Assembler assembles source code. It supports the high-level syntax of the instruction set and provides flexible macro processing. A C language preprocessor handles C directives in source code. Source code may be partioned into a defined set of files (modules) and assembled in one pass using the "include file" capability. A full range of diagnostics is also provided.

- Linker, Chapter 4

The Linker links separately assembled modules. It searches directories for library routines to link in. It maps the linked code and data output to the target system hardware, as specified by the System Builder output, and can produce multiple boot memory page image files.

- Simulator Functions, Chapter 5

The Simulator performs instruction-level simulation. The user interface is both interactive and symbolic, and supports symbolic disassembly. The Simulator fully simulates the hardware configuration described by the System Builder. It flags illegal operations and provides several displays of the internal operations of the ADSP-2101 microcomputer.

- Custom Simulator Configurations, Chapter 6

The ADSP-2101 Simulator supports a user-configurable interface of windows and commands. This chapter describes how to customize the interface for your preferences and how to store and recall screens and customized commands.

- C Compiler, Chapter 7

The C Compiler supports the proposed ANSI Standard version of the popular C programming language. The Compiler produces ADSP-2101 source code and can directly invoke the Assembler.

- PROM Splitter, Chapter 8

The PROM Splitter reads the Linker-output executable file and generates PROM burner compatible files in a variety of industry standard formats. Boot memory requirements are supported by the PROM Splitter.

- Instruction Set Reference, Chapter 9

Chapter 9 provides a reference section for each ADSP-2101 instruction group. Running headers in this chapter allow you to look up any instruction.

These chapters are supplemented by several appendices:

- Appendix A is a complete reference to ADSP-2101 opcodes.
- Appendix B describes the file format for input and output files used by the Cross-Software system.
- Appendix C lists the hardware and software requirements for the computer systems that can host the Cross-Software system and any differences between the operation of the Cross-Software on each system.
- Appendix D lists the differences between ADSP-2101 C and the ANSI draft standard.
- Appendix E details how the Linker handles data and code used on multiple boot pages.
- Appendix F lists and defines all error messages generated by the Cross-Software modules.

1 Overview

1.1.2 Development Flow

Figure 1.1 shows a flow chart of the ADSP-2101 development cycle.

The development process begins with the task of defining the target system hardware environment. To define the hardware environment, you use the System Builder. The System Specification file includes the target hardware information. The System Builder reads this file and creates an Architecture Description file which passes information about the target hardware to the Linker, Simulator, and Emulator.

You begin code generation by creating assembly source code modules. An assembly module is a unit of source code such as a calling program, subroutine, data buffer declaration section or any combination. Each assembly code module is assembled separately by the Assembler. Several modules are then linked together to form an executable program.

The Linker needs the target hardware information located in the Architecture Description file to determine placement of code and data fragments. In the assembly modules you have the option to specify each code/data fragment as completely relocatable, relocatable within a defined memory segment, or placed at an absolute address. Absolute code or data modules are placed at the specified base address, provided the specified memory area has the correct attributes. Relocatable objects are placed in memory by the Linker.

Using the Architecture Description file and the Assembler output files, the Linker determines the placement of relocatable code and data segments (including circular buffers), and places all segments in memory locations with the correct attributes (CODE or DATA, RAM or ROM). The Linker generates an executable image file, which may be loaded into the Simulator and Emulator for debugging.

The Simulator provides windows that display different aspects of the hardware environment. To replicate the target hardware environment, the Simulator configures its memory according to the System Builder output, and simulates I/O ports according to user-entered Simulator commands. This simulation provides capabilities to debug the system and analyze performance before committing to a hardware prototype.

After debugging with the Simulator, the Emulator is used in the prototype target system to debug hardware, timing, and real-time software problems. It provides overlay memory to replace target system off-chip memory, including boot memory, if desired.

The PROM Splitter translates the executable memory image file (Linker output) into a file that is compatible with a PROM burner. Once you burn the ADSP-2101 code into PROM and plug an ADSP-2101 into the target board, your prototype is ready to run.

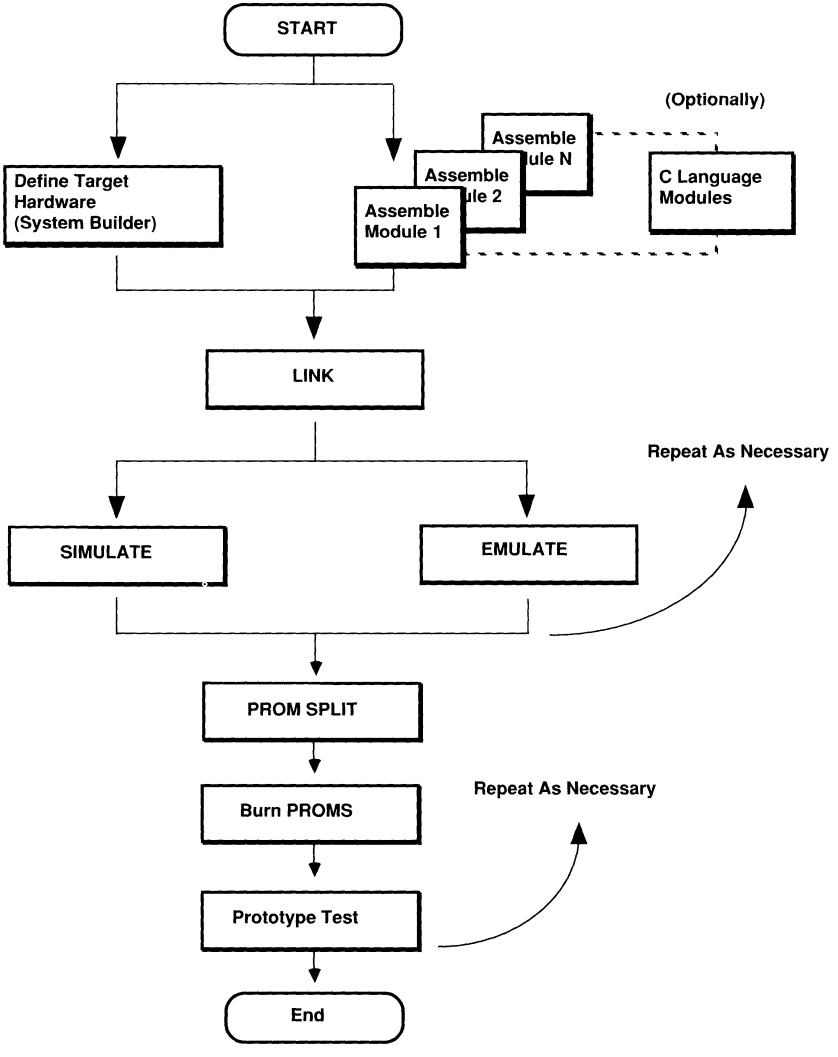


Figure 1.1 ADSP-2101 System Development Flow

1 Overview

1.2 EXPRESSION HANDLING IN CROSS-SOFTWARE TOOLS

The ADSP-2101 Cross-Software tools support general expression evaluation in locations where constants are valid. You may in most cases use an expression instead of a constant wherever a constant is expected.

Expressions are composed of numerical constants, symbolic constants, and expression operators. The operators are a subset of the arithmetic and logical operators of the C programming language (for integer values only). In order of precedence, the operators are:

( )	left, right parenthesis
~ –	ones complement, unary minus
* / %	multiply, divide, modulus
+ –	addition, subtraction
<< >>	bitwise shifts
&	bitwise AND
|	bitwise OR
^	bitwise XOR

Examples:

(*taps* + 16) / 3 *mask* & 0x55

The ADSP-2101 Simulator recognizes an additional set of expression elements and operators. These are detailed in the "Simulator Expressions" section of Chapter 5.

The most important difference between Assembler expressions and Simulator expressions is that memory contents (such as data variables) and processor register contents may be used as operands *in the Simulator only*. The Assembler cannot evaluate memory and register values at assembly-time; the Simulator, however, has access to the instantaneous values of simulated memory and registers.

1.3 CONSTANTS

Constants include numeric (or literal) constants and identifiers defined as symbolic constants. Symbolic constants can be used anywhere to replace numeric constants. The identifier must be declared a constant with the .CONST directive; see the discussion under "Assembler Directives" in Chapter 3.

1.4 NUMERIC BASES

The numeric bases which may be used in the ADSP-2101 Simulator and in source code are hexadecimal, octal, and decimal. They are specified as follows:

For hexadecimal prefix a 0x (zero and x) or H#:

0x12FA H#12FA

For octal prefix a 0 (zero):

0777

For decimal (the default) there is no prefix to denote the base. Sign (+ or –) may be specified:

1024 +1024 –55

Binary numbers are accepted only by the Assembler in a source code file, and may not be used with any of the other Cross-Software tools. Binary numbers are specified with the prefix B#:

B#0111010001011111

1.5 CHARACTER SET

The ADSP-2101 Cross-Software character set includes the following:

- Uppercase letters, "A" through "Z"
- Lowercase letters, "a" through "z"
- Digits, "0" through "9"
- The ASCII graphics characters; the printing characters other than letters and digits (punctuation, etc.).
- The ASCII non-graphics: space, tab, carriage return, line feed and form feed. (The "newline" character or characters are interpreted correctly as per the conventions of the environment in which they occur.)

1 Overview

1.6 IDENTIFIERS (SYMBOLS)

Symbols are either a user-defined identifier or system-reserved keyword. The keywords are listed in Chapter 3, Assembler, in Table 3.3.

Identifiers consist of a character from the set:

- Uppercase letters, "A" through "Z"
- Lowercase letters, "a" through "z"
- The underscore character "_"

followed by a sequence of characters from the set:

- Uppercase letters, "A" through "Z"
- Lowercase letters, "a" through "z"
- Digits, "0" through "9"
- The underscore character "_"

An identifier may have a maximum of 32 characters.

The Cross-Software tools can be either case-insensitive, with uppercase and lowercase letters treated as the same character, or case-sensitive, with differentiation between the two forms.

1.7 MANUAL NOTATION CONVENTIONS

This section provides you with a list of notation conventions.

- With the increasing use of the C Compiler (a case-sensitive programming environment) the traditionally case-insensitive Assembler and System Builder tools now support case-sensitivity as an option. The actual commands used to invoke each tool, BLD21, ASM21, LD21, etc., may be entered in upper or lower case on the PC but must be lowercase on the Sun.

- In this manual keywords (reserved symbols) are always shown in UPPERCASE, although they may be entered in either upper or lower case. Any form of the keyword is reserved.

- A lowercase word highlighted in italics, such as *jumplabel*, indicates an identifier used as an address label, data variable, etc. or a filename.

- Square brackets, [ ] , enclose optional specifications or data buffer length (literal usage); when specifying buffer length, the brackets must be used in source code.
- An ellipsis, ... , indicates that the preceding item may be repeated.
- Carriage return is represented by "Return" or <cr>. (Simulator chapter only)
- ^ denotes the control, or CNTL, key, as in a key entry sequence: ^X (Simulator chapter only)

System Builder 2

2.1 INTRODUCTION

The System Builder module of the Cross-Software system is a software tool for describing your hardware environment. Each ADSP-2101 system can have a unique hardware configuration, and may not require the full complement of possible memory. The System Builder output specifies your hardware configuration, including memory and I/O ports, in a form used by the rest of the Cross-Software system.

A target system may include:

	Maximum Available
Data Memory (16-bit data, ROM or RAM)	Up to 15K words (1K on-chip, up to14K off-chip, 1K reserved)
Program Memory (24-bit code or data, ROM or RAM)	Up to 16K words, mixed code & data (2K on-chip, up to 14K off-chip)
Boot Memory (24-bit code or data, padded to 32-bit word width)*	Up to 64K bytes, configured as 16K words (1 to 8 pages, each containing 2K words)
Memory-mapped I/O Ports	Any number, up to memory limits (Simulator limited by host file system limits)

Table 2.1 ADSP-2101 System Configurations
*see Chapter 8, PROM Splitter, for details.

2 System Builder

You specify your hardware configuration in a System Specification source (.SYS) file using System Builder directives. The System Builder processes the .SYS file and generates the Architecture Description file (.ACH). The Architecture Description file is used by the Linker to place relocatable segments in memory, by the Simulator to simulate memory configurations, and by the Emulator to set up target system memory mapping. The System Builder outputs error messages, if any, or a summary of the architecture created to the screen. You should use the operating system facilities of your computer to capture this output into a file if you need to refer to it for debugging or documentation purposes.

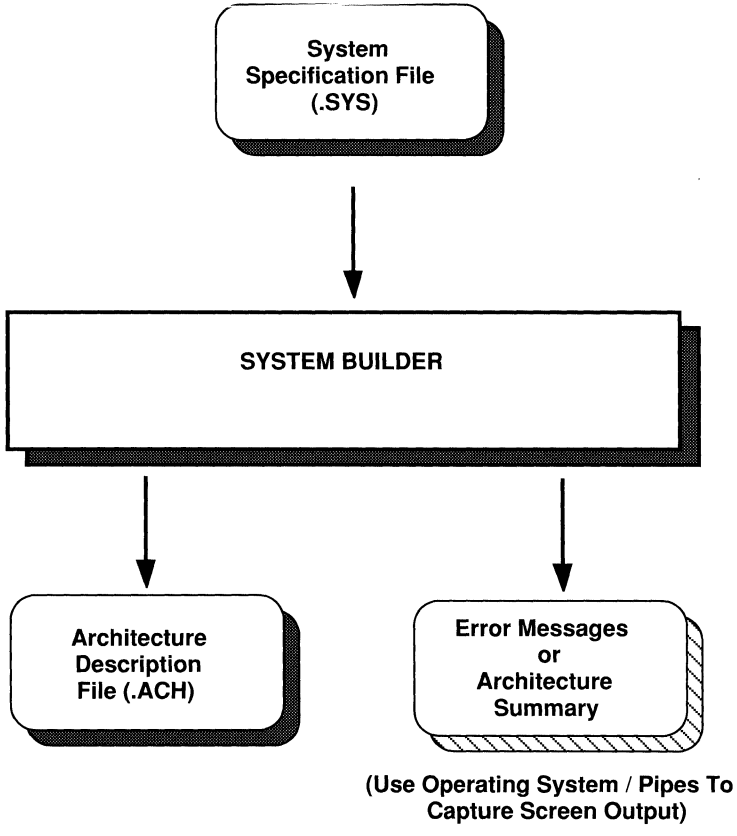


Figure 2.1 System Builder I/O

2.2 RUNNING THE SYSTEM BUILDER

To invoke the System Builder, type:

BLD21 *filename*[*.ext*] [–switch]

where *filename.ext* is the system specification source file. The filename extension is optional and defaults to .SYS.

There is one switch for invoking the System Builder. The –c switch makes the System Builder case-sensitive. This is provided primarily for compatibility with the C Compiler, which is always case-sensitive.

If the –c switch is not used, the System Builder output is in all uppercase. You must use this switch in order to preserve the case of characters as they are entered. This is necessary if the Assembler is to be run with its case-sensitive switch, as is required when assembling C-compiled code. If you refer (in assembly code) to a memory segment declared in the System Builder which is in lowercase, and the Assembler is run in case-sensitive mode, the segment name will not be recognized unless its case is preserved by the System Builder.

2.3 LANGUAGE CONVENTIONS

In a System Specification file, symbolic names are assigned to the system configuration itself, I/O ports, and memory segments. The memory segment names may be used in the Assembler; memory segment names and memory characteristics are used by the Linker.

All symbolic names must be unique. A symbolic name is a string of letters, digits, and underscores with a letter as the first character. Symbol names can be of any length. Only 32 characters are significant.

System Builder keywords cannot be used as symbolic names. Table 2.2 lists the System Builder keywords.

ABS	CODE	ENDSYS	PORT	SYSTEM
ADSP2100	CONST	MMAP0	RAM	
ADSP2101	DATA	MMAP1	ROM	
BOOT	DM	PM	SEG	

Table 2.2 System Builder Keywords

2 System Builder

Assembler keywords, listed in Table 3.3, may not be used as symbolic names either. The System Builder accepts such symbol definitions without flagging an error, however, the Linker does not.

Numeric constants and general expressions are accepted by the System Builder. See Chapter 1 for a description of allowed constants and the definition of expressions. For a description of the notation used in this manual, refer to the section "Manual Notation Conventions" in Chapter 1.

2.4 SYSTEM SPECIFICATION SOURCE FILE EXAMPLE

Figure 2.2 is an example of a system specification source (.SYS) file for an ADSP-2101 system.

Comment fields are enclosed within braces, { }, and can be inserted anywhere in the file. Nested comments are not allowed.

2.4.1 ADSP-2101 System Specification File

The System Specification Source file for the ADSP-2101 specifies the amount of data, program, and boot memory included in your development system.

The first directive in the file is the .SYSTEM directive. This directive assigns a name *fir_system* to the hardware description and signals the start of the file.

The .ADSP2101 statement identifies the processor type, here naming the ADSP-2101 microcomputer. This statement is required. The presence of

```
.SYSTEM fir_system;                          {system name}
.ADSP2101;                                   {ADSP-2101 system}
.MMAP0;                                      {boot loading enable}
.SEG/ROM/BOOT=0 boot_mem[2048];              {boot page one}
.SEG/PM/RAM/ABS=0/CODE/DATA int_pm[2048];    {on-chip program mem}
.SEG/PM/RAM/ABS=2048/CODE/DATA ext_pm[14336]; {external program mem}
.SEG/DM/RAM/ABS=0/DATA ext_dm[14336];        {external data mem}
.SEG/DM/RAM/ABS=14336/DATA int_dm[1024];     {on-chip data mem}
.ENDSYS;
```

Figure 2.2 Sample System Specification File

the .MMAP directive or the declaration of boot memory also serves to signal the Cross-Software that the system in question is an ADSP-2101 architecture. If none of these indicators are present, the System Builder assumes an ADSP–2100 processor.

The .MMAP0 directive specifies the simulated state of the MMAP pin on the ADSP-2101 in this example system. Defining MMAP as 0 indicates that boot memory is to be loaded into the chip's internal program memory space, beginning at address 0.

The .SEG directive declares the system's physical memory segments and their characteristics. In this example, the segments declared comprise the full on-chip and off-chip program and data memory configuration of the ADSP-2101. Many applications, however, do not require this much memory space.

Boot_mem identifies a 2K-word space for one page of external boot memory.

Int_pm declares the 2K-word on-chip program memory space beginning at address 0. In the ADSP-2101 this memory can always hold both code and data and should be explicitly declared as such as in this example. *Ext_pm* declares a 14K-word space for external program code and data storage beginning at address 2048, after the on-chip memory.

Ext_dm declares a 14K-word space for external data storage beginning at address 0. *Int_dm* declares the 1K-word internal data memory space beginning at address 14336. This corresponds exactly to the on-chip data memory of the ADSP-2101 which is available for general system use. The 1K of on-chip memory above this is reserved for processor use and should not be declared.

The memory segments can be declared in any order.

The last statement in a system specification file is the .ENDSYS directive. The System Builder stops processing when it encounters the .ENDSYS directive.

2 System Builder

2.5 SYSTEM BUILDER DIRECTIVES

This section describes each System Builder directive and its syntax.

2.5.1 .SYSTEM Directive

The .SYSTEM directive must be the first statement in the System Specification source file. The identifier name given as its argument is the name of the system displayed in the Simulator.

The .SYSTEM directive has the form:

.SYSTEM *system_name;*

2.5.2 .ENDSYS Directive

The .ENDSYS directive must be the last statement in the file. The System Builder processing terminates at the .ENDSYS directive statement.

The .ENDSYS directive has the form:

.ENDSYS;

2.5.3 .ADSP2101 Directive

This directive identifies the processor. Its use is mandatory to clearly differentiate between ADSP-2100-based and ADSP-2101-based systems. If the directive is not present, the Cross-Software system assumes that the processor is an ADSP-2100.

2.5.4 .CONST Directive

The .CONST directive defines System Builder constants. Once you declare a constant, you may use it in place of its numeric value. This symbolic constant is recognized only by the System Builder, however the definition is not carried over to the Assembler or Simulator.

The .CONST directive has the form:

.CONST *constant_name* = constant or expression, ... ;

A single .CONST directive may declare one or several constants, separated by commas.

If you wished to define the value 15 for the term *taps*, for example, the directive would be as follows:

```
.CONST taps = 15;
```

The above example system does not declare any constants.

2.5.5 .PORT Directive

The .PORT directive declares a memory-mapped parallel I/O port. Ports can be placed in either data or program memory, and must be declared in one or the other. The directive takes the absolute physical address of the I/O port as a modifier, and the symbolic name of the port as an argument.

The .PORT directive has the form:

.PORT/qualifier ... *port_name*;

There are two required qualifiers:

PM or DM	(in which memory space)
ABS=address	(absolute address (constant))

The port address is specified by a constant; *port_name* is an identifier.

For example,

```
.PORT/DM/ABS=0x0400 ad_sample;
```

declares a port identified as *ad_sample* located at absolute data memory address 1024 (decimal). Assembler references to this same symbolic name are correctly interpreted by the Linker, using the .ACH file information.

This ADSP-2101 example system does not have any I/O ports declared.

2.5.6 .MMAP Directive

The .MMAP directive specifies the state of the MMAP pin on the ADSP-2101. It has the form .MMAP0 (MMAP pin held LO) or .MMAP1 (MMAP pin held HI).

If .MMAP0 is used, boot loading takes place and on-chip program memory begins at address zero. If .MMAP1 is used, no boot loading takes place and on-chip program memory is mapped at the top of the program memory space.

When this directive is omitted, the default is to .MMAP0.

See the *ADSP-2101 User's Manual* for further information.

2.5.7 .SEG Directive

The .SEG directive names a specific section of physical memory in the target system, and describes its attributes. In effect, the default memory map from the perspective of the System Builder is no memory at all. Until you declare and define a memory segment it does not exist.

The .SEG directive has the form:

.SEG/qualifier ... *seg_name[length]*;

The following qualifiers are mandatory:

PM or DM or BOOT=0, 1, 2, 3, 4, 5, 6, 7	(in which memory space)
RAM or ROM	(memory type)

While the following are optional:

ABS=address	(absolute start address (constant))
DATA or CODE or DATA/CODE	(what is stored in segment)

Seg_name is an identifier; *length*, which must be a constant or expression enclosed in brackets, is the number of words in the segment.

The .SEG directive declares three types of memory segments: program memory (PM), data memory (DM) and boot memory (BOOT). Qualifiers may specify the absolute start address of the segment, the physical memory type (RAM or ROM) and what is stored (DATA and/or CODE).

PM memory segments can be either CODE only, DATA only, or both CODE and DATA (defaults to CODE). For a PM segment that contains code and data, both modifiers must be used in the directive statement. The processor requires that any data access to PM must be made to sections

with the DATA attribute. If a system requires that executable code be read or written by the processor, these sections should be declared with both CODE and DATA attributes.

DM memory segments must be DATA only. Therefore, the /DATA modifier can be omitted. An error is generated if a DM segment is assigned the CODE attribute.

BOOT memory segments may be either ROM- or RAM-type; in most systems, however, the boot memory chips are PROM and all BOOT segments are specified as ROM-type. Boot memory always defaults to both CODE and DATA; the CODE and DATA attributes are unnecessary. The BOOT modifier always specifies the page number, for example, BOOT=0. A system may have up to 8 boot pages, with page numbers from 0 to 7. Each page can hold up to 2K words of code and data. The System Builder knows how long a page can be and the possible boundaries for each page; it ignores the ABS modifier for boot pages. An individual declaration must be made for each boot page required.

Memory segments are assigned symbolic names. In the Assembler you may locate individual code modules and data objects (buffers and variables) in segments by name. The Assembler accepts the segment references; the Linker resolves them using the .ACH file.

The length of the segment is specified by the bracketed expression, as in *somedata[1024]*. The unit is always words, either 16-bit data or 24-bit instructions. This means that data memory segment size in bytes is 2x the word count, program memory size in bytes is 3x the word count and boot memory size is 4x the word count. The latter reflects the padding of boot memory with an extraneous byte per instruction in order to place the beginning of every instruction on an even byte boundary.

The example

```
.SEG/BOOT=0/ROM boot_mem[2048];
```

declares the boot segment, *boot_mem*, which is physical memory type ROM, residing in boot page zero (corresponding automatically to absolute address 0). The length of the segment is 2048 words corresponding to one page of boot memory.

2 System Builder

The example

```
.CONST onchip_pm = 2048;
.SEG/PM/RAM/ABS=0/CODE/DATA  int_pm[onchip_pm];
```

declares a program memory segment called *int_pm*, which is memory type RAM at absolute location 0. This segment may hold both code and data. The length of the segment is 2048 words. This corresponds to the ADSP-2101 on-chip program memory space.

Assembler 3

3.1 INTRODUCTION

The ADSP-2101 Assembler translates source code modules into object code modules. You create a source code file (.DSP) using the ADSP-2101 assembly language and define variables, data buffers, and symbolic constants using assembler directives. Separately assembled modules are linked together to form an executable program.

Figure 3.1, on the next page, shows the Assembler input and output files. The ADSP-2101 Assembler reads the source code file (.DSP) and generates four output files with the same root name: an object file (.OBJ), a code file (.CDE), an initialization file (.INT), and a list file (.LST). The object file, code file and initialization files are passed to the Linker. The object file contains information on memory allocation and symbol declarations. The code file contains instruction opcodes with unresolved symbols marked. The initialization file contains initialization information for data buffers. The list file, which is optional, is for documentation.

Using assembly directives in the source code file, you can include other source code files and inform the Linker of initialization data files in the assembly process. The Assembler reads these files and processes them together with the original source file. There are two preprocessors of the Assembler, an ANSI-standard C language module and a standard preprocessor. The Assembler also supports a macro capability.

Check the system requirements in Appendix C, especially if you are running an IBM PC version of the Assembler.

3 Assembler

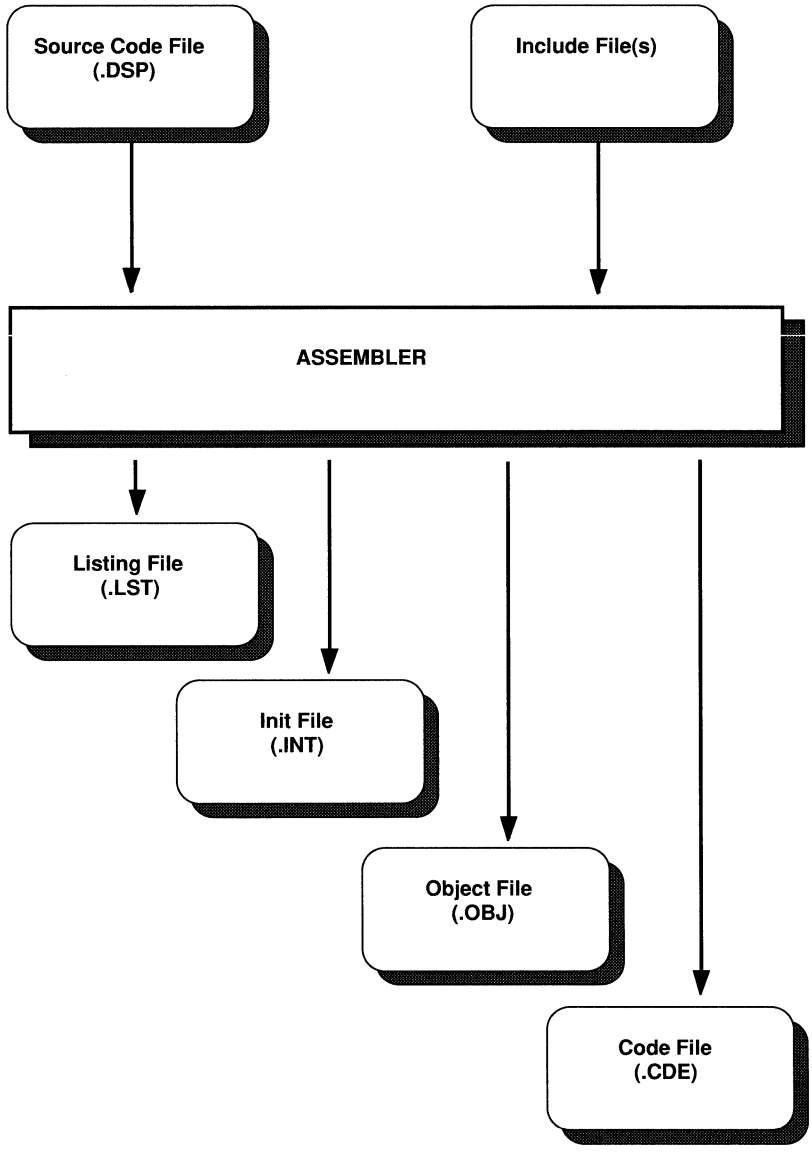


Figure 3.1 Assembler I/O

Assembler 3

3.2 ASSEMBLER MODULES

The Assembler consists of three modules:

C language preprocessor	actual filename: ASMPP
standard preprocessor	actual filename: ASM21
core assembler	actual filename: ASM2

Different combinations of the modules can be run using the Assembler switches detailed below. Invocation of the Assembler with no switches runs the standard preprocessor and core assembler only.

3.3 RUNNING THE ASSEMBLER

To invoke the Assembler from the host system, enter:

ASM21 *filename*[*.ext*] [–switch ...]

Filename[*.ext*] is the source code file. The filename extension is optional and defaults to .DSP. Other data and source code files are included in the assembly process using the directives .INIT and .INCLUDE (described later in this chapter).

3.3.1 Assembler Switches

The switches themselves are not case-sensitive, and multiple switches must be separated by spaces. The Assembler switches are listed below in Table 3.1; some require arguments as shown. To see this list on your display, invoke the Assembler with no filename or switches: ASM21.

Switch	*Result*
–cp	Runs C language preprocessor
–p	Runs standard preprocessor without core assembler
–d*variable*[=value]	Define *variable* for C preprocessor
–l	Creates .LST file
–m [number]	Macros expanded in .LST file, to depth of [number]
–i [number]	INCLUDE files expanded in .LST file, to depth of [number]
–s	No semantics checking
–c	Makes the Assembler case-sensitive

Table 3.1 Assembler Switches

3 Assembler

3.3.1.1 –cp Switch

Using the –cp switch runs the ANSI-standard C language preprocessor. This module of the Assembler allows the use of convenient C language directives in assembly code, if desired. The C preprocessor should only be used if C preprocessor directives or conditional constructs are present in the input assembly language file. These types of code are handled by the C preprocessor in the same fashion as a C compiler preprocessor. An intermediate file, *filename.CPP*, is deleted if the standard preprocessor runs without error. If an error does occur, the standard preprocessor halts execution prematurely and preserves the .CPP file.

3.3.1.2 –p Switch

The Assembler's standard preprocessor handles INCLUDE files, macro expansion, and the replacement of symbolic constants with their values, and produces a temporary .APP file which is used by the core assembler. Using the –p switch runs the preprocessor, prevents the core assembler from running, and preserves the .APP file. The .LST, .INT, .OBJ, and .CDE files are not created.

Note that the preprocessor module actually runs whether or not the –p switch is used, the switch merely determines if the core assembler is subsequently run, deleting the .APP file.

If you experience a problem using macros, you can turn on the –p switch and examine the .APP file to see if the macro invocations (calls) were correctly replaced with the macros' executable code. The .APP file is an ASCII file, although it contains some additional directives and control information.

Switch combination	*Module(s) run*	*File(s) preserved*
ASM21	preprocessor core assembler	.INT, .OBJ, .CDE, .LST (if –l switch used)
ASM21 –cp	C preprocessor preprocessor core assembler	.INT, .OBJ, .CDE, .LST (if –l switch used)
ASM21 –p	preprocessor	.APP
ASM21 –cp –p	C preprocessor preprocessor	.APP

Table 3.2 Preprocessor Switch Combinations

Figure 3.2 shows the flow of program control for the Assembler modules.

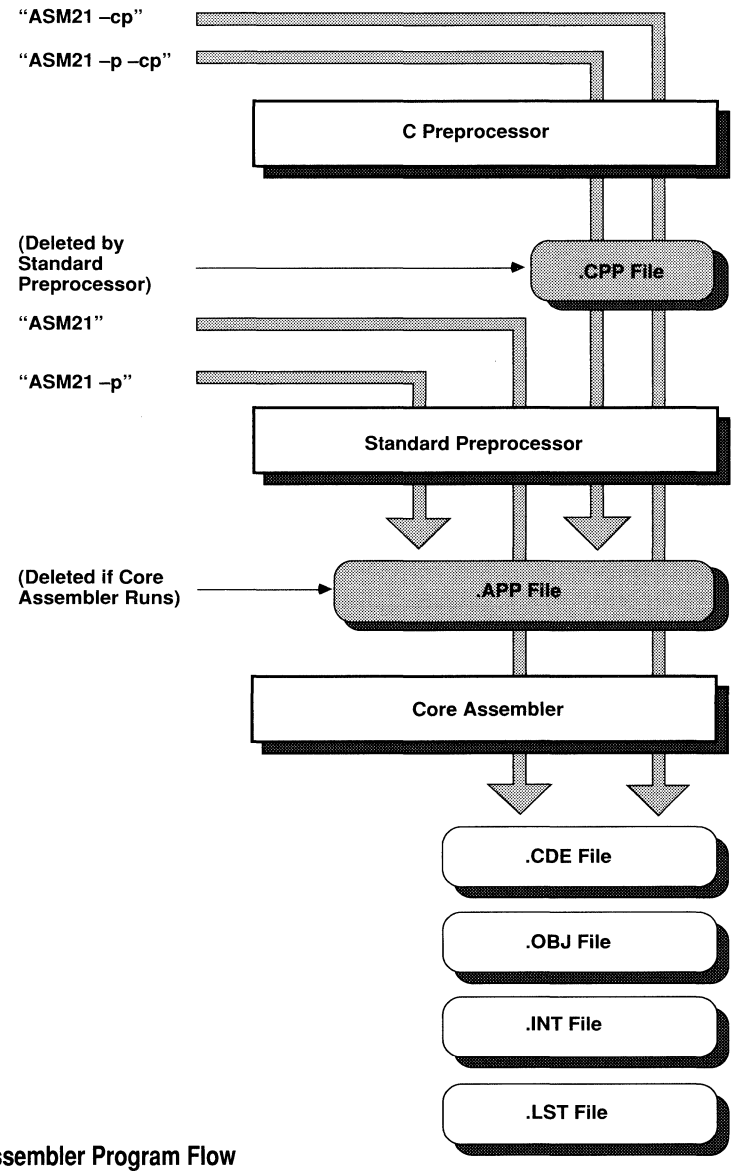


Figure 3.2 Assembler Program Flow

3 Assembler

3.3.1.3 –dvariable[=value] Switch

If a variable has been used in a C preprocessor directive in the input assembly language file it must be defined for the C preprocessor (which handles such directives for the Assembler). The variable can be any character string, and can be optionally set to a desired value which may be a character string or numerical value. Defining and/or giving a value to the variable allows the C preprocessor to evaluate a conditional statement dependent upon it.

A common use of this is to have a section of debug code written in the input file and to make its inclusion conditional. For example, place the debug code inside a conditional directive so that the code is assembled only if the variable *mydebug* is defined. The input file contains the following:

```
#ifdef mydebug
   ...
   debug assembly code
   ...
#endif
```

The Assembler must now be invoked as follows to assemble the debug code:

ASM21 *filename* –cp –d*mydebug*

3.3.1.4 –L Switch

The Assembler produces a listing file (.LST) if the –l switch is used. This file is described in the section "List File Format" later in this chapter.

3.3.1.5 –m [number] Switch

The listing file (.LST) does not normally display macros in expanded format; the –m switch expands the macros called in the file. Specifying a number determines the depth of nested macros to be expanded. For example, if number is chosen to be 3, macros invoked within other macros to a depth of 3 will be expanded. Choosing number is optional, and the default is to infinity (all nested macros expanded to infinite depth).

Examples:

–m 3

–m

3.3.1.6 –i [number] Switch

Using the –i switch causes the contents of files named with the .INCLUDE directive to be shown in the .LST file. Specifying number determines the depth of nested INCLUDE files to be shown. Giving a number is optional, and the default is to infinity (similar to –m switch). If the –i switch is not used, these directives remain in the form .INCLUDE *filename*.

3.3.1.7 –s Switch

The Assembler generates warning messages when multifunction instructions are not in the correct order. When you turn on the –s switch, the system does not check for the semantics (order) of a multifunction instruction. (In this mode, warning messages are not displayed on the screen.) For a description of multifunction instructions, refer to Chapter 9, Instruction Set Reference.

3.3.1.8 –c Switch

The default operation of the Assembler is to treat upper and lowercase letters as identical, as in previous releases. With this switch, the Assembler is made case-sensitive (similar to the C language environment); upper and lowercase versions of the same letter are treated as different characters. The –c switch supports the ADSP-2101 C Compiler.

3.4 LANGUAGE CONVENTIONS

This section describes the language conventions specific to the Assembler. See Chapter 1 for a complete discussion of general conventions including notation used in this manual, usable character set, symbols, identifiers, constants and expressions.

3.4.1 Binary Constants

Binary numbers are accepted only by the Assembler, and may not be used with any of the other Cross-Software tools. Binary numbers are specified with the prefix B#:

B#0111010001011111

Decimal, octal, and hexadecimal numbers are specified in source code in the normal fashion (as shown in Chapter 1).

3 Assembler

3.4.2 Symbols

Symbols are used in a source code program to represent various items. Symbols include identifiers and keywords.

3.4.2.1 Identifiers

Identifiers identify and name an assembly module, assembly values, data buffers and variables, I/O ports, macros, address locations and subroutines.

An identifier is a user-defined character string. The string may be of any length, but only the first 32 characters are significant. See Chapter 1 for a specification of the exact form of identifiers. As the default operation of the Assembler is case-insensitive, identifiers may be either upper or lower case (unless the –c switch is used).

The "pointer to" (^) and "length of" (%) operators are used with identifiers which label data buffers. ^*buffer_name* is evaluated by the Assembler as the base address of the buffer, and %*buffer_name* is evaluated as the number of words in the buffer.

3.4.2.2 Reserved Symbols (Keywords)

Symbol names in the source code file must be unique. Assembler-reserved symbols may not be used as identifiers. Because the Assembler is not case sensitive, both upper and lower case keywords are reserved. Table 3.3 lists the assembler keywords. Some of those listed correspond to ADSP-2101 features which are not visible to users. Avoid them because their use may cause errors.

ABS	DM	INCLUDE	MR0	RTS
AC	DO	INIT	MR1	RX0
AF	EMODE	JUMP	MR2	RX1
ALT_REG	ENA	L0	MSTAT	SAT
AND	ENDMACRO	L1	MV	SB
AR	ENDMOD	L2	MX0	SEG
AR_SAT	ENTRY	L3	MX1	SEGMENT
ASHIFT	EQ	L4	MY0	SET
ASTAT	EXP	L5	MY1	SHIFT
AUX	EXPADJ	L6	NAME	SI
AV	EXTERNAL	L7	NE	SR
AV_LATCH	FOREVER	LE	NEG	SR0
AX0	FLAG_IN	LOCAL	NEWPAGE	SR1
AX1	FLAG_OUT	LOOP	NOP	SS
AY0	GE	LSHIFT	NORM	SSTAT
AY1	GLOBAL	LT	NOT	STATIC
BIT_REV	GT	M0	OR	STS
BM	I0	M1	PASS	SU
BY	I1	M2	PC	TEST
C	I2	M3	PM	TIMER
CACHE	I3	M4	POP	TOGGLE
CALL	I4	M5	PORT	TOPOFPCSTACK
CE	I5	M6	POS	TRAP
CIRC	I6	M7	PRI	TRUE
CLR	I7	MACRO	PUSH	TX1
CLEAR	ICTRL	MF	RAM	TX0
CNTR	IDLE	M_MODE	REGBANK	UNTIL
CONST	IF	GO_MODE	RESET	US
DIS	IFC	MODIFY	RND	UU
DIVS	IMASK	MODULE	ROM	VAR
DIVQ		MR	RTI	XOR

Table 3.3 Assembler-Reserved Symbols/Keywords

3.4.3 Comments

You may insert comments anywhere in a source code file, enclosed by braces, { }. The Assembler treats all comments as "white space" and ignores them.

3.5 PROGRAM STRUCTURE

The basic unit of an ADSP-2101 program is the module. Modules are defined as:

.MODULE[/qualifiers] *module_name*;

statement; (may be any of
...
...

- [*label*:] instruction
- directive
- macro invocation)

.ENDMOD;

Each element of the module must end with a semicolon. Statements can be either an instruction, assembler directive, or macro call. Giving an instruction a label is optional. The .MODULE and .ENDMOD directives are defined in the section "Assembler Directives."

Chapter 9, Instruction Set Reference, defines the ADSP-2101 instructions. The "Macros" section in this chapter describes macro definition and invocation.

3.5.1 Source Code File Restrictions

Individual lines must be no more than 200 characters in length.

3.6 ASSEMBLER DIRECTIVES

Assembler directives are instructions that control the assembly process. They do not produce opcodes. In the source file, an assembler directive statement starts with a period and ends with a semicolon. An assembler directive may take modifiers and arguments, as specified in each of the following sections.

3.6.1 .MODULE Directive

The .MODULE directive defines the start of an assembly module and is the first statement. The default memory type is assumed to be RAM if not specified. The ABS modifier, if present, specifies the start address of the code segment.

The .MODULE directive has the form:

.MODULE[/qualifier ...] *module_name;*

Qualifiers consist of any of the following:

RAM or ROM
ABS = absolute start address
BOOT = 0, 1, 2, 3, 4, 5, 6, or 7
SEG = memory segment name defined in System Builder

The module qualifiers determine the location of the module in memory. Memory type can be specified as RAM or ROM, followed by the start address and/or a physical segment in memory defined in the System Builder. (The start address is a constant.)

There may be up to 8 boot pages of 2K length each. The BOOT qualifier can be specified as boot page 0 through 7, and multiple pages may be listed for one module (i.e. .MODULE /BOOT=0/BOOT=2). You must use this qualifier in order to have your bootable code located in the boot PROMs by the Linker and PROM Splitter. The Linker generates memory image files for an ADSP-2101 system, and only creates such a file for boot memory if this qualifier is used.

The memory type qualifier does not refer to the boot memory itself; it classifies the type of memory from which the code is executed. Boot memory merely stores the code until it is booted into the chip. Any module which is declared as bootable (with the BOOT qualifier) should in most cases be declared in RAM-type memory, because it is executed from the chip's internal 2K of program memory, which is RAM.

The BOOT qualifier also applies to all .VAR data buffer declarations within a module– remember that boot memory (and program memory in general) can contain both code and data.

The Assembler does not deal with boot memory as a separate memory space. The BOOT qualifiers for modules are passed on to the Linker to be acted upon. The crucial concept of a system with boot memory is the distinction between what is accomplished when running the Linker (locating objects in memory space), and what happens during run-time (program execution).

When you choose specifications and qualifiers for code modules and data buffers, these attributes apply to the run-time characteristics of the structures. Booted code is run from the ADSP-2101's internal program memory, when both the code and processor deal only with run-time program and data memory. When configuring the memory map of your system, you should think only in terms of program and data memory.

The example that follows defines the module *main_routine*, which is located at execution-time in RAM at address 0 (on-chip). The code is stored on boot page 0.

```
.MODULE/RAM/ABS=0/BOOT=0 main_routine;
```

The next example defines the module *filter_routine*, located in a memory segment named *fir* (as defined in a System Builder output .SYS file), which is specified as ROM.

```
.MODULE/ROM/SEG=fir filter_routine;
```

If you use the SEG qualifier and specify an address (ABS =) that is not the correct address for that segment, you receive an error message when the Linker is run.

3.6.2 .ENDMOD Directive

This directive has the form:

.ENDMOD;

The .ENDMOD directive is the last statement in a source code file. The assembly process terminates when the Assembler reads the .ENDMOD directive.

3.6.3 .VAR Directive

The .VAR directive declares data buffers. You must declare all buffers with the directive prior to any use of or reference to them. The default declaration ,with no qualifiers or length specified, is a relocatable buffer of length one (a variable) in data memory RAM.

The .VAR directive has the form:

.VAR[/qualifier ...] *buffer_name*[length], ... ;

One .VAR directive can have an unlimited number of declarations, each separated by commas, up to the maximum number of characters that can be processed. Specification of length is optional, with default to one (a single word variable).

Qualifiers consist of any of the following:

PM or DM
RAM or ROM
CIRC
ABS = absolute address
SEG = memory segment name defined in System Builder
STATIC

The following is an example variable declaration:

```
.VAR/DM/RAM/ABS=0x10F   seed;
```

This statement declares a one word variable called *seed* in data memory RAM, at hexadecimal address 10F.

The following is an example buffer declaration:

```
.VAR/PM/RAM/SEG=pmdata   coefficients[10];
```

Here a buffer is declared in program memory RAM, in a segment called *pmdata* which has been declared in the System Builder. The buffer name is *coefficients* and it has a length of 10. Note that the length, which may be a constant or expression, must be placed inside brackets: *coefficients*[10].

3 Assembler

In this manual's notation brackets are typically used to indicate a specification which is optional. .VAR, .INIT, and .INCLUDE are the only instances of Assembler syntax where brackets or angle brackets are required.

Data buffers are placed in either program memory (PM) or data memory (DM), with default to DM. The memory type qualifier specifies the type of memory: RAM or ROM. This modifier defaults to RAM for both DM and PM.

The buffer type defaults to linear unless you explicitly specify the circular attribute with the /CIRC qualifier.

The example that follows declares a circular buffer whose length is the value of the constant *taps*.

```
.VAR/DM/CIRC data_buffer[taps];
```

The /ABS qualifier specifies the start address of the data buffer. If you omit this qualifier, the buffer defaults to a relocatable buffer.

The /SEG qualifier specifies a segment in memory. If you specify a segment in memory and an address and the locations conflict, the Linker displays an error message.

The /STATIC qualifier is given to a data buffer whose contents must be preserved during software-controlled rebooting. This qualifier instructs the Linker to prevent the buffer from being overwritten by a newly-booted page. STATIC buffers are placed in memory by the Linker such that they are protected from being overwritten in multiple boot page systems. For additional information on the /STATIC qualifier and multiple boot page systems, refer to Appendix E.

If the buffer is to be initialized with data, the declaration and initialization must occur in the same module.

The .VAR directive takes an unlimited number of user-defined data variables or buffers as arguments, each separated by a comma. When you declare variables or buffers together, the Linker places them in contiguous memory segments. The length of a circular buffer is the sum of the lengths of all buffers declared in the same .VAR statement with the /CIRC qualifier.

3.6.3.1 More On Circular Buffers

Circular buffers (of any length) can only be placed at certain memory boundaries, depending on the length of the buffer. Unless you explicitly place buffers in memory, the Linker does it for you. Refer to Chapter 4, Linker, and the *ADSP-2101 User's Manual*, under "Data Structures," for additional information.

The following is an example of one circular buffer of length five (three bits required to represent), which would be located by the Linker at an address that is a multiple of eight (has three LSBs equal to zero):

```
.VAR/CIRC aa[5];
```

This example declares one circular buffer:

```
.VAR/CIRC aa[5], bb[5], cc[5];
```

Because three buffers are defined in a single .VAR declaration, this directive allocates one fifteen word circular buffer in memory. Since fifteen requires four bits to represent, the buffer is located at a base address which is a multiple of sixteen. The address of *aa* is the base address. The address of *bb* is the base plus five and the address of *cc* is the base plus ten. The three buffers named (*aa, bb, cc*) can all be individually referenced as simple buffers, but there is only one circular buffer. This is shown graphically in part A of Figure 3.3, on the following page.

The following example uses three separate directive statements to declare three separate circular buffers:

```
.VAR/CIRC aa[5];
.VAR/CIRC bb[5];
.VAR/CIRC cc[5];
```

Each of these buffers requires only three bits to represent and each is located at a different address which is a multiple of eight. Because you declare them separately, they are not necessarily contiguous. Part B of Figure 3.3 shows this.

3 Assembler

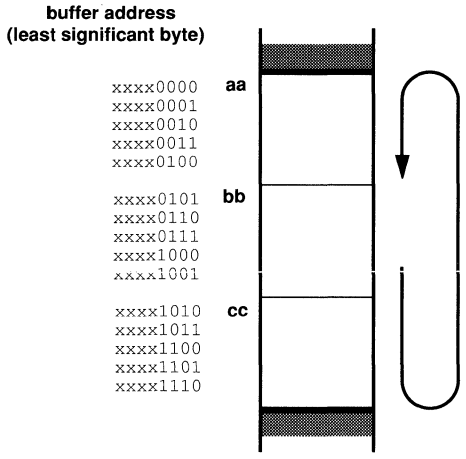


Figure 3.3A Circular Buffers

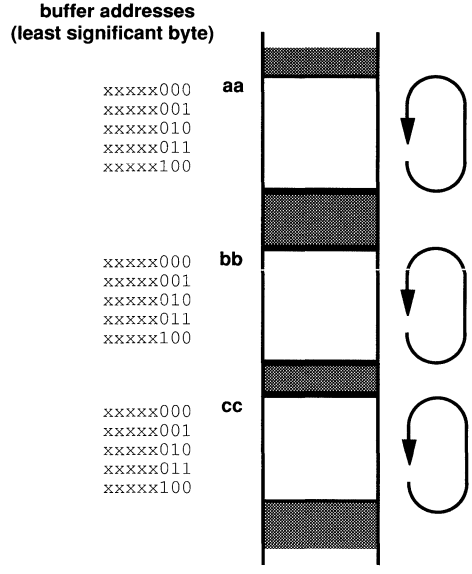


Figure 3.3B Circular Buffers

The following example creates the structure for a sine/cosine lookup table:

```
.VAR/CIRC sin[256], cos[768];
```

This example declares one circular buffer with a length of 1024, placed at an address boundary which is a multiple of 1024 (has ten LSBs equal to zero). In a program, you can initialize index registers (I registers) and buffer length registers (L registers) with this statement:

```
I0 =   ^cos; {^ is the "address pointer" operator}
L0 =   1024;
I1 =   ^sin;
L1 =   1024;
```

The address pointer operator ^ instructs the Assembler to determine the address of the memory label it is used with. In the above example the DAG index registers I0 and I1 are loaded with the addresses equated to *cos* and *sin*.

3.6.4 .INIT Directive

The .INIT directive initializes a declared variable or all or part of a data buffer (in either DM or PM). The buffer is initialized with the value(s) listed or those contained in an external file.

The .INIT directive takes the following form:

.INIT *buffer_name*: constant or expression, ... ,

^*other_buffer*[offset] or %*other_buffer*[offset], ... ,

<*filename*>;

Any combination of the three forms of initialization values shown above may be used, separated by commas.

An offset from the base address within a buffer may be specified as the destination location (or source address, as above):

.INIT *buffer_name*[offset]: ... ; offset= constant or expression

The initialization data is either listed in the .INIT directive statement or contained in a data file read by the Linker. Appendix B defines the external data file format. You should initialize all variables and buffers in the same module in which they are first declared.

3 Assembler

.INIT recognizes the "pointer to" (^) and "length of" (%) operators.

Examples:

`.INIT seed: 0x3FFF;`	Initialize variable *seed* with a constant hex value.
`.INIT seed_values: 1,2,3,5,7;`	Initialize the five-word buffer *seed_values* with the listed values.
`.INIT lookup_table: ^sin;`	Set variable *lookup_table* to point to the base address of buffer *sin*.
`.INIT cos: <cosines.dat>;`	Initialize the buffer *cos* with the contents of the external file *cosines.dat*, which is read by the Linker. (The use of angle brackets here is mandatory.)
`.INIT coefficients[5]: 2;`	Initialize the sixth element of the buffer *coefficients* with the value 2.
`.INIT buf1: 9,5,1,<sample.dat>;`	Initialize *buf1* with three constants and the contents of the file *sample.dat*.

Initializing from external files is helpful for setting buffer contents with data produced by high-level programs, such as filter coefficient or FFT twiddle factor generation routines. If you use external files, you do not need to initialize data at assembly time. The Assembler establishes a pointer to the external data files, and the data is incorporated when the Linker is run. Consequently, when changes are made in external data files, re-linking updates the program. There is no need to re-assemble.

The .INIT directive causes the Linker to initialize buffers with the specified data in the (.EXE) memory image file. This file can be used to load the initialized buffers in three cases: (1) for any external program or data memory which is ROM-type and is burned (by means of the PROM Splitter output files), (2) for any internal program memory buffers which are booted from boot PROMs, and (3) for debugging with the Simulator and Emulator.

3.6.5 .CONST Directive

The .CONST directive declares symbolic constants. You can use symbolic constants wherever you use numeric values.

The .CONST directive has the form:

.CONST constant_name = constant or expression, ... ;

One .CONST directive can have an unlimited number of assignment statements, each separated by commas, up to the maximum number of characters that can be processed.

Example:

```
.CONST taps=15, taps_less_one=14;
```

This defines two constants, equal to the numeric values shown.

3.6.6 .PORT Directive

The .PORT directive declares a memory-mapped I/O port in data or program memory. The argument for this directive is a symbolic port name. The name must be the name of a port declared in the Architecture Description file.

The .PORT directive has the form:

.PORT *port_name;*

When you reference ports, use the GLOBAL attribute in the module where you first declare the port and the EXTERNAL attribute in other modules. The Linker reads all information about this port from the Architecture Description file (.ACH) and resolves all references to it.

The following example identifies the port *ad_sample* which has been previously declared as a specific memory location in the System Builder:

```
.PORT ad_sample;
```

3.6.7 .INCLUDE Directive

The .INCLUDE directive is used to include another source file in the file being assembled. The Assembler reads the include file when it encounters the .INCLUDE statement. The Assembler processes the included file as if

it were part of the original source file. When the Assembler comes to the end-of-file of the included file, it returns to the original source file and continues reading and processing.

The .INCLUDE directive has the form:

.INCLUDE *<filename>*;

Source files specified by the .INCLUDE directive can have .INCLUDE statements within them (nesting of include files is limited only by memory).

The .INCLUDE directive supports modular programming. For example, in many cases it is useful to develop a library of subroutines or macros which are shared between different programs. Rather than rewriting these routines for each program, you can incorporate a macro library into the source code file using the .INCLUDE directive.

Example:

```
.INCLUDE <macro_lib>;
```

Here the use of angle brackets is required.

Another way to place additional source files into the file being assembled is to use the *#include* C preprocessor directive. *#Include* may be used in source code rather than .INCLUDE; however, the Assembler's C preprocessor must be invoked in order to handle the directive.

3.6.8 Macros

This section defines macros and the .MACRO directive. Macro capability simplifies source code development by allowing frequently used instruction sequences to be inserted at the point of reference. Using the argument passing feature, a macro can be a general-purpose subroutine that is shared by different programs. The macro reduces duplication of programming effort.

3.6.8.1 Macro Definition

A macro is called by name and allows argument passing. Macro definitions have the form:

.MACRO *macro_name*(arguments);

statement; (may be any of
...
...

- [*label*:] instruction
- .LOCAL (local directive)
- directive (all others)
- macro invocation)

.ENDMACRO;

Macro statements can be any legal ADSP-2101 Assembler statement.

An alternative to using the .MACRO directive to create an assembly code macro is the *#define* C language directive. If *#define* is used for macro definition, the Assembler's C preprocessor must be run in order to process the directive.

3.6.8.2 .MACRO Directive

The .MACRO directive is the start of a section of code which is to be defined as a macro, and includes the macro's name and arguments. It has the form:

.MACRO *macro_name*(argument, ...);

Arguments, which are optional, take the form: %n n=0, 1, 2, ... , 9

For example:

```
.MACRO  memory_transf(%0,%1,%2,%3,%4);
```

Within the source code of the macro, the arguments are marked by the place holder *%n*, where *n* is a number assigned between 0 and 9. When the macro is called, the *%n* placeholders are replaced with the actual values passed. The number of arguments declared and the number of parameters passed when the macro is called must match. Note that the percentage sign is used in this context to identify the place holders, not as a "length of" operator.

When the macro is called, the parameters passed to the place holders may be anything shown in Table 3.4, below.

Legal Parameter	*Comments*
constant or expression	
identifier	May include reserved words except MACRO, ENDMACRO, CONST and INCLUDE.
^identifier	"^%n" is not allowed within macro
%identifier	"%%n" is not allowed within macro

Table 3.4 Arguments/Parameters Legally Passed to Macros

The "pointer to" and "length of" operators (^ and %) cannot be used with argument place holders within the macro. However, a parameter passed when the macro is called may use these operators. For example, you could invoke the macro *read_data*(%0) and point to a buffer address with the parameter passed:

```
read_data(^input);
```

To avoid duplicate label errors when a macro is referenced multiple times within a module, a label in the macro code must be declared a local label with the .LOCAL directive; see below.

Macro nesting is limited only by memory at assemble time.

3.6.8.3 .ENDMACRO Directive

The .ENDMACRO directive has the form:

.ENDMACRO;

The .ENDMACRO directive terminates a macro definition portion of code.

3.6.8.4 Macro Example

A macro example is shown in Figure 3.4. In this example, the macro *memory_transf* is a general purpose memory transfer routine which can transfer data buffers from one memory area (program or data) to the other. This example passes five arguments (%0, %1, %2, %3, %4). PM and DM references can be passed.

```
{MACRO declaration}
.MACRO memory_transf(%0,%1,%2,%3,%4);   {pass five arguments}
.LOCAL transf;
   I4=%0;               {set I4 to source start address}
   I5=%1;               {set I5 to destination start address}
   M4=1;                {set pointer update to single increment}
   CNTR=%2;             {set length of buffer}
   DO transf UNTIL CE;  {transfer data}
   SI=%3(I4,M4);        {transfer from type %3 memory}
transf: %4(I5,M4)=SI;   {transfer to type %4 memory}
.ENDMACRO;

{MACRO invocation}
memory_transf(^coeff_table, ^buffer, buff_length, PM, DM);
```

Figure 3.4 Macro Example

3.6.9 .LOCAL Directive

The .LOCAL directive has the form:

.LOCAL *local_label, ...* ;

The .LOCAL directive is used only within a macro definition section of code. (See Figure 3.4.) The .LOCAL directive tells the Assembler to create a unique label for *local_label* at each invocation of the macro. This avoids duplicate label errors in cases where macros are called multiple times within a module.

The Assembler appends a number to each local label; this can be seen in the Simulator, or in the .LST file if macros are expanded.

Example:

```
.LOCAL transf;
```

3.6.10 .EXTERNAL Directive

The .EXTERNAL directive assigns the EXTERNAL attribute to identifiers. This attribute is typically given to variables, buffers, ports, and program memory labels declared in other assembly modules. Those symbols in other modules can only be referenced if they are assigned the EXTERNAL attribute in the referencing module and the GLOBAL or ENTRY attribute in the module where they are actually declared.

This directive has the form:

.EXTERNAL *external_symbol, ... ;*

Example:

```
.EXTERNAL fir_start;   {entry label in different module}
```

3.6.11 .GLOBAL Directive

The .GLOBAL directive assigns the GLOBAL attribute to variables, buffers, and ports. Only such identifiers declared (with .VAR or .PORT) as global may be referenced in other modules.

The .GLOBAL directive has the form:

.GLOBAL *internal_symbol, ... ;*

A variable, buffer, or port that is declared within a module can be referenced only by that module unless you explicitly specify it as global. For program labels which you intend to reference in other modules, you should use the .ENTRY directive rather than the .GLOBAL directive. Example:

```
.GLOBAL seed;
```

Other modules are able to refer to global identifiers by declaring those symbols as EXTERNAL.

3.6.12 .ENTRY Directive

The .ENTRY directive assigns the ENTRY attribute to program labels. This makes the label visible to other modules for use in subroutine calls or inter-module jumps.

The .ENTRY directive has the form:

.ENTRY *program_label, ... ;*

Example:

```
.ENTRY fir_start; {make label visible outside module}
```

3.7 PROGRAM EXAMPLE

Figures 3.5 through 3.7 illustrate a sample source code program, an interrupt service subroutine, and an include file for the ADSP-2101. In this example the module *main_routine* is the main program and *fir_routine* is the subroutine. These modules are linked together to form a complete program.

There are six possible interrupt sources for the processor plus the restart vector at address 0. Each has four locations associated with it. As described in the *ADSP-2101 User's Manual*, the first 28 addresses in program memory contain the restart and interrupt vectors (0x0000 – 0x001B). The 29th PM address (0x001C) holds the first program instruction. Since *main_routine* is declared at absolute address zero, the first 28 instructions are placed in the interrupt vector locations. Because this example uses only the restart (0x0000) vector and SPORT0 Receive (0x000C) interrupt, the remaining instructions are simply returns (RTI).

The .VAR directive defines two circular buffers in on-chip memory: one in data memory RAM used to hold a delay line of samples and one in program memory RAM used to store coefficients for the filter. *Data_buffer* and *coefficient* are declared as GLOBAL buffers in *main_routine*, while *fir_routine* declares them as EXTERNAL. The address label, *fir_start* , is declared as ENTRY in *fir_routine* and can be referenced by *main_routine*, which declares it as EXTERNAL.

This sample program, which is also described in the *ADSP-2101 User's Manual*, implements a FIR filter routine and has several features worth noting. After declaring the include file and memory buffers and

performing initialization, *main_routine* jumps to location *restarter*. Here the data and coefficient buffers are cleared and the data memory-mapped control registers of the ADSP-2101 are set up. The functions selected include SPORT0 timing specification, u-law companding, and 8-bit data words. SPORT0 interrupt is then enabled and the processor loops on the IDLE instruction until the interrupt from SPORT0 is received. The filter is thus interrupt-driven. When the interrupt occurs, program control shifts to the subroutine by jumping to location *fir_start*.

All further activity takes place in the interrupt service routine, Figure 3.6. After the return from interrupt, execution resumes at the WAIT loop.

```
{ADSP-2101  FIR Filter program
Serial port 0 used for I/O
Internally generated serial clock
12.288 MHz clock rate gives 8000 Hz sampling rate}

.MODULE/RAM/ABS=0/BOOT=0 main_routine;    {program loaded from BOOT EPROM, MMAP=0}
.INCLUDE <const.h>;
.VAR/DM/RAM/ABS=0x3800/CIRC data_buffer[taps];  {data values}
.VAR/PM/RAM/CIRC coefficient[taps];
.GLOBAL data_buffer, coefficient;
.EXTERNAL fir_start;
.INIT coefficient: <coeff.dat>;                 {initialize coeffs from external file}
              {code starts here}
              {load interrupt vector addresses}

              JUMP restarter; nop; nop; nop;    {restart interrupt}
              RTI; nop; nop; nop;               {sampling interrupt IRQ2}
              RTI; nop; nop; nop;               {SPORT0 transmit int}
              JUMP fir_start; nop; nop; nop;    {SPORT0 receive int}
              RTI; nop; nop; nop;               {SPORT1 transmit int}
              RTI; nop; nop; nop;               {SPORT1 receive int}
              RTI; nop; nop; nop;               {TIMER interrupt}

              {initializations}

restarter:    L0 = %data_buffer;                {setup circular buffer length}
              L4 = %coefficient;                {setup circular buffer length}
              M0 = 1;                           {modify=1 for increment
              M4 = 1;                           through buffers}
              I0 = ^data_buffer;                {point to data start}
              I4 = ^coefficient;                {point to coeff start}
```

```
                CNTR = %data_buffer;              {setup loop counter}
                DO clear_buffer UNTIL CE;
clear_buffer:   DM(I0,M0)=0;              {clear data buffer}

                I1 = 0x3FEF;              {point to last DM control register for
                                          initialization}
                DM(I1,M0)=0x0000;         {SPORT1 AUTOBUFF disabled}
                DM(I1,M0)=0x0000;         {SPORT1 timing not used}
                DM(I1,M0)=0x0000;         {SPORT1 timing not used}
                DM(I1,M0)=0x0000;         {SPORT1 CNTL disabled}
                DM(I1,M0)=0x0000;         {SPORT0 AUTOBUFF disabled}
                DM(I1,M0)=191;            {divide by 192 for 8KHz}
                DM(I1,M0)=0x0003;         {generate 1.536MHz serial clk}
                DM(I1,M0)=0x69B7;         {multichannel disabled}
                                          {int. gen serial clock}
                                          {receive frame sync required}
                                          {receive width 0}
                                          {transmit frame sync required}
                                          {transmit width 0}
                                          {int transmit frame sync enabled}
                                          {int receive frame sync enabled}
                                          {u-law companding}
                                          {8 bit words}
                DM(I1,M0)=0x0000;         {transmit multichannels}
                DM(I1,M0)=0x0000;
                DM(I1,M0)=0x0000;         {receive multichannels}
                DM(I1,M0)=0x0000;
                DM(I1,M0)=0x0000;         {timer not used, cleared}
                DM(I1,M0)=0x0000;
                DM(I1,M0)=0x0000;
                DM(I1,M0)=0x7000;         {external DM wait states}
                                          {0x3400 - 0x37FF 7 waits}
                                          {all else 0 waits}
                DM(I1,M0)=0x1000;         {SPORT0 enabled}
                                          {boot page 0, 0 PM waits}
                                          {0 boot waits}
                ICNTL = 0x00;
                IMASK = 0x0018;           {enable SPORT0 interrupt only}

WAIT:           IDLE;                     {wait for interrupt}
                JUMP WAIT;
.ENDMOD;
```

Figure 3.5 Main Routine Example

3 Assembler

```
.MODULE/RAM/BOOT=0 fir_routine;          {relocatable interrupt service routine module}
.INCLUDE <const.h>;                      {include constant declarations}
.ENTRY fir_start; {make label visible outside module}
.EXTERNAL data_buffer, coefficient;      {make global buffers visible to module}
                  {code}

FIR_START:        CNTR = taps-1;         {N-1 passes within DO loop}
                  SI = RX0;              {read from SPORT0}
                  DM(I0,M0) = SI;        {transfer data to buffer}
                  MR=0, MY0=PM(I4,M4), MX0=DM(I0,M0);
                                         {set up multiplier for loop}

                  DO CONVOLUTION UNTIL CE;      {CE = counter expired}
CONVOLUTION:      MR=MR+MX0*MY0(SS), MY0=PM(I4,M4), MX0=DM(I0,M0);
                                         {MAC these, fetch next}

                  MR=MR+MX0*MY0(RND);    {Nth pass with rounding}
                  IF MV SAT MR;          {saturate if overflowed}
                  TX0 = MR1;             {write to sport 0 transmit}
                  RTI;                   {return from interrupt}
.ENDMOD;
```

Figure 3.6 Interrupt Routine Example

```
.CONST taps = 15;
```

Figure 3.7 Include File, Constant Initialization

3.8 LIST FILE FORMAT

The List file (.LST) allows you to interpret the result of the assembly process. A fragment of a sample list file for the ADSP-2101 is shown in Figure 3.8.

The following information is found in the list file:

addr	The first column specifies offset from module base address in program memory.
inst	The second column contains the hexadecimal representation of the instruction loaded at that address (opcode). An appended "u" indicates that the opcode contains an undefined field.
source line	The source file line number read by the Assembler is listed in the third column.
instruction/directive	This field contains the source code, either Assembler directive or assembly language instruction.

```
Analog Devices Inc.              ADSP-210X Assembler Version 2.00
C:\2101_System\fir2101.app          Mon Oct 9 11:04:39 1989        Page 1
addr  inst    source line

                1    .MODULE/RAM/BOOT=0 FIR_ROUTINE;      {relocatable interrupt
                2                                          service routine module}
                3    .include "const.h";                  {include constant declarations}
                4    .ENTRY FIR_START;                    {make label visible outside
                5                                          module}
                6    .EXTERNAL DATA_BUFFER, COEFFICIENT;  {make global buffers visible
                7                                          to module}
                8                 {code}
                9
0000  3C00E5    10   FIR_START:  CNTR = 14;               {N-1 passes within DO loop}
0001  0D0388    11               SI = RX0;                {read from SPORT0}
0002  680080    12               DM(I0,M0) = SI;          {transfer data to buffer}
0003  E89800    13               MR=0, MY0=PM(I4,M4), MX0=DM(I0,M0);
                14                                         {set up multiplier for loop}
```

Figure 3.8 List File Example

Linker 4

4.1 INTRODUCTION

The ADSP-2101 Linker generates a complete executable program by linking together program modules which were assembled separately. It can search libraries, which are simply subdirectories, for subroutines to link. The output of the Linker is used by the Emulator, Simulator and PROM Splitter. Figure 4.1, on the following page, shows the files read and created by the Linker.

As shown in the previous chapter, the Assembler processes each source code module separately, producing an Object file (.OBJ), a Code file (.CDE) and an Initialization file (.INT), which contains information on the assembled code, source level declarations and initialization information. Initialization data files (.DAT) are created separately. Changes in initialization data only require relinking.

The Assembler output files (one set for each module to be linked), together with initialization data files and the Architecture Description file are used by the Linker. The Linker expects to find an Architecture Description file with the default name 210x.ACH unless you alter this name with a switch; the files to be linked must be specified in the invocation command or located in libraries to be searched.

The Linker creates one complete executable code file by resolving external references and assigning addresses to relocatable code and data spaces.

The Linker can generate three files. The Memory Image file (.EXE) is always created, and contains the actual program memory, data memory, and boot memory images after the linkage. This file is used by the Simulator and Emulator, and is also passed to the PROM Splitter to prepare a data file for a PROM burner. It has the default name 210x.EXE which can also be changed with a switch.

The optional map listing file (.MAP) assists you in interpreting the result of the linkage. This file is discussed in more detail later in this chapter.

4 Linker

The optional debug symbol table file (.SYM) lists all symbols encountered by the Linker, their absolute values and their scope of reference. This file is used by the Simulator and Emulator.

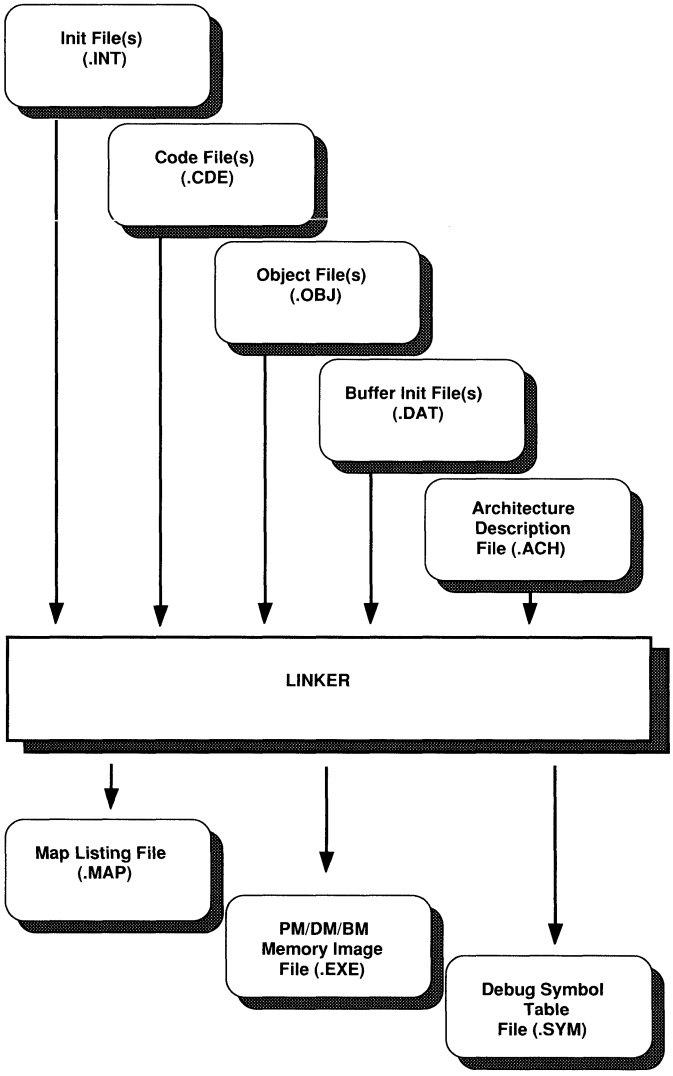


Figure 4.1 Linker I/O

Linker 4

The Linker can link together an unlimited number of modules and initialization data files. The initialization data files (.DAT) are not explicitly named in the invocation line because they are specified (with the .INIT directive) in the source code files. The data files are incorporated by the Linker. When changes are made in the data files, simply relink the modules to incorporate the new data file.

4.2 RUNNING THE LINKER

To invoke the Linker from the host system, the command form is:

LD21 *file1* [*file2* ...] [–switch ...]

or

LD21 –i *file_all* [–switch ...]

The –i switch causes the Linker to read the file *file_all* for a list of files to link. The file containing the list of files to link must be a simple text file with one pathname/file per line.

In the first form, you explicitly name all the files to be linked (separated by spaces). In both forms, the filename(s) must identify the Assembler output files (.CDE, .OBJ and .INT) without any extension. Modules to link are searched for in the current directory or in the pathname specified in the command line.

4 Linker

4.2.1 Linker Switches

The switch component of the invocation command can have any of the Linker switches (separated by spaces). The Linker switches are listed below in Table 4.1; some require arguments as shown. To see this list on your display, invoke the Linker with no files or switches: LD21.

Switch	*Result*
–a *archname*	Use *archname.ACH* Architecture Description file instead of default 210x.ACH
–c	Linker creates "top of RAM" symbol to locate the stack; this symbol is used by programs generated with the ADSP-2101 C Compiler (See Chapter 7)
–dryrun	Linker does not generate an .EXE file; quick test to check for link errors
–e *target*	Output files named *target.EXE*, instead of default 210x.EXE
–g	Linker generates a debugger symbol table, .SYM file
–i *file_all*	Links all files listed in text file *file_all*
–lib *directory; ...*	Directories listed are added to those found in ADIL environment setting for locating libraries; multiple directories are separated by commas in Unix systems or by semicolons in PC-DOS systems
–old	Not used (ADSP-2100 feature)
–p	Library subroutines are assigned to the boot pages that call them
–pmstack	Used with –c switch; moves "top of RAM" symbol to program memory
–s *stack_size*	Used with –c switch; specify a maximum size for stack
–x	Linker generates a .MAP file

Table 4.1 Linker Switches

4.2.1.1 –a archname & –e target Switches

These switches control the names of the files read and written by the Linker. The –a switch sets a new name for the Architecture Description file (read by the Linker), which defaults to 210x.ACH. The –e switch sets the name of the output files which otherwise default to 210x.EXE, 210x.SYM, and 210x.MAP.

4.2.1.2 –c Switch & ADIRTH Variable

This switch and environment variable are provided to support the linking of code modules generated by the ADSP-2101 C Compiler. You must invoke the Linker with the –c switch to link modules generated by the C Compiler. Using the switch causes two things to happen. First, the Linker creates the artificial symbol

`____top_of_ram` (four leading underscores)

which is assigned the value of the highest available address in data memory (or program memory, see the discussion of the –pmstack switch below). Second, the Linker searches for and links in the C run time header, which is an assembly language file (filename *run_hdr*) provided with the Cross-Software System. The `____top_of_ram` symbol is used by the run time header to locate and initialize the stack. See Chapter 7, C Compiler, for more information about the stack.

The environment variable ADIRTH must be equated to a pathname identifying the directory which contains the run time header. This path is searched by the Linker; the run time header must be located and linked because it is used when running compiled C code. The pathname is a function of your operating system, and is determined by where you store the *run_hdr* file.

To define the ADIRTH environment variable, execute a statement similar to the following examples, using the actual pathname for your system. The final slash must be present; do not include extra spaces.

IBM-PC Example:
SET ADIRTH=*root**subdir**subdir*\

Unix (Sun) Example:
setenv ADIRTH "/*root*/*subdir*/INCLUDE/"

4.2.1.3 –dryrun Switch

This switch causes the Linker not to produce the .EXE output file. It is provided so that you can check for the presence of any Linker error messages.

4 Linker

4.2.1.4 –g & –x Switches

These switches control the output of optional files. The –g switch causes the Linker to output the debug symbol table file, .SYM, which is not normally produced. The –x switch causes the Linker to produce the load map file, .MAP, also not normally produced. If the main filename has not changed since a previous linking operation, the previous .SYM and .MAP files are overwritten.

4.2.1.5 –i file_all Switch

This switch is used when the argument file contains a list of files to link. The Linker reads filenames from the text file, listing them one to a line, and locates the files to be linked.

4.2.1.6 –lib directories Switch & ADIL Variable

There are two paths the Linker searches for libraries of subroutines to link: one path specified by the ADIL environment variable and any listed in the *directory; ...* argument of the –lib switch.

The search pattern to find the subroutine files to link can be set using the ADIL environment variable. ADIL must be set to a pathname in your operating system leading to the subdirectory where the libraries are located. The Linker first searches the path specified by ADIL.

To define the ADIL environment variable, execute a statement similar to the following examples, using the actual pathnames for your system. Semi-colons separate individual search paths. The final slash must be present. Do not include extra spaces.

IBM-PC Example:
SET ADIL=*root**subdir**subdir*\;*root**nextsubdir**nextsubdir*\;

Unix (Sun) Example:
setenv ADIL "/*root*/*subdir*/INCLUDE/;/*root*/*nextsub*/INCLUDE/;"

The maximum number of directories that can be specified with ADIL is twenty. If ADIL has not been defined in the system environment and there is no –lib directories switch, the search terminates.

The second search path comes from the –lib switch itself. Here you specify a set of directories to search in the command line invoking the Linker. These are searched after ADIL has been searched. A convenient tool to use

in conjunction with the –lib switch is the DOS symbol for the current directory (the period). When invoked in the following fashion,

LD21 *file1 file2* ... –lib .

the Linker searches the entire current directory for subroutines to link.

4.2.1.7 –old Switch

This switch is an ADSP-2100 feature and should not be used with an ADSP-2101 system.

4.2.1.8 – p Switch

The –p switch is used when linking a program with library subroutines which are called on more than one page of boot memory. In such multiple boot page systems, a copy of a subroutine must be located on each page that calls it. This switch causes the Linker to place copies of subroutines on the boot pages where they are called.

The necessary set of subroutines is linked and incorporated into the boot memory portion of the .EXE file. When a page of code is booted under software control (during program execution), it then includes all the subroutines it uses. If the –p switch is not used, the Linker links the library routines but does not attach their memory images to specific boot pages.

Refer to Appendix E for further information on implementing multiple boot systems.

4.2.1.9 –pmstack Switch

This switch causes the top of RAM symbol and stack created by the run time header to be located in program memory. Without this switch, the stack is located in data memory by default. If your C program was compiled with the –pmstack switch for the C Compiler, it must also be linked with the –pmstack switch for the Linker.

4.2.1.10 –s stack_size Switch

Normally the stack (for compiled C code) has no limit on its size; it is allowed to grow larger (toward lower addresses) whenever new values are pushed onto it. By using the –s switch and specifying a number for *stack_size*, however, you can place a limit on how large the stack is allowed to grow. *Stack_size* must be an integer, and is evaluated by the Linker in units of words.

4 Linker

When this switch is used, the Linker creates the artificial symbol

`____top_of_ram` (four leading underscores)

which is given the following address value:

```
____top_of_ram = ____top_of_ram - stack_size
```

This symbol is used by the run time header to define and maintain the stack.

4.3 LINKER OPERATION

The Linker combines separately assembled source code modules and initialization data files into one executable program, using the hardware environment model specified in the Architecture Description file. The two main tasks are the allocation of memory and the resolution of symbols.

4.3.1 Memory Allocation

The Linker reads information from each code module and data buffer regarding the characteristics of the memory in which it is to be stored. Each module may list its memory attributes as RAM or ROM, and may specify an absolute start address (ABS= address), segment name (SEG= *name*) or boot page number (BOOT= page#). Data buffers are declared with the .VAR directive and may list their qualifiers as PM, DM, RAM, ROM, or CIRC and may also specify ABS or SEG. The Linker also receives information defining the target hardware system and available memory from the Architecture Description file (.ACH) produced by the System Builder.

The Linker assimilates this information and places the modules and buffers in memory by means of the memory image file (.EXE). A module or buffer must be placed in a portion of memory with the correct attributes. If no start address is chosen for an object, it is relocatable. The Linker decides upon a location for all such objects, with a bias toward placement in internal memory if possible.

There are three possible means of specifying a code module or data buffer's location in memory: (1) giving an absolute start address (ABS), with or without a segment name, (2) naming a System Builder-defined segment (SEG) in which to place the structure, or (3) listing neither. The first specification defines a non-relocatable object; the second is an object

which is relocatable within the named segment only; the third is an object which is completely relocatable.

The Linker places objects in memory in the following sequence.

1. Place all data buffers and modules with the ABS=address modifier (non-relocatable).

2. Place data buffers with the CIRC and SEG=name modifiers (relocatable within named segment).

3. Place all non-circular data buffers and modules with the SEG=name modifier (relocatable within named segment).

4. Place data buffers with the CIRC modifier (completely relocatable).

5. Place all remaining non-circular data buffers and modules (completely relocatable).

While non-circular, or linear, data buffers have no special placement constraints, circular buffers are handled differently. The Linker places circular buffers at 2^n modular boundaries (2, 4, 8, 16, etc.) corresponding to the buffer length. If a circular buffer has a length of 16, for example, it is placed at a base address which is a multiple of 16. If a circular buffer has a length of 13, it is similarly placed at the start of a 16-location block. See the discussion of circular buffers in Chapter 3, Assembler, for further information.

Circular buffer placement by the ADSP-2101 Linker is identical to that performed by the ADSP-2100 Linker except for the case where buffer length is equal to 2^n. The 2101 Linker places two separate 2^n-word circular buffers one right after the other in contiguous 2^n-word blocks. The 2100 Linker places two such buffers in memory with an unused 2^n-word block between them.

For example, the ADSP-2101 places two 1024-word circular buffers in contiguous blocks (address LSBs 0-1023 and 1024-2047). The ADSP-2100 places the two buffers with an unused 1024-word block between them (address LSBs 0-1023 and 2048-3071).

4 Linker

4.3.1.1 Boot Memory Allocation

A distinctive feature of memory allocation in an ADSP-2101 system is the use of boot memory. Any code module declared with the BOOT qualifier is placed in the boot memory space by the Linker. One or more boot page numbers are chosen for each bootable module. Each boot page can store a total of 2K words of code and data.

Boot memory should be thought of as a place to store your program until it is run. The crucial concept of a system with boot memory is the difference between what is accomplished when running the Linker (locating objects in memory space), and what happens during run-time (program execution).

When you choose specifications and qualifiers for code modules and data buffers, these attributes apply to the run-time characteristics of the structures. Booted code is run from the 2101's internal program memory, when both the code and processor deal only with run-time program and data memory. Thus when configuring the memory map of your system, you too should think only in terms of program and data memory.

The Assembler does not deal with boot memory as a separate memory space. It is the Linker which handles the logical interfacing of boot storage to run-time program memory. For systems with multiple boot pages, the Linker can handle placement of library subroutines and data buffers shared between pages. This is specified by means of the Linker's –p switch and the Assembler's STATIC buffer qualifier. See Appendix E.

4.3.2 Symbol Resolution

Any symbol (address label or data buffer) declared within a module can be used only by that module unless the .ENTRY or .GLOBAL directives are used. These directives expand the scope of reference of the symbols beyond the local module. For each symbol declared as .EXTERNAL, the Linker searches all other modules for occurrences of these symbols in an .ENTRY or .GLOBAL declaration. If this search fails, or if the search produces multiple matches, the Linker issues an error message. Once the allocation of memory segments is complete and all external references are resolved, the Linker assigns values to all symbols.

In resolving the symbols, the Linker creates a Debug Symbol Table (.SYM) file, which contains a list of all symbols encountered. The file gives information on which symbols can be referenced by each module. This file is used by the Simulator and Emulator to provide symbolic debugging. Appendix B describes this file in detail.

4.4 MAP LISTING FILE

The Map Listing file is generated to help you interpret the Linker result. The file provides information on:

- Symbols

A cross-reference listing of all symbols encountered, arranged by module. For each module a list is shown of the symbols referenced in that module, with the following information for each symbol: its absolute address, its length, the type of symbol (module, variable, or label) , and the type of memory (PM, DM, or BM).

- Memory segments

A map of physical memory segments declared for the system with the absolute address, length, and attributes of each. The information here reflects the content of the Architecture Description file.

- Boot memory & Run-time program memory

An address map of modules and data structures on each boot page, and the corresponding map of booted code in internal program memory ("bootable run-time program memory"). Information on PROM byte addresses and boot PROM sizes required is also provided.

- Fixed vs. Dynamic memory

Maps of fixed program memory, dynamic data memory, and fixed data memory. These maps include address, length, and attribute specifications.

- Error messages

Linker error messages (see Appendix F).

- Libraries

A list of libraries searched and used.

A sample Map Listing file is shown in Fig. 4.2, on the next page.

4 Linker

```
ADSP-210x Linker, version 2.00, copyright Analog Devices, Inc.
final (final.exe) mapped according to FIR_SYSTEM (sysb2101.ach)

xref for module: MAIN_ROUTINE     boot memory page(s) 0,

   MAIN_ROUTINE                pm 0:0000 [003B]    module(global)
   DATA_BUFFER                 dm 0:3800 [000F]    variable(global)
   COEFFICIENT                 pm 0:0040 [000F]    variable(global)
   RESTARTER                   pm 0:001C           label
   CLEAR_BUFFER                pm 0:0024           label
   WAIT                        pm 0:0039           label
   FIR_START                      0:004F [0000]    extern(FIR_ROUTINE)

xref for module: FIR_ROUTINE     boot memory page(s) 0,

   FIR_ROUTINE                 pm 0:004F [000A]    module(global)
   FIR_START                   pm 0:004F           label
   CONVOLUTION                 pm 0:0054           label
   COEFFICIENT                    0:0040 [000F]    extern(MAIN_ROUTINE)
   DATA_BUFFER                    0:3800 [000F]    extern(MAIN_ROUTINE)

210x memory per FIR_SYSTEM (sysb2101.ach):

   internal 2101 pm ram mapped to 0000 - 0800 (auto booted at reset)
   internal 2101 dm ram mapped to 3800 - 3BFF
   0000 - 07FF [  2048.] external bm rom code BOOT_MEM
   0000 - 07FF [  2048.] internal pm ram data/code INT_PM
   0800 - 3FFF [ 14336.] external pm ram data/code EXT_PM
   3800 - 3BFF [  1024.] internal dm ram data INT_DM
   0000 - 37FF [ 14336.] external dm ram data EXT_DM

boot memory and bootable run time program memory map:

   boot page 0 (auto boot)
   bm:0000-003A (x8rom:0000-00EB) pm:0000-003A [59.]  ram module  MAIN_ROUTINE of
                                                      MAIN_ROUTINE
   bm:0040-004E (x8rom:0100-013B) pm:0040-004E [15.]  ram circ variable COEFFICIENT of
                                                      MAIN_ROUTINE
   bm:004F-0058 (x8rom:013C-0163) pm:004F-0058 [10.]  ram module  FIR_ROUTINE of
                                                      FIR_ROUTINE

   8k of boot memory rom space required for this bootable run time map.
   Most convenient boot memory rom size is 8k bytes (64k bits).

fixed program memory map:

   fixed program memory rom:      0.
   fixed program memory ram:      0.

dynamic data memory map:

   boot page 0
   3800 - 380E [15.]          ram circ variable  DATA_BUFFER of MAIN_ROUTINE

fixed data memory map:

   fixed data memory rom:      0.
   fixed data memory ram:      0.

210Xlnk: final, 210x memory use:
   program memory rom: 0.; program memory ram: 0.;
   data memory rom: 0.; data memory ram: 0.
```

Figure 4.2 Map Listing File

Simulator Functions 5

5.1 INTRODUCTION

The ADSP-2101 Simulator is an interactive window-oriented software tool for instruction level simulation and debugging of your program. The Simulator configures itself according to your target system architecture as defined in the Architecture Description file (.ACH). This allows it to flag illegal operations such as reading from non-existent memory. Using the symbol table created by the Linker, the Simulator is able to provide a fully symbolic environment for simulation and debugging.

Briefly, the Simulator provides the following functions:

- Instruction level simulation of booting and execution
- Simulation of ports and SPORTs using host data files
- Simulation of internal and external interrupts
- Complete assembly and disassembly of the ADSP-2101 instruction set
- Multiple break conditions including break at address, break on condition, break on expression and break on address ranges
- Full view of all processor registers and the ability to directly change any register's contents interactively
- Profiling usage of portions of code during execution

Upon first booting the Simulator, you see the command window display as shown on the next page. From this window you open, configure and use all other features of the Simulator. Typing ^W (control-w) displays a menu of window commands including, for example, OPEN, which in turn displays a submenu of windows to be opened.

You can customize the contents and layout of many windows, the arrangement of multiple windows on the screen and the command strings used to invoke various Simulator functions. All customized settings can be stored in an external file and invoked automatically upon startup. For details, consult the next chapter, Custom Simulator Configurations.

5 Simulator Functions

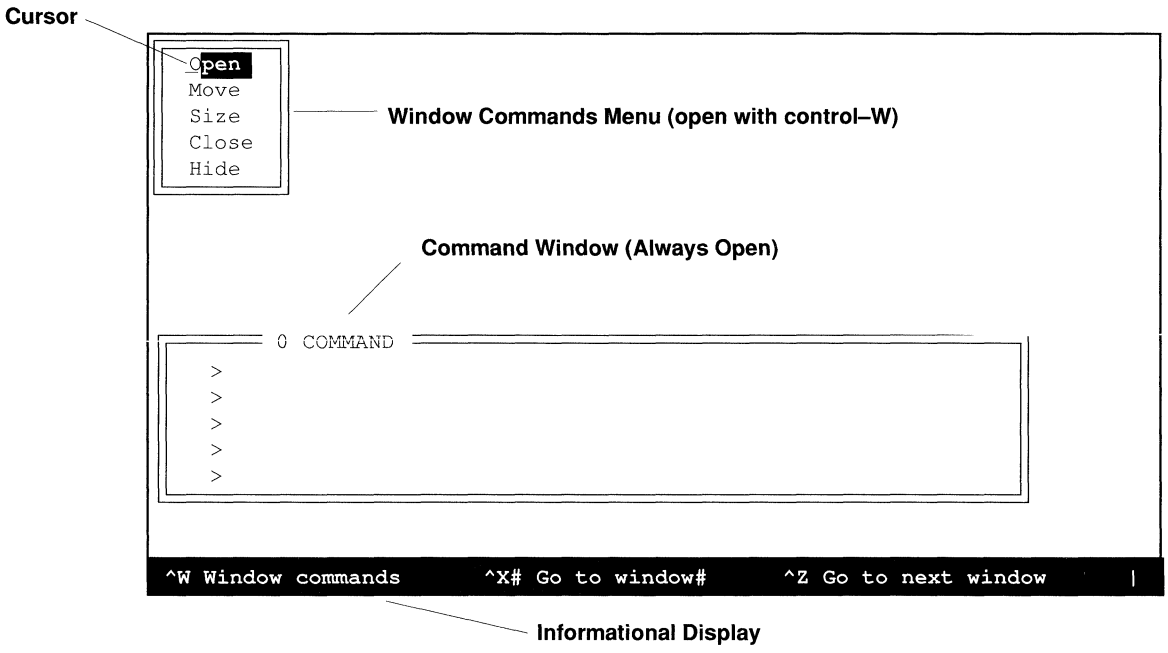


Figure 5.1 Initial Display & Window Commands Menu

5.2 GETTING STARTED

To get started with the Simulator, you need to prepare your linked program, install all Simulator program files and invoke the Simulator with the proper command line arguments.

5.2.1 Help Files & ADIDOC Variable

In order for the Simulator help files to be accessible, the following condition must be met:

- The path (subdirectories, etc.) to the help files (.DOC) must be identified by the environment variable ADIDOC.

See the section "Using Help" later in this chapter for instructions on how to set ADIDOC. Complete installation instructions can be found in the Release Note included with each shipment of the Cross-Software system.

5.2.2 Simulator Files

The Simulator uses a variety of files, illustrated in Figure 5.2, on the following page and listed in Table 5.1 below.

File Description	*Extension or name*
Required User Files	
Linked executable ADSP-2101 program	.EXE
Architecture Description file	.ACH
Optional User Files	
Symbol Table file	.SYM
Data files for I/O ports and SPORTs	.DAT (optional extension)
Required Simulator Files	
Simulator program	SIM2101.EXE
Help files only)	.DOC (required for Help
Optional Simulator Startup Files	
Initial window configuration	DD.WIN
Startup scripts	STARTUP
Example startup	EXAMPLE
Simulator-Created Files	
Temporary cache storage	BOOT.CAC BOOTE.CAC

Table 5.1 Simulator Files

5.2.3 Invoking The Simulator

The Simulator invocation command is:

sim2101 [–a *archname*] [–w *window*] [–s *scripts*]

If you have not given your Architecture Description file a unique name, filename 210x.ACH is assumed and need not be specified. If you have renamed the file, however, you must list this name as *archname* with the optional –a switch. The extension .ACH is assumed for this filename and need not be included. This Architecture Description file must have been used to link your program; the Simulator configures itself according to this target architecture. The Architecture Description is also displayed in the defaults window, as shown in that section of this chapter.

5 Simulator Functions

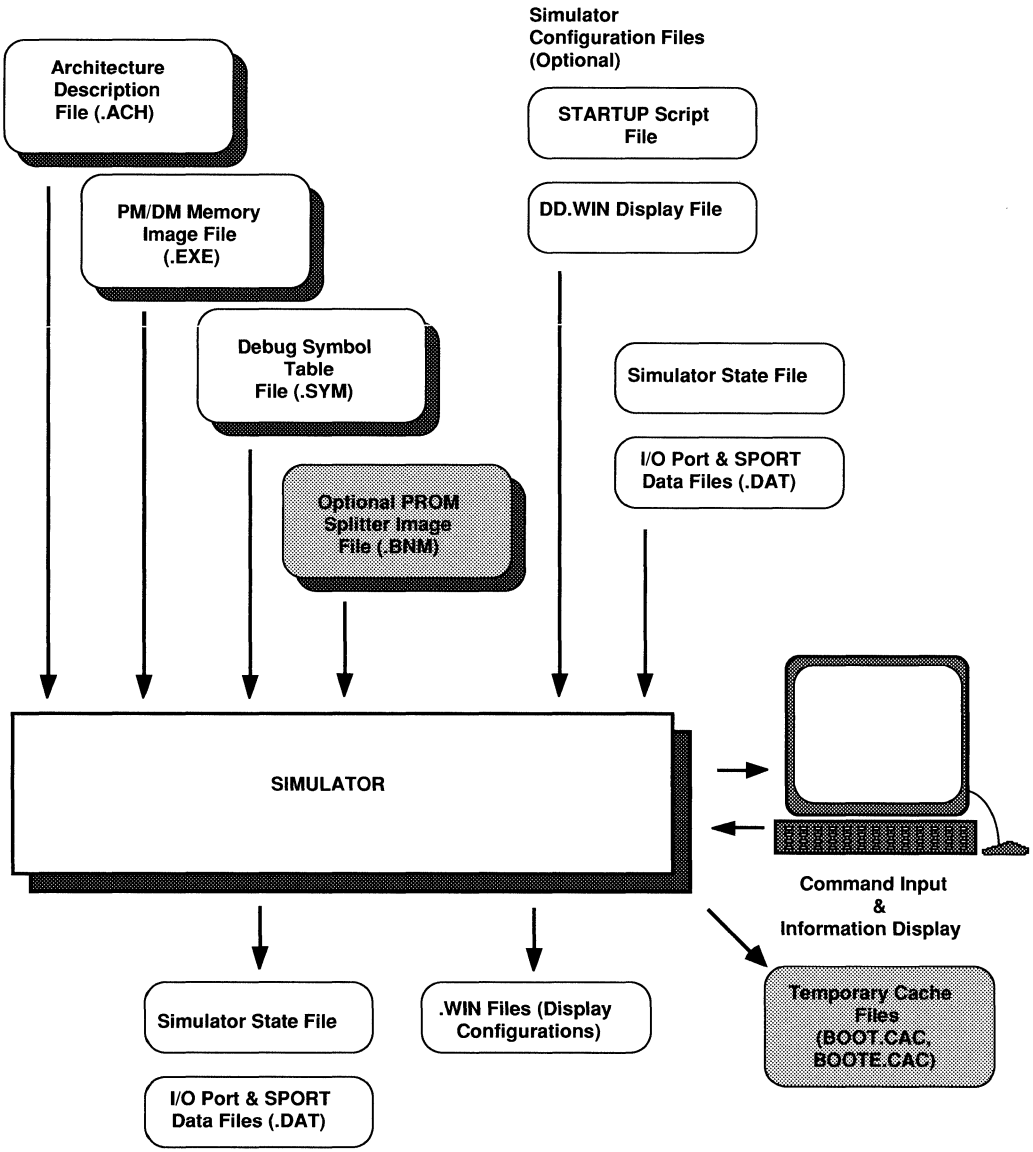


Figure 5.2 Files Used By The Simulator

The optional –w switch identifies a .WIN file containing a stored windows configuration which is loaded as the initial display when the Simulator is first booted (see "Saving A Rearranged Screen" in Chapter 6 for instructions on how to create this file). If this switch is omitted, the Simulator looks for a file named DD.WIN; this is the default for the startup screen. The Simulator automatically writes the file DD.WIN when exiting; it always contains the last screen/window display configuration. If this default display file is not found at startup, the screen looks like Figure 5.1.

The optional –s switch identifies a file containing Simulator commands to be executed automatically upon startup. If this switch is omitted, the Simulator looks for a file named STARTUP; this is the default name for the script file.

The script, or batch, file is a text file containing Simulator commands. Typically it would contain command aliases you have defined. It could also contain commands for loading a program into the Simulator, configuring I/O ports and the like. This file can be created with any editor. A sample startup file named EXAMPLE is provided with the Cross-Software package; the file contains an extensive set of aliased commands, and is intended for use only after the basic Simulator concepts have been mastered. See Chapter 6 for further information.

The Simulator creates two temporary files to store the contents of any boot memory of the system being simulated. These files are named BOOT.CAC and BOOTE.CAC. The files are normally purged upon quitting the Simulator; if, however, the Simulator program aborts prematurely for any reason, these files remain on your hard disk. They are of no use and can be deleted.

5.2.4 Simulator Command Overview

The Simulator generally provides multiple methods for achieving a given result. For example, there are two different methods for setting breakpoints in program memory. Consequently, it makes sense to think of the Simulator's *functions* rather than *command structure.*

The functional capabilities of the Simulator are described in detail in the rest of this chapter. They have been grouped into these broad classes:

- Interface management functions

These functions include the opening and closing of windows, changing the size and position of windows, and changing the appearance of a

window (removing or adding items to the window and rearranging the items displayed within the window's space). Additional functions described under this heading include navigating from window to window. Saving specific window configurations is possible and is described in the next chapter. Aliasing commands is another aspect of interface management.

- Set-up functions

These include loading the program to be simulated, opening I/O ports and associating data files with I/O ports and SPORTs for the purposes of simulating input and output data streams. Also included is the configuring of simulated interrupts.

- Register inspection & change functions

These functions allow you to view the contents of all the registers in the processor and, in most cases, to change their contents directly if desired. Several windows are dedicated to register displays.

- Memory inspection & change functions

These functions include simple display of the various memory spaces (as either data or code), saving the contents of memory to files for later analysis and plotting the contents of data memory.

- Simulator control & debugging functions

Control functions include starting and stopping the execution of your program and resetting the simulated processor. Debugging functions include setting breakpoints, break conditions and watchpoints. The Simulator supports a wide variety of break expressions for debugging purposes.

5.2.5 Simulator Notation Conventions

The Simulator understands a slightly different set of notation conventions than the Assembler, System Builder, etc. Most importantly, memory addresses and contents are specified differently. Remember also that the Simulator is a generally case-insensitive environment; uppercase and lowercase are used in the manual to highlight important terms for the reader but need not be entered this way. The exception to this convention is address labels (see below).

5.2.5.1 Specifying Addresses & Address Ranges

Addresses must be one of the following:

- A symbol. Using the symbol table, the Simulator determines the actual address specified by the symbolic reference. See also the discussion of boot memory labels versus program memory labels below. *Address labels are case-sensitive in the Simulator.*

- The memory specifiers PM[addr], DM[addr], or BOOT[addr], where the address is a symbol, constant, or expression. PM denotes program memory, treated as code or data, DM denotes data memory, and BOOT denotes boot memory. There is no difference in addressing between program memory code and program memory data.

  This form of address specification can be confusing; DM[addr] can be interpreted as either the address itself or the contents of that address. The guideline to follow is that DM[addr] is seen as an address when used to specify an address in a Simulator command, but DM[addr] implies the data contained at that address when evaluated in an expression (see "Simulator Expressions," below).

- A constant. The address space context is determined implicitly. For example, using a constant when prompted for an address while the program memory window is the active window is understood as an address in program memory.

An address range may be specified, using the address possibilities above, as either

start, end

where both terms are addresses as above, separated by a comma, or

start / length

where the first term is an address and the second term is a constant specifying how many memory locations are included in the range. The terms must be separated by the slash mark as shown.

An example of the first form is

```
pm[0x10], pm[0x18]
```

while an example of the second form is

```
pm[0x10] / 0x8
```

In ADSP-2101 programs with boot pages, labels are shown in boot memory displays in their standard form, such as

```
RESTARTER
```

but once booted into on-chip program memory (via a simulated reset or software boot) all such labels receive a prefix denoting their boot page of origin, as in

```
BOOT0_RESTARTER
```

Both labels resolve to the same 14-bit address. See the discussion in the section "Locating Symbols & Values," later in this chapter.

5.2.5.2 Simulator Expressions

General expressions may be used in place of constants in Simulator commands. Expression handling for the other ADSP-2101 Cross Software Tools is detailed in Chapter 1. For the System Builder and Assembler, the arithmetic and logical operators available for use in expressions are a subset of the C language operators. In the Simulator, however, the complete set of C operators is usable. For the Simulator, the following operators are added to those listed in Chapter 1 :

!	logical NOT
< > <= >=	relational operators
== !=	is equal, is not equal
&&	logical AND
| |	logical OR

In order of precedence, the complete set of operators available for use in the Simulator now becomes:

( )	left, right parenthesis
! ~ –	logical NOT, ones complement, unary minus
* / %	multiply, divide, modulus
+ –	addition, subtraction
<< >>	bitwise shifts
< > <= >=	relational operators
== !=	is equal, is not equal
&	bitwise AND
|	bitwise OR
^	bitwise XOR
&&	logical AND
||	logical OR

Another feature of Simulator expressions is that memory contents, such as data variables, and register contents may be used as operands. See the section "Registers Window" and Figure 5.7 for the available registers. Remember, though, that this is possible *in the Simulator only*. (The Assembler cannot evaluate memory and register values at assembly-time.)

Examples:

AX0 && AX1 DM[coeff] == 0x0035 (DM[taps + 16] * AR) - 3

5.3 INTERFACE MANAGEMENT FUNCTIONS

The Simulator, as of Release 2.0 and after, supports a user-configurable interface. Detailed examples of how to configure the interface and how to store and recall these configurations are given in the following chapter. This section gives a terse description of the basic functions.

(Note: ^ denotes the control, or CNTL, key.)

Figure 5.3, on the next page, shows the parts of a typical window.

5 Simulator Functions

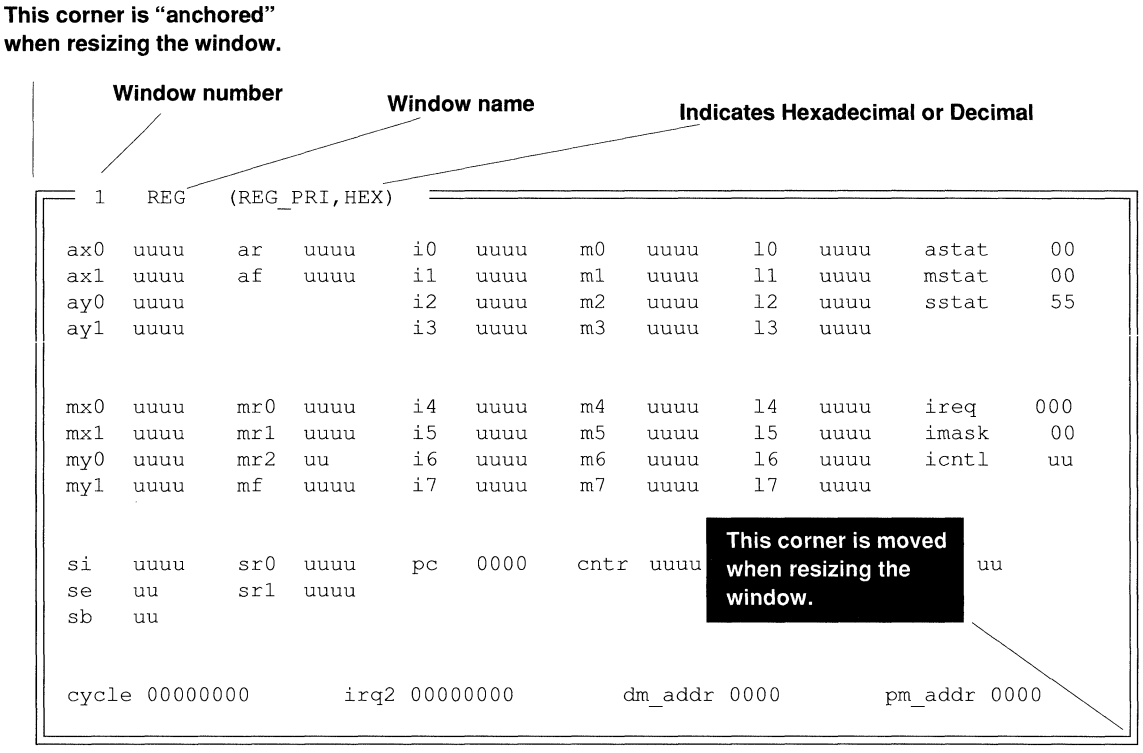


Figure 5.3 Parts of a Typical Window

5.3.1 Opening Windows

You can open any window from any context with the following sequence:

1. Key ^W to display the main menu (as shown in Figure 5.1).
2. Select OPEN, the default selection, by pressing Return.
3. A submenu of window selections appears; choose the window you wish to open. You may move the cursor down the list and then press Return or you may type the letter corresponding to the desired window, e.g. "d" for the register window. Pressing the ESC key exits the submenu without making a selection.

4. The default version of the window opens in the upper left corner of the screen and becomes the active window. Open windows are numbered; the newly opened window is given the next available number.

5.3.2 Changing Window Contents From Hex to Decimal

You may also change the numeric base of the contents of many windows from decimal to hexadecimal and back. All windows that can be changed in this way show the DEC or HEX notation in the title of the window. When the window is active, ^E toggles back and forth between these two choices.

The exception to this capability is that program memory (PM) addresses are always displayed in hexadecimal; data memory (DM) and program memory data (PMD) addresses can be toggled between DEC and HEX display.

5.3.3 Closing Windows

You cannot close the command window; it must remain open while the Simulator is running. Also, you can only close the active window. To close the active window, take these steps:

1. Key ^W to display the main menu (as shown in Figure 5.1).

2. Select CLOSE from the menu by typing the letter "c" or by moving the cursor down with the arrow key and pressing Return when CLOSE is selected. Pressing the ESC key exits the menu without closing a window.

3. The active window disappears from the display.

5.3.4 Moving From Window To Window

Regardless of the number of windows open (or visible) there is a single cursor. The window containing the cursor is the active window. On IBM PCs with color displays the border of the active window is a different color than inactive windows.

At startup the command window is the active window. To move through a group of open windows you may use any of the following procedures. The active window always lies on top of other windows in the event that windows overlap.

5.3.4.1 To Cycle Through All Windows

Keying ^Z activates the next window in the numbered sequence. Thus, ^Z moves you from the command window (always window zero) to window one, then window two, then window three and so on back to zero.

5.3.4.2 To Activate A Window By Number

Keying ^X, following by the window number and Return, directly activates the specified window. For example the sequence

^X3 (Return)

activates window number three. A maximum of ten windows may be open at any time; they are numbered from 0 to 9.

5.3.4.3 To Activate The Command Window

Keying ^X (Return) activates the command window directly. This is identical to keying ^X0 (Return).

5.3.5 Sizing Windows

The upper left corner of each window is anchored. The window is resized by moving the lower right corner of the window relative to the anchored corner.

A window must be active to be resized. To resize the active window, follow these steps:

1. Key ^W to display the main menu (as shown in Figure 5.1).

2. Select SIZE from the menu by typing the letter "s" or by moving the cursor down with the arrow key and pressing Return when SIZE is selected. Pressing the ESC key exits the menu without making a selection.

3. Reposition the lower right corner using the arrow keys. The window border moves one character or line space at a time as you press the arrow key. Press Return when the window reaches the desired size. Alternatively, you may quickly size the window a chosen number (#) of spaces by typing: #arrow key (Return is not necessary). For example, the following entry resizes a window by 4 line spaces upward: 4↑

5.3.6 Moving Windows

A window must be active to be moved. To move the active window, take these steps:

1. Key ^W to display the main menu.

2. Select MOVE from the menu by typing the letter "m" or by moving the cursor down with the arrow key and pressing Return when MOVE is selected. Pressing the ESC key exits the menu without making a selection.

3. The window's contents temporarily disappear, indicating that you may move the window.

4. Move the window using the arrow keys. The window moves one character or line space at a time as you press the arrow key. Press Return when the window reaches the desired location. The window's contents redisplay after Return. Alternatively, you may quickly move the window a chosen number (#) of spaces by typing: #arrow key (Return is not necessary). For example, the following entry moves a window to the left by 3 characters : 3← .

5.3.7 Rearranging Window Contents

You may rearrange the contents of active windows that have individual fields, like the register window. You may also delete individual fields from the window, and restore them later. See Table 5.2 for a list of windows which display processor registers in this way. Chapter 6 gives a detailed example of these procedures.

5.3.7.1 Deleting Window Fields

The procedure for deleting a field in an active window is:

1. Select the field by moving the cursor onto it.

2. Key ^D.

3. The field disappears from the display.

5.3.7.2 Undeleting Window Fields

The procedure for restoring a deleted field from an active window is:

1. Move the cursor to a blank location in the window; this is where the undeleted field will appear.

2. Key ^U. A menu of deleted fields for that window appears.

3. Select the desired field by moving the cursor down the list then press Return.

4. The deleted field reappears in the window at the current location of the cursor.

5.3.7.3 Moving Window Fields

The procedure for moving a field around in the active window is:

1. Select the field by moving the cursor onto it.

2. Key ^Y to toggle on this function.

3. Move the field, using the arrow keys, until it reaches the desired location.

4. To toggle off this function, key ^Y again or hit Return.

Saving specific window configurations is possible and is described in the next chapter.

5.3.8 Command Line Aliases

Aliasing commands – substituting a more desirable mnemonic for the Simulator's native command set – is another powerful feature. The aliasing must be done from the command window and follows the syntax

```
>j mystring 'command'
```

where J is the Simulator aliasing command, *mystring* is the new alias being defined and 'command' is any legal Simulator command enclosed in single quotation marks.Up to ten arguments may be passed to aliased commands using $1, $2 etc. For example the Simulator command to write the value 40 into data memory location hexadecimal 2FF is

```
>e dm[0x2ff] 40
```

which can be aliased to resemble the SETDM command of earlier Simulator releases (before Release 2.0) by entering this command

```
>j setdm 'e dm[$1] $2'
```

Now the command

```
>setdm 0x2FF 40
```

is executed as

```
>e dm[0x2FF] 40
```

If a filename is part of the command to be aliased, the filename itself must be enclosed in double quotes, as in:

```
>j loadpgm 'l "calc"'
```

It is also possible to list and save lists of aliased commands for use in a startup batch file. Details are given in the following chapter, Custom Simulator Configurations.

5.3.9 Using Help

The ADSP-2101 Simulator provides a basic help system with individual topics; there are no nested topics. To use the help first open the help window. You may wish to resize and relocate the help window for optimal reading.

This window displays an initial text introducing the help system. If the window is blank, this means that the Simulator cannot locate the help files on your computer. A warning message is given, saying that you must set an environment variable, ADIDOC, to identify the pathname of the directory containing the .DOC files used by the help system. For example, on an IBM PC with your Simulator in the subdirectory C:\DSPTOOLS and the help files in a subdirectory of that named \DOC, you would execute the following DOS command to set this variable.

```
> SET ADIDOC=C:\DSPTOOLS\DOC\
```

Remember, this is a DOS command, not a Simulator command

There are two navigational tools for reading help. First, within a given help text, you may use the arrow and PgUp and PgDn keys (or their equivalents on your keyboard) to scroll the contents of the current help text up and down for reading.

Second, you may key ^G (the go to command) in the help window to specify another help text and topic. You are prompted for the name of a topic. The list of topic names is given in the first help text that appears.

5 Simulator Functions

This initial help text is called "Help" and can be recalled by typing that name (and Return) at the ^G prompt. (The "Help" text is also returned to if the ^G command is given incorrectly.)

The list of help texts will change as new versions of the Simulator are released, so no definitive list is given here. In general, however, there is a help text corresponding to every command window command. For example, the breakpoint command B is described in a help text named "B" and so on. A list of the help topics other than Simulator commands is shown below. All the help files are simple text files. You may print them out to read if desired and even add your own help topics as you customize the interface of your Simulator.

The non-command help files are:

HELP	main help
BASES	numeric bases
COMMANDS	list of commands with brief definitions
EDITOR	command line editor
ADDR	address format
RANGE	address range format
EXPR	expression format

5.4 SET-UP FUNCTIONS

These include loading the program to be simulated, opening I/O ports and associating data files with I/O ports and SPORTs for the purposes of simulating input and output data streams. Also included is the configuring of simulated interrupts and some housekeeping operations.

These actions are accomplished by issuing commands in the command window. Multiple commands may be given on one line, separated by semicolons, as in

```
> L 'filename' ; J symbol 'command' ; D address
```

5.4.1 Loading A Program

The L command, given in the command window, loads the linked ADSP-2101 .EXE file and implicitly loads the corresponding .SYM file if it is present in the same directory. The syntax is simply

```
> L 'filename'
```

where *filename* is the main filename of your .EXE file, enclosed in single quotation marks. You need not append the .EXE extension; it is added by the Simulator. If the symbol table file cannot be found, a message reports this but the simulation can still be run. Without the .SYM file, however, labels and variable names do not appear, only addresses.

As a further check on program correctness, it is possible to load the boot PROM image file produced by the PROM Splitter. There should be no difference in the contents of the boot code and boot data windows whether loaded from the .EXE file or from the PROM Splitter output .BNM file.

The syntax of the LR command ('load ROM') is

```
> LR 'filename'
```

where *filename* is the name of the boot image file produced by the PROM Splitter. It is not necessary to use the .BNM extension of this filename. You must use the Motorola S record format for this purpose.

5.4.2 Opening & Closing An I/O Port

Parallel I/O Ports in data or program memory which have been defined in the System Builder (and .ACH file) must be explicitly opened in order to simulate them. Opening means that you associate them with data files. The data files serve as the source for simulated input and/or as the destination for simulated output. The data files may later be analyzed, graphed etc. to assess the processing of your algorithm.

Ports are opened from the command window. The command is

```
> O address [>'outfile.ext'] [<'infile.ext']
```

where *address* is a standard address specifier or symbolic port name, *outfile* is the pathname of a file to write output data to and *infile* is the name of a file to read simulated inputs from. Files may be specified in either order, always in single quotes; you must give the full filenames including extensions, if any. Giving both an input and an output file opens a bidirectional port. Giving just an input or an output file opens an input-only or output-only port.

Data files for I/O port data follow the .DAT format described in Appendix B, File Formats, at the end of this manual.

Giving the O command with no file arguments closes the port at the specified address.

The I/O status window, an example of which is shown in Figure 5.4, displays the opened ports and the files associated to provide simulated data flow. When a port is opened, a "P" is displayed to the left of the port's address in either the data memory or program memory window.

5.4.3 Opening A SPORT

SPORTs (serial ports) must be explicitly opened in order to simulate them. Opening means that you associate them with data files. The data files serve as the source for simulated serial input and/or the destination for simulated output. The data files may later be analyzed, graphed etc. to assess the processing of your algorithm.

SPORTs are opened from the command window with the command

```
> P 0 or 1  [>'outfile.ext'] [<'infile.ext']
```

where the digit 0 or 1 identifies which of the processor's two SPORTs is being opened, *outfile* is the pathname of a file to write simulated output to and *infile* is the pathname of a file to read simulated input from. Files may be specified in either order, each in single quotes; you must give the full filenames including extensions, if any. Listing both an input and an output file opens a SPORT for both sending and receiving. Listing just an input or an output file creates a send-only or receive-only configuration.

The data files for SPORT simulation must contain only ones and zeros to simulate the serial bit stream, and carriage returns (which are ignored). This .DAT format for SPORT data files is completely described in Appendix B, File Formats.

Giving the P command with no file arguments closes the numbered SPORT. Also, if you open a SPORT and later give the chip reset command (causing a re-boot of on-chip program memory), the SPORT is closed. The best procedure to follow is to do the boot load first and then open any SPORTs needed.

The SPORT status window, shown in Figure 5.5, displays the open/closed status of serial ports and the files associated with the simulated data flow.

SPORT operation in the Simulator has one limitation: externally-generated serial control signals cannot be simulated. The serial clock (SCLK),

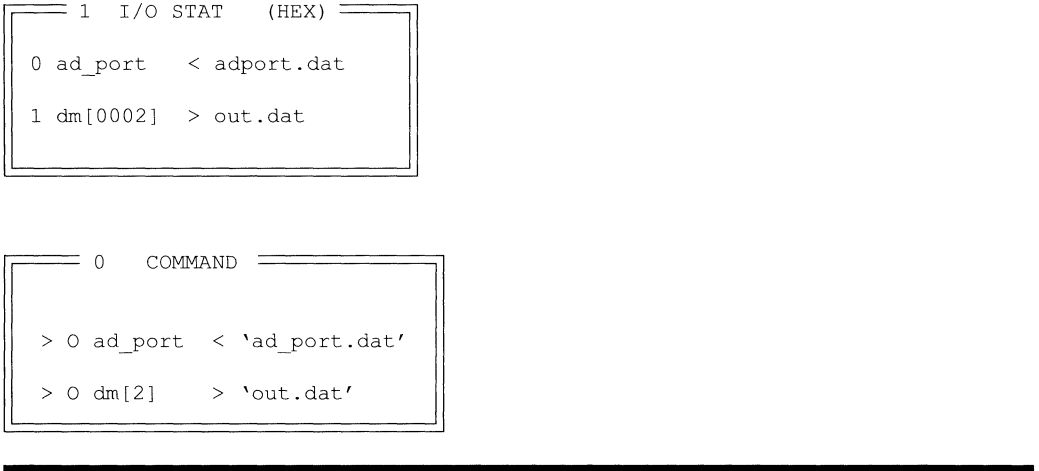


Figure 5.4 I/O Status Window

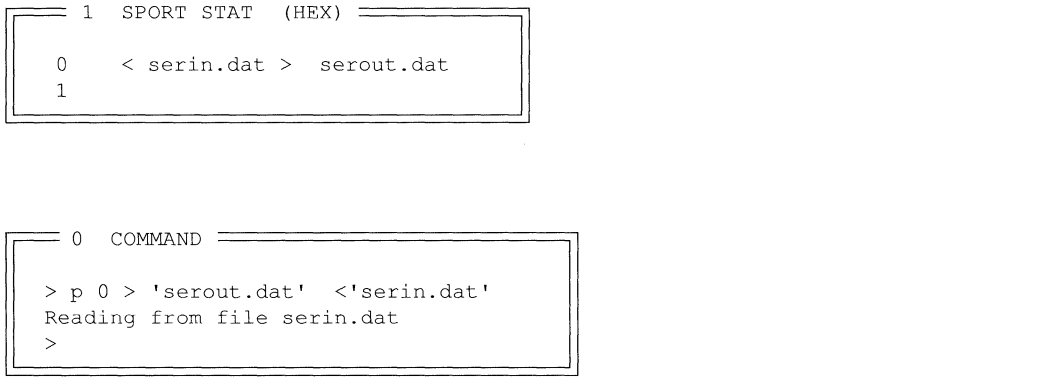


Figure 5.5 SPORT Status Window

transmit frame sync (TFS), and receive frame sync (RFS) signals must be internally-generated in order for simulated serial data flow to occur properly. Internal generation of SCLK is chosen by setting the ISCLK bit to 1 in the appropriate SPORT Control Register. Internal generation of TFS and RFS is chosen by setting the ITFS and IRFS bits to 1.

5.4.4 Simulating External Interrupts

Depending on the configuration of SPORT1, the ADSP-2101 may have one or three external interrupt pins. Internal interrupts, such as timer or SPORT interrupts, are simulated directly by the operation of those features. External interrupts can be simulated with a selected time interval of occurrence. From the command window, give the command

```
> I 0,1,or 2  mincycles maxcycles
```

where choosing 0, 1, or 2 identifies processor interrupts $\overline{\text{IRQ0}}$, $\overline{\text{IRQ1}}$ or $\overline{\text{IRQ2}}$, and *mincycles* and *maxcycles* are numbers of instruction cycles. The selected interrupt is generated randomly within a time range at least *mincycles* and no more than *maxcycles* from the last interrupt. For example, the command

```
> I 2 320 420
```

turns on $\overline{\text{IRQ2}}$ and generates this interrupt at a random time, once every 320 to 420 instruction cycles.

To halt the interrupt, repeat the command with no cycle arguments or with cycle arguments equal to zero.

5.4.5 Other Defaults (Defaults Window)

There are a number of miscellaneous defaults for the operation of the Simulator. These can be changed in the defaults window. The defaults window also displays the contents of the Architecture Description file. A sample of this window is shown in Figure 5.6.

When the window is first opened, the cursor is positioned on the "0" by profile enable. Typing a 1 enables profiling, as described in the section "Execution Profiling", later in this chapter. Enabling echo makes the Simulator echo every valid instruction in the command window as it is fetched while single-stepping through a program. Beep enable turns on the bell or beep of your computer or terminal. It sounds for each error or breakpoint. Screen update is the number of instruction cycles simulated

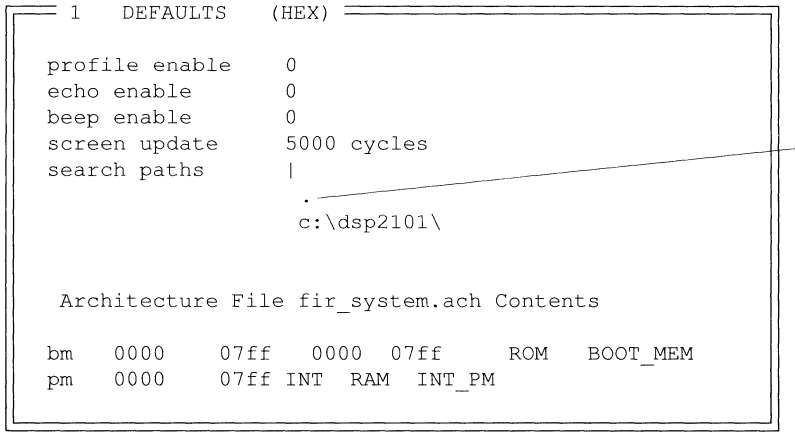


Figure 5.6 Defaults Window

before the screen is updated during continuous execution (see the discussion of the G command in "Control & Debugging Functions.").

Search paths are shown for any file to be read and are in addition to ADIDOC and the other environment variables. The dot (.) is the DOS symbol for the current directory; this is always one of the default search options.

The contents of the Architecture Description file are shown at the bottom of this window; you may need to scroll the window to see the complete architecture.

5.5 INSPECTING & ALTERING REGISTERS

You may view the contents of all registers in the processor and, in most cases, change their contents directly. The windows listed in Table 5.2 below are dedicated to register displays. The following sections show the default format of each window displaying registers and identify those registers. For debugging convenience, the Simulator also displays stacks and memory-mapped control registers in several windows, using the mnemonics given in the *ADSP-2101 User's Manual*.

5 Simulator Functions

Window	*Contents*
Register	Named registers
SPORT	Named registers and memory-mapped control registers, shown by mnemonic
Status registers	Individual bits identified from the MSTAT, SSTAT, IMASK and ICNTL
Control registers	Memory-mapped registers only; those controlling wait states, timer values, SPORT enables and boot configuration
Stack	Complete contents of count, loop, status and PC stacks

Table 5.2 Windows Showing Registers

Table 5.3, at the end of this section, summarizes the Simulator window location of each processor register.

5.5.1 Inspecting A Register

You may inspect the contents of a register in two ways. First, you may display the window containing that register and simply read its value from the screen. Second, regardless of whether or not the register's window is displayed or open, you may query the register's value in the command window with the question mark command (see the section "The ? Command and Expressions Window"). For example, the command

```
> ? ax0
```

invokes a response such as

```
ax0 = 0x002c
```

5.5.2 Altering A Register

You may alter the contents of a register directly in two ways. (The execution of your program, of course, also alters the contents of registers.) First, you may display the window containing that register and make it the active window. Move the cursor to the register field (the cursor positioned over the name of register) and type the new value. When you press Return, the new value replaces the old value.

When you directly type over a register in the window displaying the register, you may notice that the command window echoes the command equivalent of the direct change. This is the second method for changing a

register. From the command window you can change any register (whether displayed or not) with the command

```
> R register expression
```

where *register* is the name of a processor register and *expression* is the value to be loaded into the register.

Since changing the value in a register is such a frequent operation, an alternative form of this command is also provided. This form consists of the register name, and equal sign and the new value, as in

```
> ax0 = 0x002c
```

5.5.2.1 *"Undefined" Registers*

Uninitialized registers are denoted by "uuuu." The Simulator flags reading undefined registers as an error. You may wish to reset a register back to an uninitialized state. The U command, given in the command window, accomplishes this; you need only specify the register. For example, to undefine the ALU result register, the command is

```
> u ar0
```

5.5.3 Registers Window

The register window is shown in Figure 5.7, on the following page. It contains all the computational unit registers, the DAG registers and the values of most status registers in the processor, as shown plus the following information:

pc	program counter
cycle	execution cycle counter; reset only upon chip reset (or manually)
irq2	external interrupt request counter
dm_addr	data memory address
pm_addr	program memory address

These values can be used in break, watch, or general expressions.

5 Simulator Functions

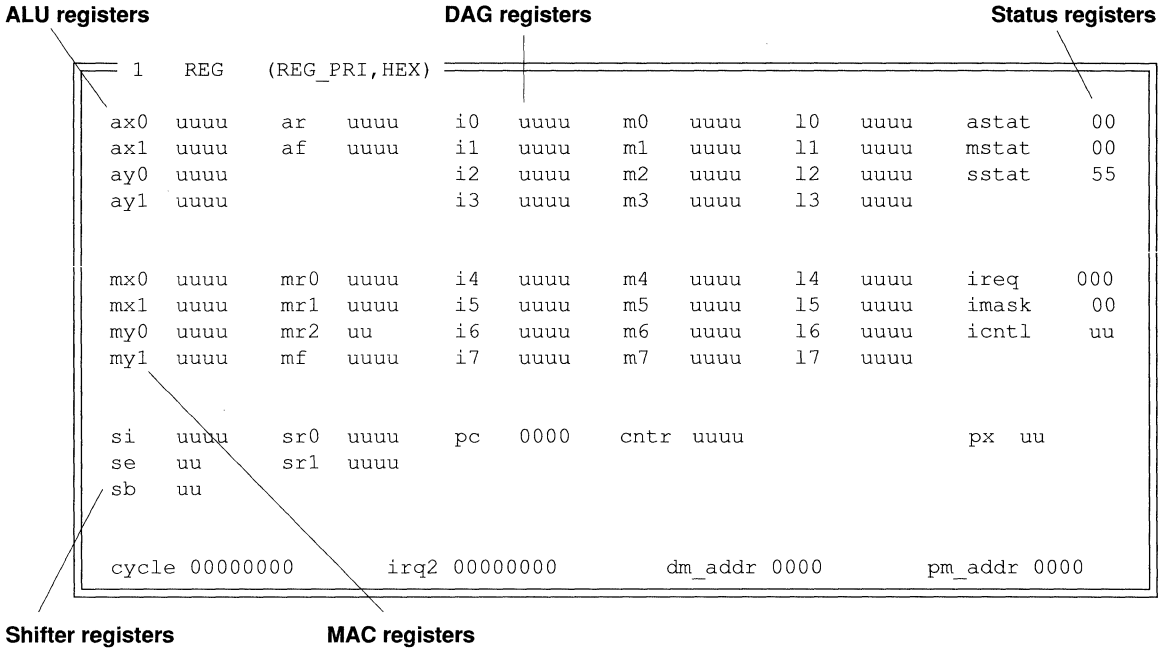


Figure 5.7 Register Window

5.5.4 SPORT Register Window

The SPORT register window is shown in Figure 5.8. It contains the SPORT data and control registers. The SPORT status window (discussed under "Set-up" earlier in this chapter) shows the open/closed status of SPORTS and the simulated serial data files.

5.5.5 Status Register Window

Figure 5.9 shows the status registers window. Note that this window shows certain selected bits in the MSTAT, SSTAT, IMASK and ICNTL registers while the register window itself (Figure 5.7) shows each complete status register as a single value.

The bits displayed are status/control bits for the primary program flow operations. Most of these can be toggled directly in the window in order to quickly enable or disable the associated function. Some of the listed bits are read-only status bits; these should not be changed by the user.

```
1    SPORT   (HEX)

slen0      0     dtype0    0     isclk0     0      mce0       0
irfs0      1     rfsr0     0     rfsw0      0      invtfs0    0
itfs0      0     tfsr0     0     tfsw0      0      invtfs1    0
sclkdiv0   uuuu                 rfsdiv0    uuuu
rbuf0      0     rireg0    u     rmreg0     u
tbuf0      0     tireg0    u     tmreg0     u      mcfd       0

slen1      0     dytpe1    0     isclk1     0
irfs1      0     rfsr1     0     rfsw1      0      invrfs0    0
itfs1      0     tfsr1     0     tfsw1      0      invrfs1    0
sclkdiv1   uuuu                 rfsdiv1    uuuu
rbuf1      0     rireg1    u     rmreg1     u
tbuf1      0     tireg1    u     tmreg1     u
mcre1 uuuu                      mcte1   uuuu
mcre0 uuuu                      mcte0   uuuu
rx0 uuuu                        tx0 uuuu
rx1 uuuu                        tx1 uuuu
```

Figure 5.8
SPORT Register Window

```
1 STATUS   (HEX)

pc_empty 0
pc_overflow 0
count_empty 1
count_overflow 0
status_empty 1
status_ovr 0
loop_empty 0
loop_overflow 0
data_bank_sel 0
bit_reverse 0
alu_overflow 0
ar_sat 0
int0_sens 0
int1_sens 0
int2_sens 0
int3_sens 0
int0_enable 0
int1_enable 0
int2_enable 0
int3_enable 0
int_nesting_mode 0
```

Figure 5.9
Status Register Window

5 Simulator Functions

5.5.6 Control Registers Window

Figure 5.10 shows the control registers window. This window shows the contents of the individual fields of the first five 16-bit memory-mapped processor control registers (from DM[0x3FFF] to DM[0x3FFB] inclusive).

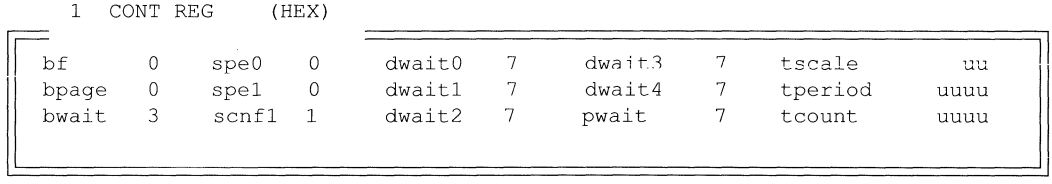


Figure 5.10 Control Registers Window

5.5.7 Stack Window

The stack window, shown in Figure 5.11, shows the four stacks in the program sequencer: CNTR stack, LOOP stack, STATUS stack, and PC stack. The top line of each stack display is the top of the stack. The four LSBs of the loop stack are the termination code; the 14 MSBs are the return address.

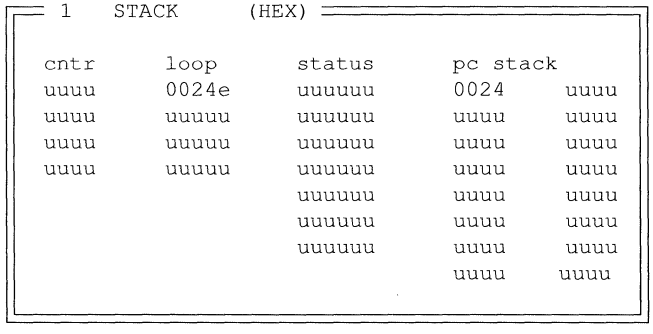


Figure 5.11 Stack Window

For this register ...	*Look in this window*
AX0	Register
AX1	Register
AY0	Register
AY1	Register
AR	Register
AF	Register
MX0	Register
MX1	Register
MY0	Register
MY1	Register
MR0	Register
MR1	Register
MR2	Register
MF	Register
SI	Register
SE	Register
SR0	Register
SR1	Register
SB	Register
PC	Register
PX	Register
I0-7	Register
M0-7	Register
L0-7	Register
ASTAT	Register, Flag
MSTAT	Register, Bits 0-3 in Status
SSTAT	Register, Status
IMASK	Register, Status
ICNTL	Register,
IREG	Register
CNTR	Register
CNTR_NO_PUSH	(same contents as CNTR)
RX0	SPORT
TX0	SPORT
RX1	SPORT
TX1	SPORT
Memory-mapped Control Registers	Control Registers, SPORT

Table 5.3 Register Location By Window

5 Simulator Functions

5.6 INSPECTING & ALTERING MEMORY

This section describes the methods for viewing and altering specific locations in any of the ADSP-2101's memory spaces. These functions include simple display of the various memory spaces (as either data or code), entering new values and saving the contents of memory to files for later analysis.

These actions are accomplished by issuing commands in the command window. Multiple commands may be given on one line, separated by semicolons, as in

```
> L 'filename' ; J symbol 'command' ; D address
```

5.6.1 Inspecting A Memory Location

There are two methods for inspecting a location in memory. First, you may open the appropriate memory window and make desired location visible in the window. Second, you may directly query any location or range from the command window.

For the first method, once you have opened the desired memory window, you can change the range of memory addresses and contents displayed in several ways as listed below.

- Paging / memory window is active window

  The arrow and PgUp and PgDn keys (PC keyboard) and their equivalents on other computers cause the memory window to scroll.

- Go To Address / memory window is active window

  Keying ^G interactively prompts you (in the memory window itself) to enter an address. The address must be entered without the pm[ ], dm[ ], or boot[ ] notation. When you press Return the first line of the window displays this address.

- Go To Address / command window is active window

  If you use the ^G method from within an active memory window, you may note the command window echoing a command. This command (K), issued in the command window, alters what is displayed in an open window without making the window active. The form of this command is

  ```
  > K windownum address
  ```

where *windownum* is the number of the window and *address* is the memory location to be displayed. Again, the address given must be the constant or symbol only. In other words, you must enter

```
> K 1 25          correct
```

rather than

```
> K 1 PM[25]      incorrect
```

The result of this command is like the ^G method above; the specified address is brought into view in the window.

The second method displays the memory contents directly in the command window; the memory window does not have to be open or visible. The command has the form

```
> D address or range  [> 'filename']
```

where *address* or *range* identify a location or range of locations in memory. The default action is to display this location (or range) in the command window. If the optional file redirection is given, the contents of memory are written to the file instead of the screen. This can be used for saving the state of arrays in data memory or other memory ranges. The format of this file is described in Appendix B, File Formats.

5.6.2 Tracking

Tracking enables you to view the code and data accessed when a program is run with the single step (S) or go (G) commands. When tracking is enabled in the program memory window, the window will automatically scroll through the code being executed to follow the processor's program counter. If tracking is turned on in the data memory window, any data memory locations accessed are scrolled into view in that window. Tracking is enabled/disabled by giving the command

```
> T window#
```

where *window#* is the number of the program memory, data memory, or program memory data windows.

Tracking can also be toggled in the active window with the control key sequence ^T.

5.6.3 Locating Symbols & Values

The cross-reference command, X, given in the command window, locates symbols. The general form of the command is

```
> X symbol
```

where *symbol* is any symbol you believe is defined in your program. For example, to find the label *RESTARTER*, the command (and its response) would be

```
> x RESTARTER
RESTARTER = boot[0x001c]
```

Remember that labels are case-sensitive in the Simulator. If you enter *restarter* when the actual label is *RESTARTER* it cannot be found.

The find command, F, finds numeric values in a given memory range. The numeric value may also be an opcode. In addition, for ease of use, the F command accepts source versions of commands and assembles them. The syntax of the F ("find") command is

```
> F address expression
```

where *address* is any valid address or address range specifier and *expression* is any valid expression including instructions and simple numeric values. Because this command is actually searching for an exact numeric match, it does not do partial matching. For example, to locate the instruction

```
jump RESTARTER;
```

you cannot give the F command as

```
> f pm[0]/100 jump
```

Incomplete specification of expression!

The F command is not a word processing command. To obtain the correct answer, you must give the complete instruction (the semicolon is optional) including the specific label in a case-sensitive form, as in:

```
> f pm[0]/100 jump RESTARTER
```

When the expression is found, the location is displayed:

```
pm[0019]  00035f  jump RESTARTER
```

The best way to locate a symbol, then, is to use the X command. It is case-sensitive and can discriminate between data, program and boot memory when it searches for a symbol. The best way to find a data value or specific opcode is to use the F command (which does not discriminate between program and boot memory).

5.6.4 Plotting The Contents Of Memory

An additional and useful way to inspect memory is with the PL or plot command. This allows you to plot up to 640 points on the screen. The syntax of PL is

```
> PL range decimation
```

where *range* is an address range and *decimation* is an integer value (or expression) indicating which memory locations to select for graphing. Each graphed point corresponds to a single 16-bit data value. For data that is interleaved for example, you would select every other word with

```
> PL dm[0x100]/0xff 2
```

The screen clears and the graph is displayed. Pressing Return returns you to the previous display.

Since no more than 640 points total can be graphed, the length of the range divided by the decimation factor must not exceed 640.

5.6.5 Altering A Memory Location

You may alter the contents of memory directly in two ways, although there are some differences between program and data memory. First, you may display the window containing that memory and make it the active window. Position the cursor on the address to be changed and type the new contents. When you press Return, the new value replaces the old contents.

If you are entering a new value for data memory that value is a number. If you are entering a new value in the program memory (code) window that value is an instruction. But, if you are entering a new value in the program memory data window, that value is a number, including an opcode.

When you directly type over a data memory location, you may notice that the command window echoes the command equivalent of the direct change. This is the second method. From the command window you can change the numeric contents of any address in memory (whether displayed or not) with the command

```
> E address expression
```

where *address* is the address or address range to be altered and *expression* is the value to be written into the specified portion of memory. For example, to write a zero into the range of data memory from address 200 to address 300, the command is

```
> E dm[200]/100 0
```

Likewise, to write the opcode for NOP (which is all zeroes) into program memory location two, the command is

```
> E pm[2] 0
```

You may also enter the contents of a file into a memory range with a variation of the E command. Its syntax is

```
> E start_address <'filename'
```

where *start_address* is any address in memory and *filename* is the name of a file to be read from. This file must be in the data file format (.DAT) described in Appendix B. The range of memory written is determined by the size of the file; the file is read until end of file is reached. If the file is too large, unpredictable results may occur.

5.6.5.1 Altering Instructions

The command equivalent for directly typing an instruction is the A (assemble) command whose syntax is

```
> A address instruction
```

where *address* specifies a location in program memory and *instruction* is a valid ADSP-2101 instruction (not opcode). The terminal semicolon in the instruction is optional; the Simulator correctly assembles the instruction regardless of whether or not the semicolon is entered. (See also the discussion of the V command under "Miscellaneous Features" at the end of this chapter.)

For example, to alter program memory at location two to the NOP instruction (as shown numerically above) you would type

```
> A pm[2] nop
```

Any change is immediately displayed in the pertinent memory window, if open.

5.6.5.2 "Undefined" Memory Locations

Uninitialized memory locations are denoted by the word "undefined." The Simulator flags operations such as reading undefined memory as errors. You may wish to reset portions of memory back to an uninitialized state. The U command, given in the command window, accomplishes this. You must specify the address (individual address or address range). For example, to undefine the first sixteen data memory addresses, the command is

```
> U dm[0x0] / 16
```

5.6.6 Program Memory (Code) Window

The program memory window shows program memory as code with fully symbolic disassembly. The default arrangement of this window cannot be altered and appears (after loading a program) as in Figure 5.12, on the next page. To the left of each address is a two-letter code which indicates the following:

I/X internal/external memory
A/O RAM/ROM

When an I/O port is opened in program memory , a "P" is displayed with the I/X, A/O code at the port's address.

Address labels are shown in the disassembled code. Labels which originate in boot memory code have the boot page number appended to them. An example of this is the BOOT0_RESTARTER symbol shown in Figure 5.12, on the next page.

The contents of the processor's internal program memory will change when a new page of boot code is loaded.

5 Simulator Functions

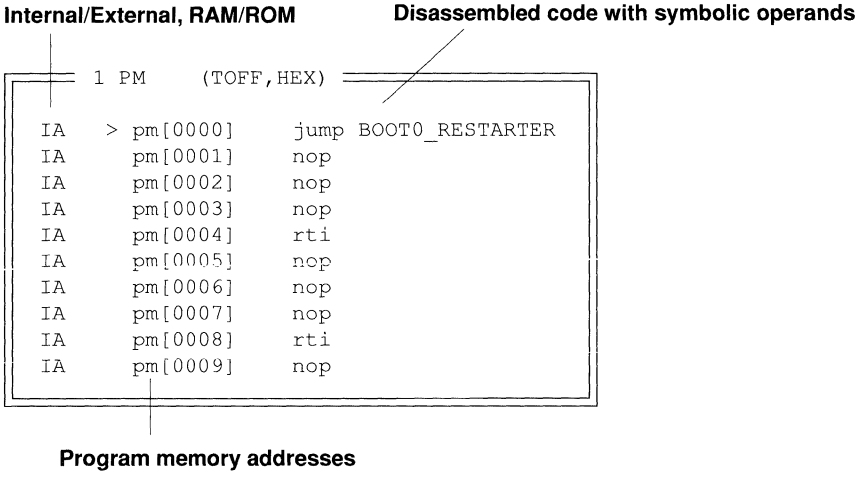


Figure 5.12 Program Memory (Code) Window

5.6.7 Program Memory As Data

The program memory data window shows program memory as numeric data. This consists of either opcodes for code segments in program memory or actual numeric values for data segments in program memory. The only way to view opcodes is to open this window. The default configuration of this window, which cannot be changed, is shown in Figure 5.13. To the left of each address is a two-letter code which indicates the following:

I/X	internal/external memory
A/O	RAM/ROM

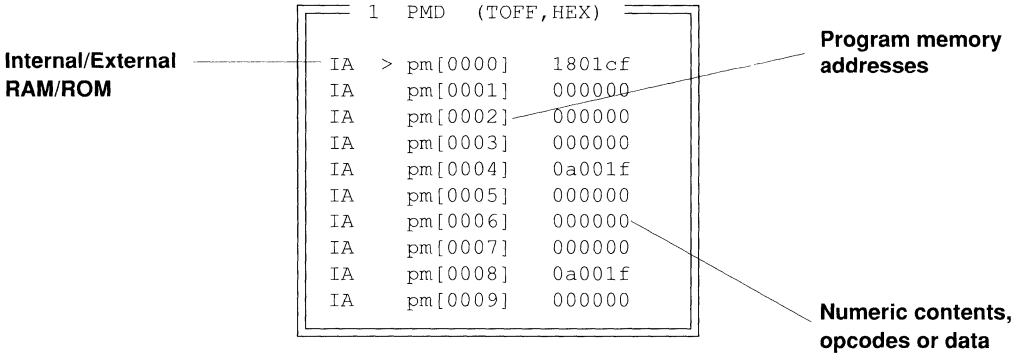


Figure 5.13 Program Memory Data Window

5.6.8 Data Memory

The data memory window shows the numeric contents of data memory and any symbols defined for data structures. The default configuration of this window, which cannot be changed, is shown in Figure 5.14, on the next page. To the left of each address is a two or three-character code which indicates the following:

I/X	internal/external memory
A/O	RAM/ROM
0, 1, 2, 3, or 4	wait state zones 0-4 of external DM

When an I/O port is opened in data memory , a "P" is displayed with the I/X, A/O code at the port's address.

External data memory is divided into five address zones for purposes of wait state programming. Where external memory is displayed in the data memory window, the zone number is shown. The numbers represent the zones for DWAIT0 through DWAIT4; these wait states are selected in the Data Memory Wait State Control Register, located at DM[0x3FFE]. Refer to the *ADSP-2101 User's Manual*, under "Data Memory Interface," for further information.

5 Simulator Functions

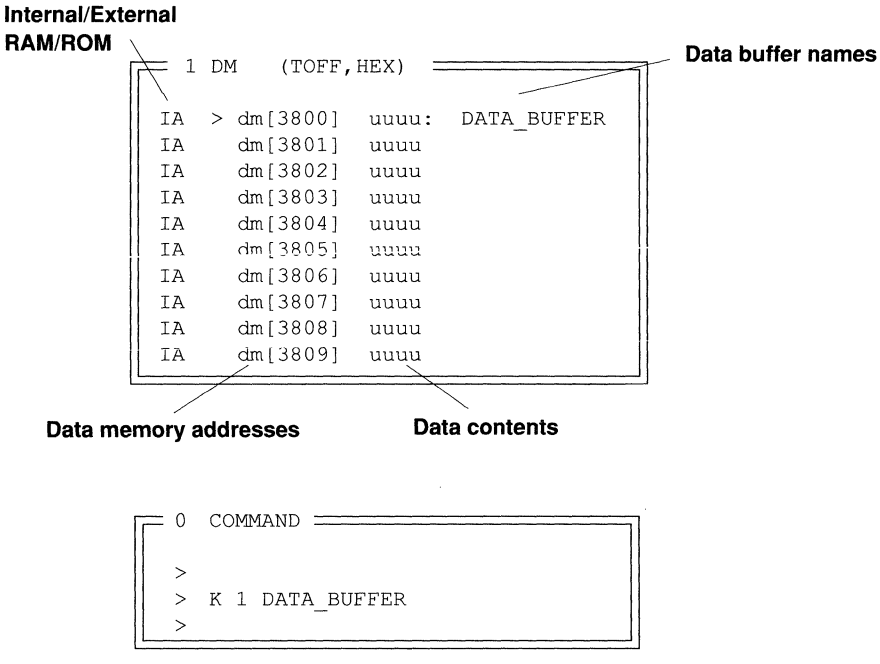


Figure 5.14 Data Memory Window

5.6.9 Boot Memory

Like program memory, boot memory may be viewed in two ways, each through its own window. Boot memory code (Figure 5.15) shows disassembled source code by address in boot memory while boot memory data (Figure 5.16) shows opcodes and numeric data values. In both cases the address shown is a (24-bit) word address. The entire 16K length of boot memory is contained in the windows, from page 0 up to page 7. The Simulator does not support byte addressing in boot memory directly, nor does it show the extra bytes added to pad each instruction or data word for PROM alignment purposes. Byte addresses are determined only by the PROM Splitter.

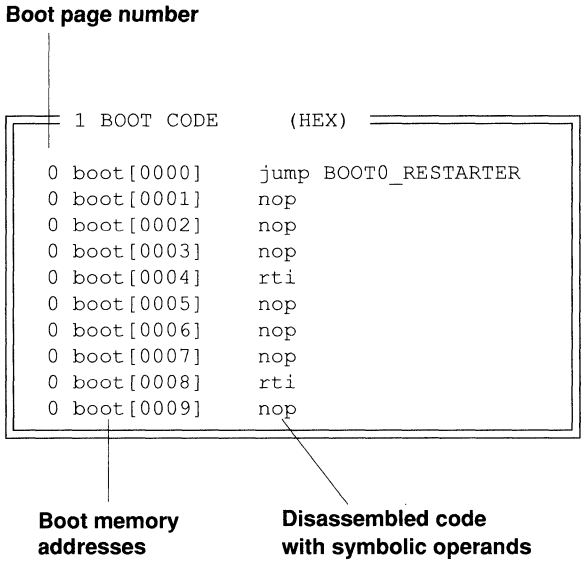


Figure 5.15 Boot Memory Code Window

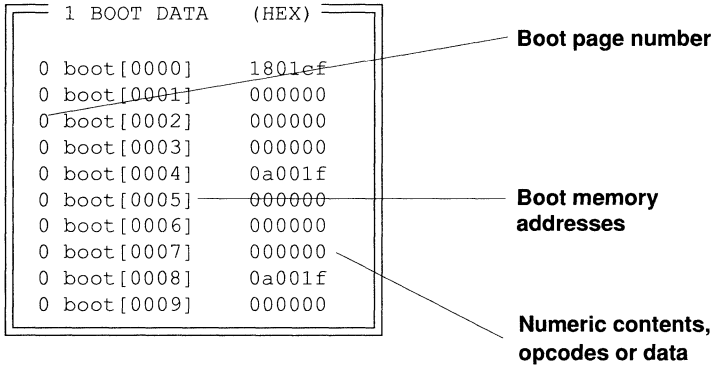


Figure 5.16 Boot Memory Data Window

5 Simulator Functions

5.7 CONTROL & DEBUGGING FUNCTIONS

Control functions include starting and stopping the execution of your program and resetting the simulated processor. Debugging functions include setting breakpoints, break conditions and watchpoints. Any expression can be quickly evaluated in the expressions window. The trace window provides a history of external bus activity. Profiling (via the profile window) is a tool for analyzing the time spent executing various parts of your program.

Remember that multiple commands may be given on one line, separated by semicolons, as in

```
> L 'filename' ; J symbol 'command' ; D address
```

5.7.1 Resetting The Processor: CR and RE

There are two command window commands for resetting the processor: CR and RE.

CR, which stands for chip reset, simulates a hardware reset of the processor. It is the same as pulling the $\overline{\text{RESET}}$ line low in a hardware system. All clocks, registers and stacks become reset or undefined. The state of on-chip memory is undefined. (While in some cases you may see values in on-chip memory "surviving" a reset, this is not guaranteed to be the case.) Finally, boot page zero is booted into the processor if MMAP equals 0 in the Architecture Description file, and the PC is set to the restart vector at address 0. Execution does not actually begin in the Simulator, although it would in hardware.

RE, which stands for reset, performs a subset of the functions of CR. It omits the boot loading sequence, but otherwise resets the processor. On-chip memory remains intact.

5.7.2 Single-Step Execution

The S command, given in the command window, steps the processor through one or more instructions. Execution always begins at the current program counter value.

For example, the command

```
> S 10
```

executes the next ten instructions, while

```
> S
```

executes only the next instruction. Execution can always be interrupted by pressing any key. If echoing is enabled (in the defaults window), the next instruction is shown in the command window as you step through your program.

5.7.3 Running & Halting

The G command with no arguments, as in

```
> G
```

starts the simulated processor running from the current PC value for an unlimited number of instructions. The Simulator halts only for the following events:

- You press any key to interrupt execution
- A simulation error occurs
- A breakpoint is reached or a break change or expression becomes true.

The G command can also be given an address (constant, expression, or label) to stop at; execution continues until that address is reached as in

```
> G fir_halt
```

The command RUNFAST is a slightly different version of the G command. RUNFAST also causes the Simulator to run, but will not stop when a key is pressed– only on a break reached or an error. Care must be taken when using this command, though, since the Simulator does not stop if a break is not reached.

5 Simulator Functions

5.7.4 Breaks

Breaks halt execution. Breaks include breakpoints, break expressions, break changes, and break ranges. A breakpoint is a location in program memory where execution halts. In the program memory (code) window breakpoint locations are marked with a B in the first column. Break expressions, changes, and ranges are defined in the following sections.

5.7.4.1 Setting Breakpoints & Break Ranges

There are two ways to set breakpoints. From the command window, you may identify the program memory location in the command

```
> B address
```

where *address* is any valid program memory address expression, such as

```
> B pm[0x001A]
```

which sets a breakpoint at location hexadecimal 001A. A running simulation halts when this instruction is fetched, but before it is executed.

From the program memory (code) window, you can set a breakpoint at the instruction marked by the cursor by keying ^B. The command window echoes this action with the command as above.

Break ranges cause execution to stop when a selected range of program memory is accessed by the processor. An instruction fetch from any address within the range will cause the break to occur. A break range is set with the following command:

```
> BR range
```

5.7.4.2 Viewing Breaks

Instructions selected as breakpoints are marked with the B indicator in the first column of the program memory window. There are two ways to recall the complete list of breakpoints, beyond what can be viewed directly in the program memory window.

You may view a list of current breakpoints, break expressions, break changes, and break ranges in the command window by giving the B

command with no arguments. The breaks are listed in the command window. For example, if only breakpoints are defined you might see

```
> B
[20]  pm[001a]
[21]  pm[0032]
[22]  pm[002c]
```

An alternative is to open the breakpoints window, which displays the list of breakpoints, as shown in Figure 5.17. Note that in both the breakpoints window and command window lists the numbering on the left reflects the order in which breakpoints were declared. The numbers are assigned from 20 to 39; thus you may have a total of 20 breakpoints defined at one time.

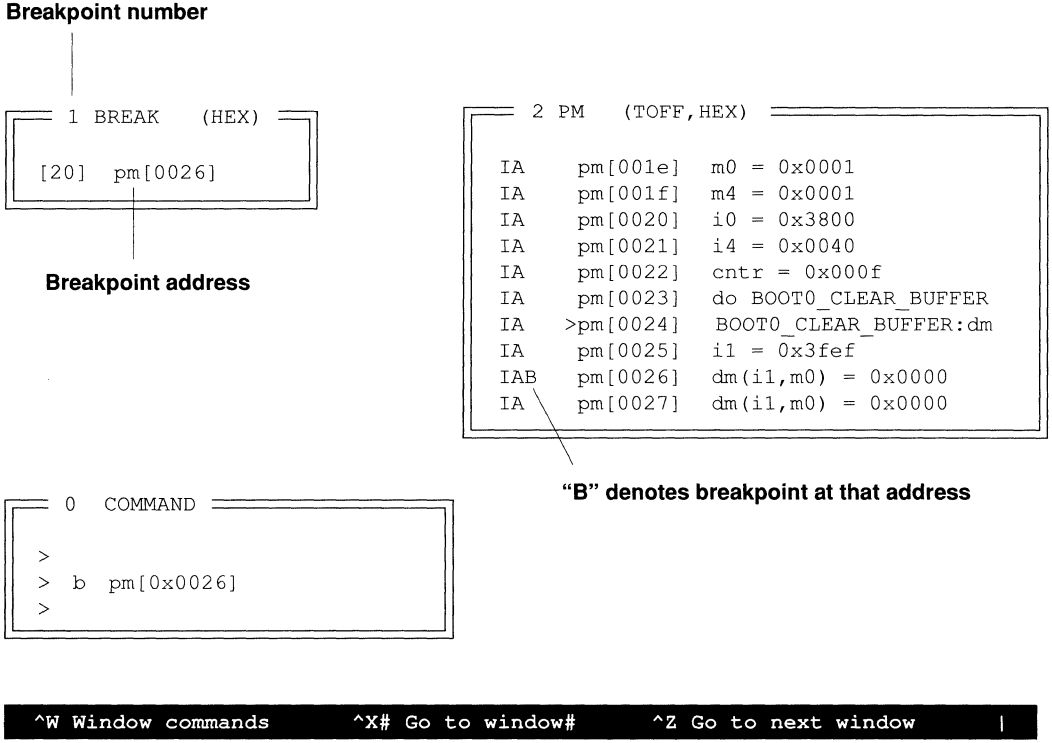


Figure 5.17 Breakpoints Window

5.7.4.3 Break Expressions & Changes

In addition to breakpoints, the Simulator can break on the state of expressions. (Refer to the section "Simulator Expressions" in this chapter.) It can break whenever a break expression is not equal to (logical) zero and it can break whenever the value of an expression changes. The nature of the expression itself is the same for both.

For example, you may wish to break execution whenever the expression

```
AX0 > 3
```

is true. As long as the AX0 register contains a value less than or equal to three, the expression is false (logical zero). When AX0 is loaded with a larger value, however, the expression becomes true (logical non-zero).

Such a break expression is entered with the BE command in the command window as

```
> BE AX0 > 3
```

The BC ("break on change") command, on the other hand, enters a break expression and halts execution upon *any change* in the value of the expression. An expression can be as simple as a single register, as in the command

```
> BC AX0
```

which halts upon every change in the AX0 register, or it can be more complex. All register names, address locations and constants may be used with the full set of operators to create the expression.

The break expressions window shows the current list of break expressions and break changes, along with their current value. It is opened like any window. If you had executed both of the examples just above, the window might appear as shown in Figure 5.18.

Numbering of break expressions in this window runs from 0 to 19, again limiting the user to 20 expressions at one time.

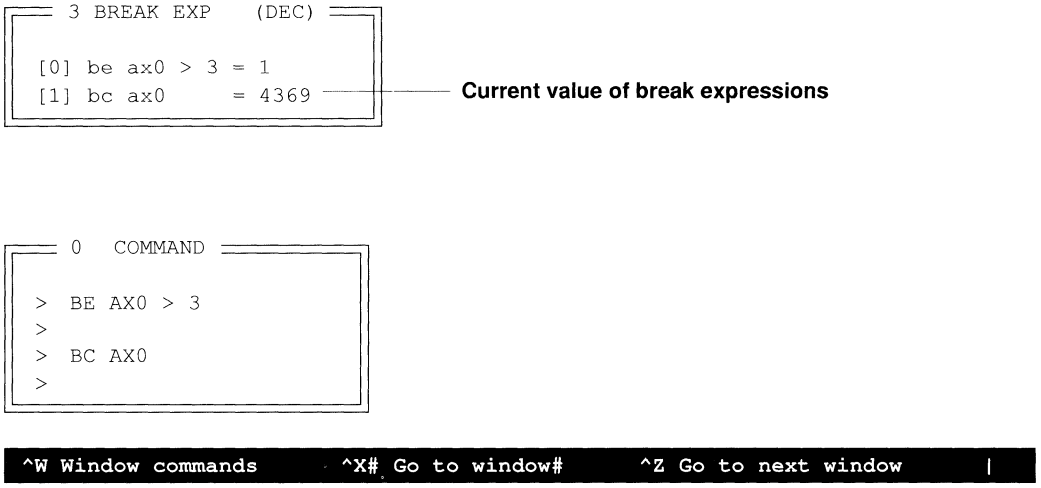


Figure 5.18 Break Expressions Window

5.7.4.4 Deleting Breaks

There is a single command for deleting breakpoints, break expressions, break changes, and break ranges. From the command window, enter

```
> BD address
```

where *address* is an address expression for a program memory location currently selected as a breakpoint or break range.

For break expressions and changes, use the break numbers shown in the break expressions window, as in

```
> BD 5
```

Another way to delete breakpoints is to use the ^R control key sequence in the program memory window; this is analogous to keying ^B to set a breakpoint. The cursor must be positioned at the breakpoint to be deleted when ^R is keyed.

5.7.5 Watchpoints & Watch Expressions

Breaks halt execution, while "watches" display a message but do not halt execution. Watchpoints are memory locations; whenever they are read or written a message and the location's value are displayed in the command window. A watchpoint can also be set over a range of addresses. A watch expression is identical to a break change except that execution does not halt.

5.7.5.1 Setting Watchpoints

The W command, given in the command window, identifies an address or address range as a watchpoint. The general form is

```
> W address
```

where *address* is an address or address range specifier. For example, to set a watchpoint on data memory location 0x003F, the command is

```
> w dm[0x3f]
```

A maximum of 25 watchpoints may be active simultaneously.

5.7.5.2 Setting Watch Expressions

The WE command, given in the command window, defines a watch expression, which acts like a break change without halting execution. The command form is

```
> WE expression
```

where *expression* is any valid expression. (Refer to the description of "Simulator Expressions" in this chapter). For example, to watch the ALU carry bit (AC) you would give the command

```
> we ac
```

After giving this command, any change in the AC bit is reported in the command window. To watch the sum of the carry and overflow bits you could give the command

```
> we ac + av
```

which would display a message whenever the sum of these bits changed.

A maximum of 25 watch expressions may be active simultaneously.

5.7.5.3 Listing Watchpoints and Watch Expressions

The W command, given in the command window with no arguments, lists all watchpoints and watch expressions along with their assigned numbers. You must know these numbers to delete them.

5.7.5.4 Deleting Watchpoints and Watch Expressions

The WD command, given in the command window, deletes the watchpoint or watch expression whose number is given as the argument. The general form is

```
> WD watchnum
```

where *watchnum* is a number identifying a current watchpoint or watch expression. For example, the command

```
> wd 0
```

deletes watchpoint zero.

5.7.6 The ? Command and Expressions Window

The ? command is a general purpose debugging tool which evaluates any valid expression. (Refer to the section "Simulator Expressions" in this chapter). The current value of the specified expression is shown on the command line when the ? command is given in the command window. For example, if the AX0 register contains the value 512 and data memory location 55 holds the value 128, then the following entry:

```
> ? ax0 + dm[55]
```

returns this in the command window:

```
ax0 + dm[55] = 640
```

In this example, DM[55] is evaluated to be the data contained at address 55 (128) rather than the address itself (55). This illustrates the difference in how this type of specification is interpreted by the Simulator. If used in an expression which must evaluate to a numerical value, DM[addr] denotes the *contents of the location*. If listed where an address is expected in a Simulator command, DM[addr] denotes the *address itself*.

The expressions window, which is opened in the normal manner, can be used to display and evaluate expressions as other debugging actions are

performed. The ?+ command adds an expression to the window, while the ?– command deletes one. These commands take the following form:

```
> ?+ expression      (add expression to expressions window)

> ?- #               (delete expression # from expressions window)
```

Expressions added to this window are numbered from 0 to 9; you may have a total of 10 active at any time. Figure 5.19 gives an example of the expressions window.

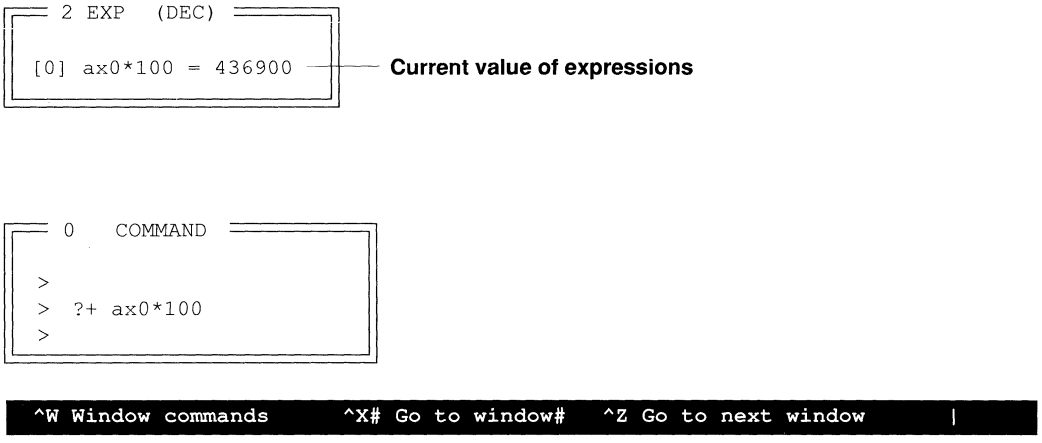


Figure 5.19 Expressions Window

5.7.7 Execution History (Trace Window)

The trace window allows you to capture a history of processor activity during program execution. You must set up tracing before running some portion of code. Open the trace window and key ^S. You are prompted for the number of lines of code to record (trace). A "line" is required to record one cycle of execution history. Enter any reasonable number (memory limits on a PC may restrict this) and press Return. If you do not choose a number of lines, the default value of 10 will be used.

You can now execute code with the S or G commands. As execution proceeds the trace history shows the value of the PC, the instruction

executed, and any data bus activity. Once the specified number of cycles/ lines have been filled up, the oldest lines are deleted as new lines are recorded.

Figure 5.20 shows a trace window. There are four fields of information displayed for each instruction in the window. From left to right, these are: program counter value, disassembled code (instructions) with address labels, data memory read or write performed, and program memory data read or write performed.

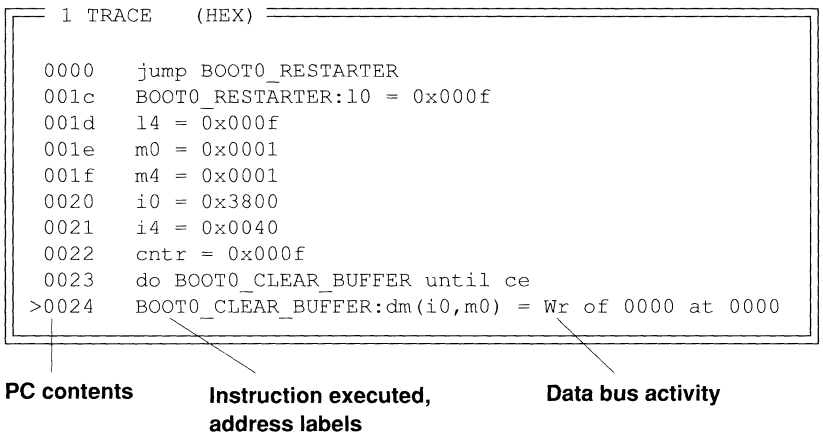


Figure 5.20 Trace Window

5.7.8 Execution Profiling (Profile Window)

The profile window and profiling commands allow you to analyze the execution patterns of your program. You specify sections of code to be monitored and the list of profiled ranges plus the execution profile itself is displayed in the profile window. The form of the profile window is similar to the ADSP-2101 Emulator.

The fundamental profiling information this window displays is the time spent executing different ranges in your program. This information is expressed in percentages. For any given time base (see below), percentages are given for each address range you have identified and for "other," which is all other code outside the defined ranges.

The profiling activity uses three time bases: short term, long term and cumulative. You choose the number of instruction (execution) cycles in the short term and long term time bases; the cumulative total includes all execution cycles since you last reset the profile count information. The cycle history moves forward in time one cycle at a time. In other words, any time base of, say, 10 cycles, is always based on the most recent 10 cycles.

5.7.8.1 Turning On Profiling

Profiling does not operate until it is enabled in the defaults window. Open the defaults window and type a 1 into the profile enable field. (See the discussion and illustration of the defaults window in the "Set-Up" portion of this chapter.)

5.7.8.2 Setting A Profile Range

The PA command, given in the command window, adds or replaces a range of code to the profile list. The ranges in the list are identified by number. The general form of the command is

```
> PA range# address
```

where *range#* is a number identifying the range and *address* is an address or range of addresses in program memory. For example, to define profile range number one as the addresses of the interrupt vector table in the ADSP-2101, the command would be

```
> pa 1 pm[0x0], pm[0x1b]
```

You may also enter a range directly into the profile window. Simply position the cursor at a line in the list and type in the addresses as they appear in Figure 5.21.

The profile ranges are numbered from 1 to 22; you may define a total of 19 to add to the list.

In addition to the ranges you define, there are always three other items in the profile list (see Figure 5.21): other, idle, and extra cycles. The "other" line collects data on all code outside the defined range(s)."Idle" shows the profile data for all processor IDLE instructions executed. "Extra cycles" displays the profile data for any extra cycles executed due to memory wait states or multi-cycle instructions.

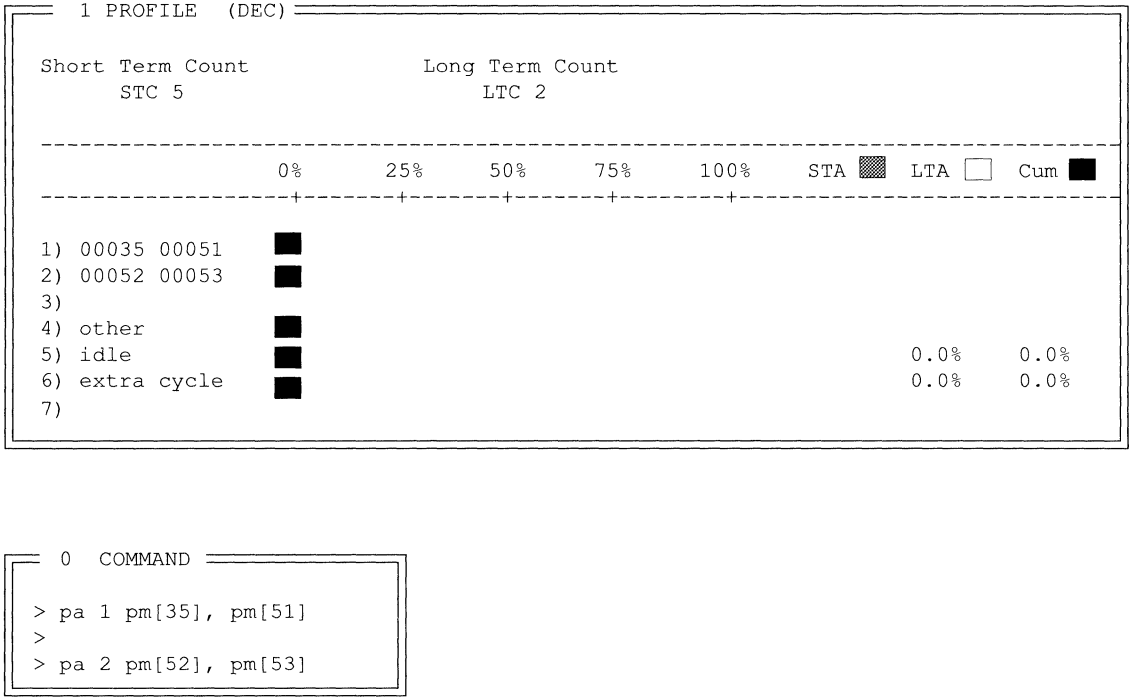


Figure 5.21 Profile Window

5.7.8.3 Deleting Profile Ranges

The PD command, given in the command window, deletes a single profile range by number. Its form is just

```
> PD range#
```

The range may also be removed from the list by deleting the addresses directly in the profile window.

The PC command, "profile clear", deletes all ranges and associated data. It takes no arguments and its form is

```
> PC
```

5.7.8.4 Resetting Profiling Data

The PR command, given in the command window, clears all profiling data but leaves the current definition of ranges undisturbed.

5.7.9 Setting Time Bases

The profiling operation captures information about the amount of time spent executing the specified ranges of your program, and the amount of time spent executing the rest of your program (the "other" data). The basic mechanism for showing this information is a breakdown of the percent of execution time spent in each profile range. Figure 5.21 shows a typical profile window with two ranges defined.

The profiler expresses this information in two different counting periods and a cumulative total. The total is always active. The rightmost column labeled Cum shows the cumulative average: the percent time spent in each range since you last reset the accumulated data. The area under the percentage signs (0%, 25%, 50%, etc.) displays a simple histogram of the same information.

5.7.9.1 Short Term Count (STC)

The smaller counting range is the short term count. This parameter defines how many execution cycles to look back at, starting at the present point in time. The PP ("set profile parameter") command, given in the command window, sets the number of cycles to include in the short term count. The syntax of the PP command is

```
> PP STC num_cycles
```

where STC identifies the short term cycle count and *num_cycles* is some chosen number of cycles. For example, the command

```
> pp stc 10
```

sets the short term count to 10 cycles. This means that the short term averages (seen in the profile window under the "STA !" column) reflect the percent of execution time spent in each address range during the last 10 cycles, that is, the last short term count interval.

The short term count can be set directly in the profile window by positioning the cursor next to the "STC" and typing in the desired value. To move the cursor to top portion of the window, key a ^T. (^T toggles the cursor between the profile parameters and the profile range list.)

5.7.9.2 Long Term Count (LTC)

The other counting range is the long term count, which is a multiple of the short term count. The profile parameter command also sets this value with the form

```
> PP LTC num_shortcounts
```

where LTC identifies the long term count and *num_shortcounts* is the number of short term count intervals to include in the long term average. To set LTC to 3, for example, you would give the command

```
> pp ltc 3
```

The long term average percentages are displayed under the column heading "LTA #" in the profile window.

The long term count may also be set directly in the profile window by positioning the cursor next to the "LTC" and typing in the desired value. To move the cursor to top portion of the window, key a ^T. (^T toggles the cursor between the profile parameters and the profile range list.)

Note again that the long term count is a multiple of short term counts. In other words, a short term count of 10 cycles identifies the number of cycles over which to produce the short term average percentages. A long term count value of 3 identifies the number of (10-cycle) intervals over which to produce the long term average percentages. Thus, in this example, the short term average looks back at the last 10 cycles, while the long term average looks back at the last 30 cycles (see Figure 5.22, on the next page).

The profiling mechanism counts all cycles including cycles spent executing the IDLE instruction and extra cycles required when the processor must perform two off-chip memory fetches or when the processor is executing wait states for slow memory access.

5 Simulator Functions

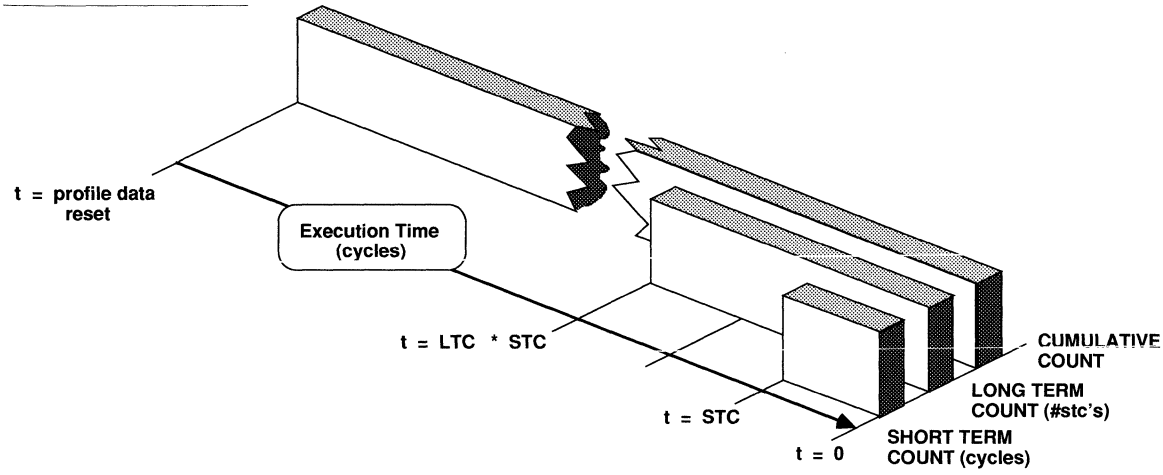


Figure 5.22 Short Term, Long Term and Cumulative Profiling Time Bases

5.8 EXITING & SAVING A SIMULATOR SESSION

You can quit the Simulator and, if you wish, save the state of the simulation to continue working from the point you leave off. You can also save selected ranges of memory as detailed earlier in this chapter in "Inspecting A Memory Location."

Note that the display configuration when you quit is written to the file DD.WIN and is automatically loaded as the default display when the Simulator is restarted. If you want a specific starting display each time the Simulator is restarted (other than the last configuration saved), you must use the –w switch with the filename of your preferred configuration when invoking the Simulator.

5.8.1 Saving Simulation State

In addition to the D command, described under inspecting memory in this chapter, the Z token, which represents the state of the Simulator, may be written to a file or read from a file using the > and < operators as pipes. For example, to write the current state of the Simulator to the file *coffebrk*, the command is

```
> z >'coffebrk'
```

Note that the filename is enclosed in single quotation marks. Similarly, to restore the state of the Simulator after rebooting, resetting or performing any other operations, you would read it back with the command

```
> z <'coffebrk'
```

which restores the Simulator to the previous state.

5.8.1.1 What Is Saved

The following items are saved when the state of the Simulator is saved:

- contents of all registers
- contents of all memories
- defaults as listed in the defaults window
- symbols
- breakpoints

5.8.1.2 What Is Not Saved

As of release 2.01X, the items listed below are not saved and must be recreated when the saved state is restored:

- all general, break, and watch expressions
- aliases not stored in a startup file
- profile address ranges and data
- file pointers to data files used to simulate SPORT and I/O port activity

5.8.2 Quitting The Simulator

Quitting (with or without saving anything) is very simple. From the command window give the command

```
> Q
```

A double-check window appears asking you to confirm. Keying ^Q while working in any window will quit the Simulator in the same fashion.

To bypass the double-check window, type the full command

```
> QUIT
```

and the Simulator exits without further interaction.

5 Simulator Functions

5.9 MISCELLANEOUS FEATURES

The Simulator has several useful capabilities that do not map directly into the breakdown of functions. These capabilities are described here.

5.9.1 Executing Operating System Commands

You can temporarily suspend the Simulator and exit to the operating system of your computer with the SH ("shell") command. Because the Simulator requires a large amount of memory, you may not be able to execute any other programs from the operating system prompt on a PC. You should be able to rename files, copy files and inspect directories with no difficulty. When you are done, typing EXIT returns you to the Simulator. No values in the simulation are changed by this operation.

For this command to work properly on the PC, your COMMAND.COM (DOS) file must be in the current or top-level directory, or a PATH statement must be defined leading to the directory in which it is contained.

5.9.2 Executing ADSP-2101 Instructions Directly

In addition to the on-line assembly of instructions intended to patch a program loaded in memory, you can enter and execute any ADSP-2101 instruction directly without loading it into program memory or otherwise disturbing the contents of your program. This is accomplished with the V command, given in the command window. The general form is

```
> V instruction
```

where *instruction* is any valid ADSP-2101 instruction, with or without a terminal semicolon. For example, to move the contents of register AY0 to register AX0, you would enter

```
> v AX0 = AY0;
```

5.10 SUMMARY OF COMMANDS & CONTEXTS

Table 5.4 below lists the keys and control key sequences which allow you to navigate between different windows or within the active window. These keystrokes can be used in *any window*. Table 5.5 lists all the commands that are entered in the command window. Table 5.6 defines the control key sequences used only in particular windows. Table 5.7 provides a cross reference to show which of these control key sequences can be used in which windows.

Multiple commands may be given on one line in the command window, separated by semicolons, as in

```
> L 'filename' ; J symbol 'command' ; D address
```

^W	display main menu of window configuration actions
ESC	exit a menu without making a selection
^Z	move to next (consecutively numbered) window
^X# (Return)	move to window number # (^X (Return) or ^X0 (Return) moves to command window)
Arrow keys, PgUp, PgDn	scroll through text in a window

Table 5.4 Window Navigation Controls

A *addr instr*	assemble instruction at address
B	list breakpoints, break expressions, and break change expressions
B *addr*	set breakpoint at address
BC *expr*	set break change expression
BD *addr* or *number*	delete breakpoint, break range, break change or break expression
BE *expr*	set break expression
BR *range*	set break for address range
CR	chip reset with boot page 0 load
D *addr* or *range* [>'*file*']	dump (display) contents of memory
E *addr* or *range* *expr*	enter value of expression into memory
E *addr* <'*file*'	load memory from file
F *range expr*	find value of expression in range
G [*addr*]	start program execution

5 Simulator Functions

I *int# min max*	cause periodic interrupt
J *symbol 'command'*	alias a command string
J	list aliases
J >*'file'*	dump aliases to file
JD *symbol*	delete alias
K *window# addr*	display address (constant or symbol only) in window#
L *'file'*	load program into memory
LR *'file'*	load boot PROM image file into boot memory
O *addr* [<*'file'*] [>*'file'*]	open I/O port at addr and assign I/O files
O *addr*	close I/O port at addr
P SPORT# [<*'file'*] [>*'file'*]	open serial port# 0 or 1 and assign I/O files
PA *range# addr, addr*	add or replace an address range in the profile window
PC	clear all profile ranges and data
PD *range#*	delete range# in profile window
PL *range decimation*	plot memory
PP STC or LTC *#cycles*	set profile parameter
PR	reset profile data; retain defined ranges
Q	quit Simulator, with verification from user
QUIT	quit Simulator, without user verification

R *reg expr*	set register equal to value of expression
RE	reset chip without boot page load
RUNFAST	start program execution, no halt on key hit
S [*number*]	single step program execution
SH	temporarily exit to operating system
T *window#*	toggle tracking on/off in window#
U *addr* or *range* or *reg*	undefine contents of an address, range, or register
V *instr*	assemble and execute instruction
W *addr* or *range*	set watch point
W	list all watch points and watch expressions
WD *number*	delete watch point# or watch expression#
WE *expr*	define watch expression
X *symbol*	give address of symbol
Y [>'*file*'] [<'*file*']	save/restore display configuration (see "Saving A Rearranged Screen" in Chapter 6)
Z [>'*file*'] [<'*file*']	save/restore Simulator state
? *expr*	evaluate expression
?+ *expr*	add expression to expressions window
?– *number*	delete expression# from expressions window

Table 5.5 Command Window Commands

5 Simulator Functions

^B	set breakpoint at cursor location in program memory window
^R	reset (delete) breakpoint at cursor location in program memory window
^D	delete field in active window
^U	undelete (restore) field in active window
^Y	move field in active window
^E	toggle numeric display of (active) window contents between HEX and DEC
^G	go to (prompted for address to be displayed)
^S	choose how many lines (instructions) to trace in trace window
^T	toggle tracking on/off in active window*
^Q	quit Simulator

Table 5.6 Window-Specific Control Key Sequences

* ^T toggles between display of primary and secondary register banks in the register window. ^T moves the cursor between profile parameters and profile ranges in the profile window. When the cursor is positioned at the profile parameters, the parameters (STC and LTC) may be set by typing in the desired values. The PP command is echoed in the command window.

		Control Keys									
		^B	^R	^D	^U	^Y	^E	^G	^S	^T	^Q
Window	data mem						✔	✔		✔	✔
	program mem	✔	✔					✔		✔	✔
	program mem data						✔	✔		✔	✔
	boot code							✔			✔
	boot data						✔	✔			✔
	cross reference										✔
	register			✔	✔	✔	✔			✔*	✔
	flag			✔	✔	✔					✔
	stack			✔	✔	✔	✔				✔
	status register			✔	✔	✔					✔
	I/O status										✔
	SPORT register			✔	✔	✔	✔				✔
	SPORT status										✔
	control register			✔	✔	✔	✔				✔
	break expressions						✔				✔
	breakpoints										✔
	expressions							✔			✔
	profile						✔	✔		✔*	✔
	trace						✔		✔		✔
	default						✔	✔			✔
	help							✔		✔	

Table 5.7 Window to Control Key Sequence Cross Reference

* ^T toggles between display of primary and secondary register banks in the register window. ^T moves the cursor between profile parameters and profile ranges in the profile window. When the cursor is positioned at the profile parameters, the parameters (STC and LTC) may be set by typing in the desired values. The PP command is echoed in the command window.

5 Simulator Functions

Simulator Configurations 6

6.1 INTRODUCTION

The previous chapter describes the functions of the Simulator. This chapter describes how to customize the "look and feel" of the Simulator for your everyday needs.

With the release of version 2.0X (and after) the Simulator provides a customizable user interface. You can change and store the following items:

- The location of individual fields within windows
- The location and organization of windows on the screen
- The names used to invoke commands and the required order of arguments
- Any sequence of commands (including aliased commands)

The best way to tailor the Simulator to your requirements is to begin using it and build up custom screens and commands as you go. At some point, perhaps as soon as a few hours after you begin, you can organize all the custom screens and commands into a clean set of external files. Thereafter, you can invoke the Simulator with the appropriate startup file identified and the Simulator appears automatically in your desired configuration.

The Simulator uses two types of external configuration files: displays and scripts. Display files (file extension .WIN) store the look and layout of a particular set of windows, one to a file. Each display can be stored and then recalled, in the desired configuration, with a simple command. Scripts (default filename is STARTUP) are text files of command window inputs typically storing command aliases you create.

6 Simulator Configurations

As shipped to you, the Simulator package includes a sample STARTUP file (named EXAMPLE) and a number of sample display windows (.WIN files). Rename EXAMPLE to STARTUP to automatically invoke it or name it explicitly (with the –s switch) when you start the Simulator.

6.2 CONFIGURING SCREENS & WINDOWS

The tools for configuring an individual window are briefly described in the previous chapter. This section spells them out in greater detail with an example.

(Note: ^ denotes the control, or CNTL, key.)

6.2.1 Opening Windows

Windows are opened by keying ^W to display the menu shown in Figure 6.1 and selecting OPEN by typing the letter "O" or pressing Return (since OPEN is the default selection on this menu).

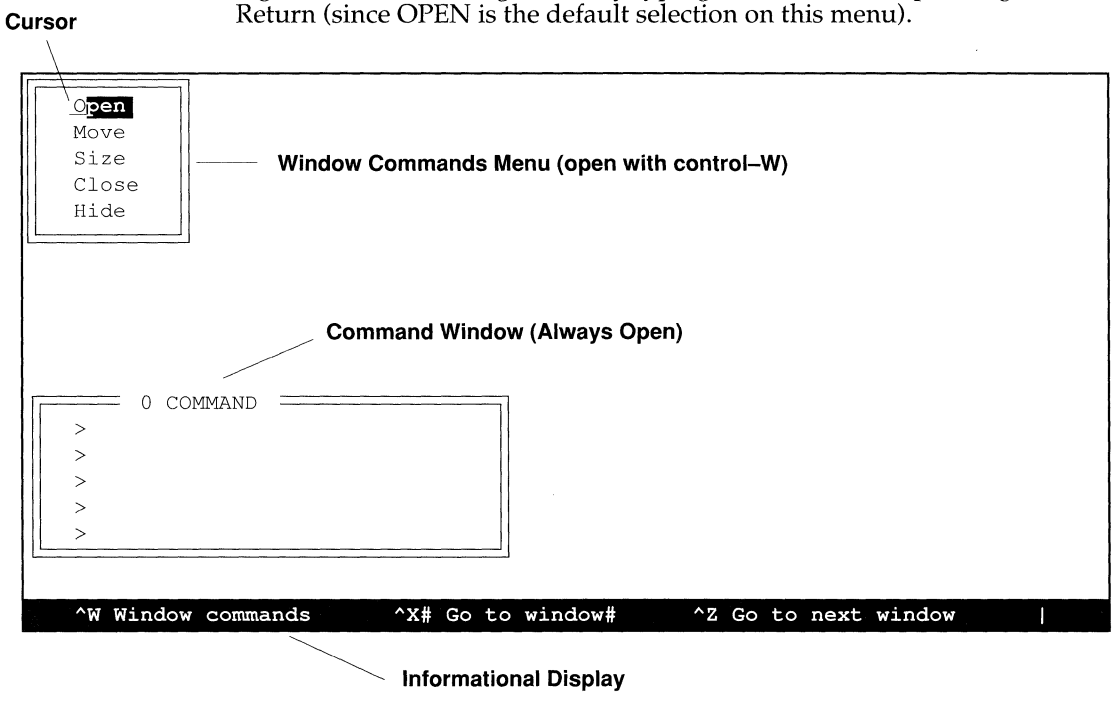


Figure 6.1 Main Menu For Configuring Windows

Doing this displays the window selection submenu shown in Figure 6.2. Select the desired window; our example uses the register window. You select the register window by moving the cursor down with the arrow keys and pressing Return or by keying the menu letter ("D").

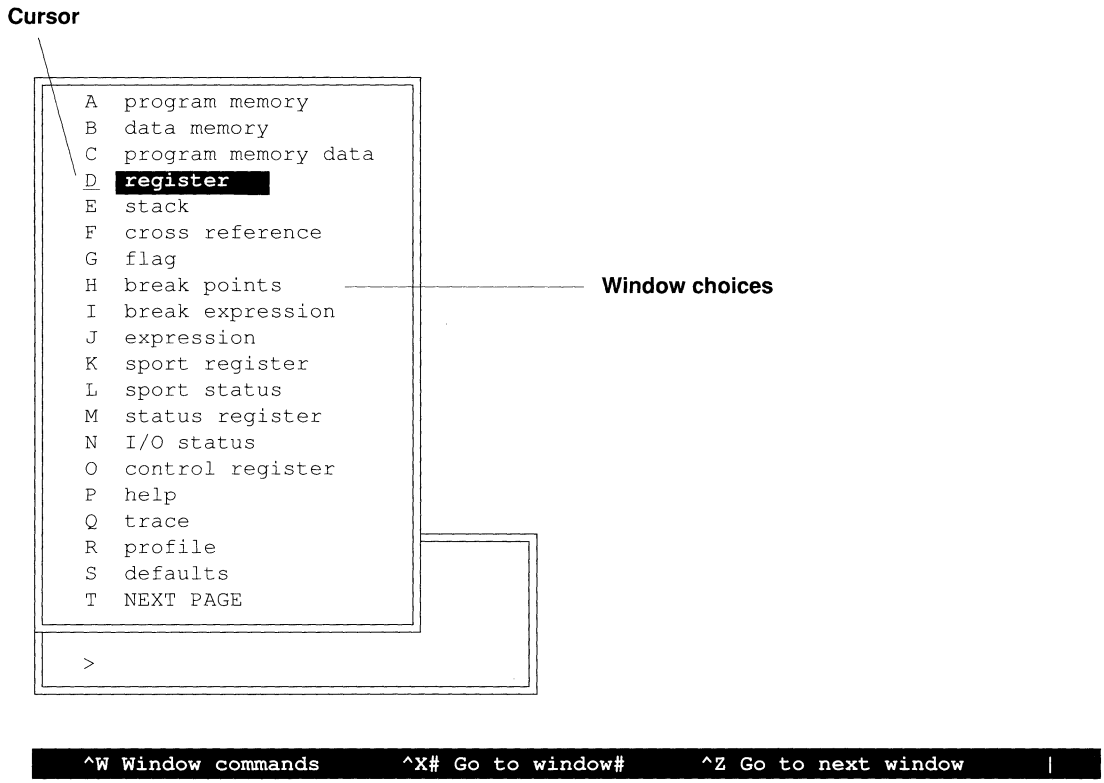


Figure 6.2 Window Selection Submenu (with Register Window selected)

6 Simulator Configurations

Figure 6.3 shows the default register window layout. This is the starting point for rearranging the fields of this window.

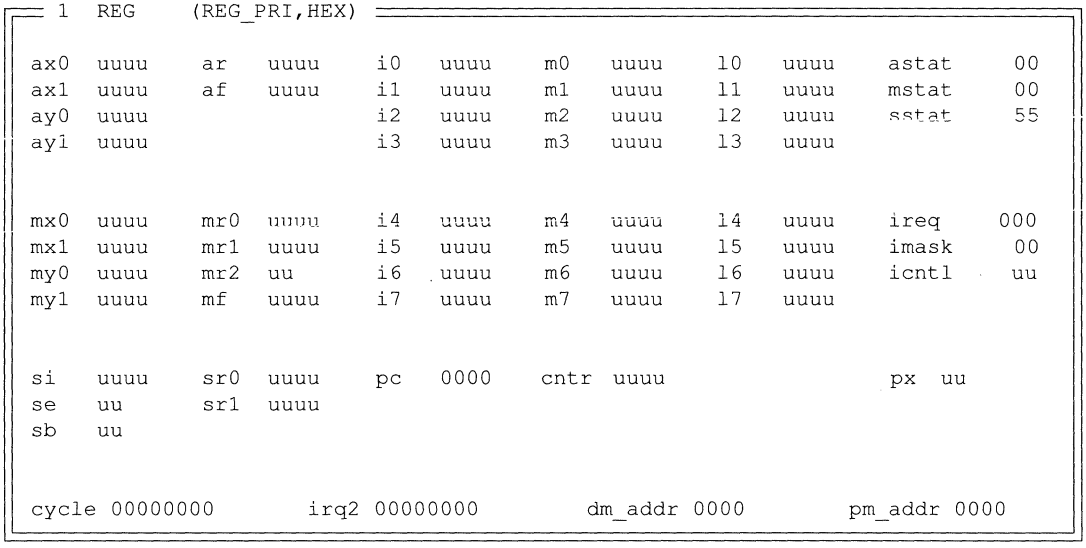


Figure 6.3 Default Register Window Layout

6.2.2 Selecting, Deleting & Rearranging Fields In A Window

Note that only the windows displaying individual fields, like the register window, can be rearranged. The internal layout of memory windows and informational windows (like the breakpoints window) cannot be altered.

In the program example used in this manual there are some registers in the processor which are never used. For example, only the SI register in the Shifter is used (as a temporary holding register for signal data) and none of the ALU registers are used. Likewise, only some of the DAG registers are used. For the purposes of illustration we are going to delete unused registers from our display and rearrange the remaining registers for compactness.

The first step is to make the register window the active window, if it is not already. Key ^Z until it becomes the active window or key ^X and the number of the register window followed by Return. The cursor appears in the active window.

Move the cursor with the arrow keys until it is over the SE field, one of the fields to be deleted. Key ^D to delete this field and it disappears from the display. Now move the cursor and delete the SB, SR0, and SR1 registers in the same way. You can go on to delete all of the ALU registers and the unneeded DAG registers: I2, I3 and I5-7, M1-3 and M5-7, and L1-3 and L5-7. After these deletions, the register window looks like Figure 6.4.

```
 1  REG  (REG_PRI,HEX)

                          i0   uuuu    m0   uuuu    l0   uuuu    astat    00
                          i1   uuuu                              mstat    00
                                                                 sstat    55



mx0  uuuu    mr0  uuuu    i4   uuuu    m4   uuuu    l4   uuuu    ireq    000
mx1  uuuu    mr1  uuuu                                          imask    00
my0  uuuu    mr2  uu                                            icntl    55
my1  uuuu    mf   uuuu


si   uuuu                pc  0000    cntr uuuu                px uu


cycle 00000000       irq2 00000000       dm_addr 0000       pm_addr 0000
```

Figure 6.4 Example Register Window with some registers deleted

6 Simulator Configurations

If you accidentally delete a register, it can easily be restored ("undeleted"). Move the cursor to a blank spot in the window and key ^U. A menu drops down that lists all the deleted registers. Move the cursor along the menu and press Return to restore any register. Press ESC to abort the operation. Note that the contents of the deleted registers are also displayed. If a deleted register has a value other than undefined, the value is visible in this menu.

Now we can rearrange our pruned-down set of registers for a more compact display. Move the cursor to the MX0 register field. To move any field you select it for moving with ^Y, move it using the arrow keys, and deselect it with another ^Y or Return. Move the MX0 field up to the top line of the window this way.

Repeating this procedure we could rearrange all the fields of this register window example until the entire window looked like Figure 6.5.

```
 1  REG    (REG_PRI,HEX)

mx0  uuuu     mr0  uuuu     i0   uuuu     m0   uuuu     l0   uuuu     astat   00
mx1  uuuu     mr1  uuuu     i1   uuuu                                   mstat   00
my0  uuuu     mr2  uu       i4   uuuu     m4   uuuu     l4   uuuu     sstat   55
my1  uuuu     mf   uuuu

si   uuuu                   ireq  000    imask  00     icntl  uu     px  uu

cycle 00000000              pc  0000     cntr   uuuu       pm_addr  0000
irq2 00000000                                              dm_addr  0000
```

Figure 6.5 Example Register Window with registers rearranged

Now you can resize the window outline, bringing up the bottom edge to make that space on the screen available for other windows. Key ^W to display the main menu and select SIZE from it. As you press the Up arrow, the lower edge of the window moves up on the screen. Press Return to end the sizing operation. The final version of this window might look like Figure 6.6.

```
 1  REG    (REG_PRI,HEX)

mx0  uuuu     mr0  uuuu     i0   uuuu     m0   uuuu     l0   uuuu     astat   00
mx1  uuuu     mr1  uuuu     i1   uuuu                                  mstat   00
my0  uuuu     mr2  uu       i4   uuuu     m4   uuuu     l4   uuuu     sstat   55
my1  uuuu     mf   uuuu


si   uuuu                   ireq  000    imask  00     icntl  uu     px  uu


cycle 00000000              pc  0000     cntr   uuuu       pm_addr  0000
irq2 00000000                                              dm_addr  0000
```

Figure 6.6 Final Register Window Arrangement

6.2.3 Saving A Rearranged Screen

If you change the display without saving your custom register window, all the work of creating it is lost. To save a new screen configuration with a desired set of windows opened, resized, and internally reconfigured, you must store the screen in a file. The Simulator token Y stands for the display and the greater than and less than symbols are directional pipes. The display files are given the default file extension .WIN. To save the current display (such as our example in Figure 6.6) enter this command in the command window:

```
> y >'myscreen'
```

This stores the current display configuration in the file MYSCREEN.WIN in the default directory.

6 Simulator Configurations

You may recall this or any other display configuration with the command

```
> y <'filename'
```

where *filename* is the main filename of a screen/windows file.

Note that the .WIN file stores the complete display, not just the contents of one window. You must create the complete constellation of windows you want and then store the entire display. Loading the Simulator display from a .WIN file overwrites the complete screen, not just part of it.

When you quit the Simulator, the last screen configuration is saved to the file DD.WIN, which becomes the default display the next time you startup. If this display is the only one you want, there is no need to use the –w *window* switch or the Y command. If several different custom screens are desired, then you should save each one and use the –w switch to load the startup screen and the Y command to recall others.

6.3 COMMAND ALIASES

A command alias is a character string (plus any required arguments) which replaces one of the Simulator's native commands. The J command creates the alias. The syntax of this command is

```
> j alias 'command'
```

where *alias* is a symbol which will subsequently stand for the Simulator command enclosed in single quotation marks. Arguments are passed using a dollar sign token as in $1, $2, etc.

For example, to create a special command to set the PC, you could type:

```
> j setpc 'r pc $1'
```

Then, to set the PC to point at address four, you could enter:

```
> setpc 0x4
```

Note that when you enter and execute this aliased command, the command window echoes both what you type and then the "unaliased" version of the command. This feature can be used to double-check your alias and to correct errors in arguments.

You can create a command that calls up a previously saved display configuration. If the file MYSCREEN.WIN contains a customized arrangement of the register, program memory, program memory data, and data memory windows, along with a small command window, it can be recalled by giving the command:

```
> y <'myscreen'
```

This may be inconvenient to type each time you want this display, especially because the "<" symbol requires the Shift key on most keyboards. You can alias this command with the new command string "VIEW" by entering the following:

```
> j view 'y <"myscreen"'
```

Note that the filename, *myscreen*, is enclosed with double quotes; nested quotation marks must be double, inside single.

6.3.1 Managing Aliased Commands

The J command, given with no arguments, lists all currently defined aliases in the command window. This is useful for managing command aliases as they become more numerous. Aliases are listed in the command window in the form

```
symbol = command
```

where *symbol* is the name you assigned as the alias and *command* is the actual command executed by the alias.

The JD command deletes an alias. The form of this command is

```
> JD symbol
```

where *symbol* is a currently defined alias name. The alias is deleted from the list.

6 Simulator Configurations

You may also pipe the list of currently defined aliases to a file for reference and editing with the command:

```
> J >'filename'
```

The contents of the text file created by this operation are actual J commands, like those executed to create the alias, such as:

```
j setpc 'r pc $1'
j view 'y <"myscreen"'
```

This file can be directly read back into the command window, using the redirection operator, as in

```
> <'filename'
```

which takes the contents of the named file as keyboard input to the command window until the end of the file is reached. This file can be also incorporated in a startup file, as described in the next section.

6.4 THE STARTUP FILE

The Simulator is a highly flexible tool. By designing custom display configurations and aliasing both built-in Simulator commands and additional Simulator commands you can create the debugging environment that best serves your purposes.

Embedding this all in the STARTUP file allows you to combine initialization, housekeeping, customized displays, and commands into one reconfigured Simulator.

The sample startup file EXAMPLE contains a large set of aliased commands. Examine the contents of this file to determine if you want to use any or all of the commands. You may choose to import all of the file's contents into the Simulator; to do this, rename EXAMPLE to STARTUP to automatically invoke it or name it explicitly (with the –s switch) when you start the Simulator.

Here are some lines out of a possible STARTUP file with a comment about each function performed. (Comments are not permitted in the actual STARTUP file.)

Line in STARTUP	*Function*
`j reg 'y <"wreg"'`	The new command string REG now calls up the predefined display stored in the file WREG.WIN.
`j help 'y <"whelp"'`	The new command string HELP now calls up the predefined display stored in the file WHELP.WIN.

`j setdm 'e dm[$2]/$3 $1'`	The new command SETDM (similar to earlier releases of the ADSP-2100 Simulator) accepts arguments in a different order. Instead of
	`e dm[start]/length value`
	you now enter:
	`setdm value start length`

`l 'main'`	This is an initialization command, loading your default program (MAIN.EXE) and symbol table (MAIN.SYM) files.
`p 0 >'serout' <'serin'`	This is another initialization command, opening data files used to simulate/ capture the I/O of serial port zero.

There is almost no limit to what can be accomplished via a STARTUP batch file. You can create specialized keyboard scripts files that actually execute and test your programs. The ADSP-2101 Simulator strives to deliver a tool-rich environment which can be configured to support many different requirements.

C Compiler 7

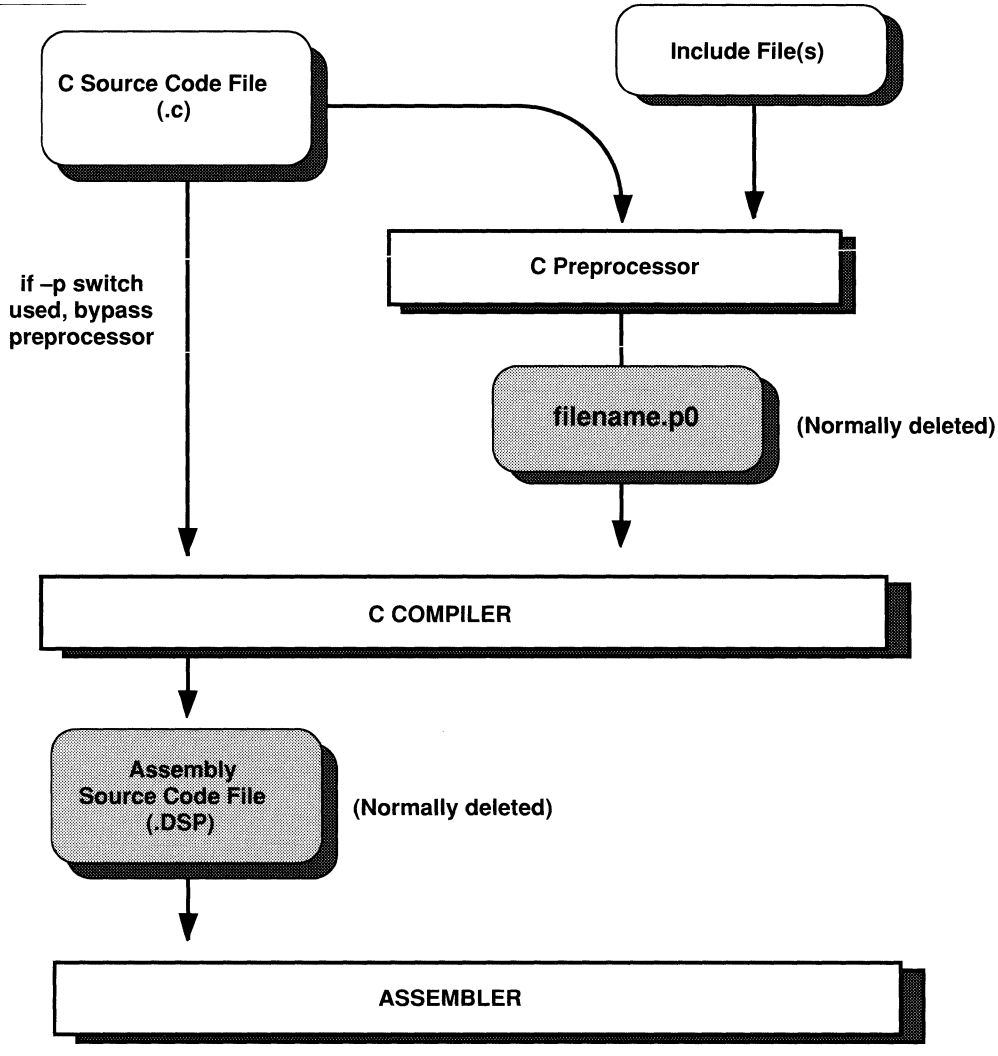
7.1 ADSP-210X C LANGUAGE SYSTEM

The ADSP-210X C language system allows programmers familiar with the C programming language to write and compile C programs to be run on the ADSP-2100 family of processors. C programs can be written, compiled and executed without extensive knowledge of the ADSP-2101 architecture. However, because of the unique architecture of the ADSP-2100 family, C programmers must understand various limitations detailed in this chapter in order to interface C to ADSP-2101 code and to optimize the output of the C Compiler. Nevertheless, the ability to program in C means that working ADSP-2101 systems can be created with fewer development resources. The C Compiler supports inline assembly code, conditional compilation, include files, and conforms to the draft ANSI standard except as noted in this chapter. Differences from the standard are summarized in Appendix D.

The ADSP-210X C language system consists of two primary modules, the preprocessor and the Compiler. The preprocessor reads directives beginning with the pound sign (#) and takes the actions necessary to resolve them. The simplest example is the *#include* directive. The preprocessor opens and inserts the contents of the requested include files, removing the *#include* directive from its output file.

The Compiler reads the output of the preprocessor and produces ADSP-2101 source code, as shown in Figure 7.1, on the next page. This source code is then assembled with the ADSP-2101 Assembler and is later linked with the ADSP-2101 Linker. (The Linker is not automatically invoked.) The ADSP-210X C language system must be used with Cross-Software Modules of Release 2.0 or after. Errors will occur if you try to assemble the output of the C Compiler with previous releases of the Assembler.

7 C Compiler



The Assembler List file (.LST) will contain C source in comment lines if an interlist merge file was requested.

Figure 7.1 C Compiler I/O

7.1.1 README File

The C Compiler is accompanied by a README file on the delivery media. You should consult this file for supplemental information about the files supplied, known bugs and other recent information for use and installation.

7.1.2 C & The ANSI Standard

At the time of this publication, there is an ANSI draft standard, X3J11, under consideration for the C programming language. The C language system conforms to the ANSI draft standard except as noted explicitly in this manual. Appendix D summarizes all exceptions. Any references to "standard C" are references to this draft standard. You may request a copy of the ANSI draft standard from ANSI. See Appendix D for the address.

This chapter is not a description of or reference document for the C programming language. For additional information about C you may want to consult the two books listed below. There are many other books available on the subject as well.

Harbison and Steele, *C: A Reference Manual*. Prentice-Hall, 1987.

Kernighan and Ritchie, *The C Programming Language*. Prentice-Hall, 1978.

7.1.3 Upper and Lower Case Usage

The ADSP-2101 Assembler and its source code is traditionally case-insensitive: upper and lower case versions of names and reserved words are treated identically. The C language, however, as practiced and as defined in the draft standard, is a case-sensitive environment. A lowercase identifier is not the same as an uppercase identifier. The Assembler's –c switch toggles the Assembler's interpretation of upper and lower case symbols. The default is case-insensitive, like previous versions of the Assembler. The Assembler must be made case-sensitive (with the –c switch) when assembling C-generated programs.

This must be done in order to link together code modules originally written in C with modules written in assembly language.

7.2 COMPILING

The C Compiler processes standard ASCII text files only. Do not use editors that produce files with special characters or formatting commands.

7 C Compiler

7.2.1 Filename Usage

The C language system may create several files. In the default operation, the output name is based on the main name of your input file. This can be changed when the Compiler is invoked (see below). The usage of these files is shown in Figure 7.1 at the beginning of the chapter; the naming conventions are summarized here.

Input C source filename:	*Compiler looks for*:
filename	filename.c
filename.c	filename.c
filename.*ext*	filename.*ext* (exactly as given)

File	*Module producing it*	*Extension appended*
preprocessed C source*	C preprocessor	.p0
ADSP-2101 source*†	C Compiler	.dsp
List file†	ADSP-2101 Assembler	.lst

* Normally deleted by C Compiler, can be preserved with switch
† Contains C source in listing if requested, otherwise standard listing

7.2.2 Invoking The C Compiler

The Compiler is invoked by entering the name of the executable Compiler file with the name of your source file and any desired switches as arguments. The brackets below indicate that switches and *outfile* are optional.

cc2101 *infile* [–switch ...] [*outfile*]

Typically, all switches can be omitted:

cc2101 *test.c*

The output files are normally given the same file name as the input file. By giving a second file name when the Compiler is invoked (shown above as *outfile*) the output file(s) use this name, instead of the input file name. Switches may come anywhere after "cc2101."

If you invoke the Compiler with no arguments at all, it displays a brief summary of the switches as explained below.

There be any number of switches from the group shown in Table 7.1. Switches that are not self-explanatory are detailed in the following sections. Note that the Compiler can generate a ROMable system.

Switch	*Result*	*Default*
–a	Do not invoke Assembler	Assembler invoked
–abs=#	Specifies absolute memory location	Location determined by Linker
–b#[#...]	Specifies boot page(s) destination	No boot page information generated in assembly module
–crom	Code placed in ROM	Code in RAM
–D*var*[=value]	Define a variable name for macros	No such definition
–e	Call FP emulation routines	Inline code
–gpm	Force all globals into PM	Data memory (DM), or as specified in initial declaration statement
–I=path	Specifies search path for include	Current directory & ADII path
–lpm	String literals & switch tables in PM	Placed in DM
–lrom	String literals & switch tables in ROM	String literals in RAM
–m	Produce a merged listing file and request Assembler .LST file	No merged listing file created
–p	Do not invoke preprocessor	Preprocessor invoked
–pp	Invoke preprocessor only	Compiler also invoked
–pmstack	Stack in program memory (PM)	Stack in DM
–w	Suppress warning messages	Display warning messages
–0	Save preprocessor output file	Delete this file
–1	Save Compiler output .DSP file	Delete this file

Table 7.1 Compiler Switches

7 C Compiler

7.2.2.1 –a Switch

The C Compiler normally invokes the Assembler directly. This switch cancels that invocation. When the C Compiler does call the Assembler, it uses a specific set of Assembler switches (listed below). If you are going to run the Assembler manually, it is important to use the exact same switches. A complete list of Assembler switches is shown in Chapter 3 of this manual. Here are the Assembler switches used by the C Compiler:

Assembler switch	*Meaning*
–c	Forces the Assembler to be case sensitive; always set by Compiler.
–s	No semantics checking on parallel instructions as described below. This switch is always set by the C Compiler, and must be manually set if you later attempt to assemble code generated by the C Compiler.
–r	Assembler deletes the assembly source code .DSP file (unless the Compiler's –1 switch is set).

The semantics checking of parallel instructions must always be relaxed. Certain of the emulation routines use parallel instruction forms that are flagged as semantically incorrect by earlier versions of the Assembler. See the discussion under "Multifunction Instructions" in the Instruction Set Reference chapter of this manual.

7.2.2.2 –abs = # Switch

This switch assigns a C-style constant to be used as an absolute memory location for module placement in program memory.

7.2.2.3 –b#[#...] Switch

This switch results in assembly code declaring the code module's boot page(s) destination. The information will be used by the Linker to create boot page images.

Multiple boot pages can be specified by adding on numbers. For example, to specify a destination of boot page zero, the switch would be

cc2101 *infile* –b0

and to specify destinations of boot pages zero, one, and two for the resulting module, the switch would look like this:

cc2101 *infile* –b012

7.2.2.4 –Dvariable [=value] Switch

The –D*variable*[=value] switch allows you to define a variable name for the preprocessor to use (such as in conditional compilation). You may optionally assign it a value.

7.2.2.5 –e Switch

A library of basic floating-point emulation routines is provided with the Compiler. The default is to compile these routines by inserting the ADSP-2101 source from the library directly into the output of the Compiler. This switch causes the Compiler to generate a CALL to these routines, which then terminate with a RTS (Return from Subroutine).

Inline code results in a larger program while calling subroutines requires some additional overhead and results in somewhat lower performance. Note, however, that some routines are so large (e.g. floating-point division) that they are *never* placed inline, regardless of this switch.

7.2.2.6 –gpm Switch

This switch forces all globals into program memory. It overrides any storage classes modifiers used in the declaration of globals.

7.2.2.7 –I = path Switch

This switch provides an additional way to specify a search path for include files (in addition to the ADII variable). The directories specified by –I are searched before those specified by ADII.

7.2.2.8 –lpm & –lrom Switches

These switches control the placement of string literals and the tables created for and used by the "switch" and "fastswitch" statements. Normally, these are placed in data memory, assumed to be RAM. You can switch either of these to store string literals and switch tables in program memory and/or in ROM.

7.2.2.9 –m Switch

For debugging and learning purposes, the Compiler produces a merged listing file (in the .DSP file) when this switch is used. In this file each C

source statement appears as a comment followed directly by the ADSP-2101 source it generates. An example fragment of a merged listing file is shown below. If the –m switch is used, the .DSP file (if not deleted) contains these comments and the Assembler passes them through, as it does any comments, into the Assembler output .LST file (which is generated by the Compiler's using the Assembler –l switch). For example:

```
!  i=3;                  /* C source line*/
   si=3;                 /* ADSP-2101 source line*/
   dm(i_)=si;            /* ADSP-2101 source line*/
!  j=k+1;                /* C source line*/
   ax0=dm(k_);           /* ADSP-2101 source line*/
   ay0=1;                /* ADSP-2101 source line*/
   ar=ax0+ay0;           /* ADSP-2101 source line*/
   dm(j_)=ar;            /* ADSP-2101 source line*/
```

7.2.2.10 –pmstack Switch

This controls the placement of the stack created for and used by the C environment. If the stack is in program memory, you must invoke the Linker with its –pmstack switch. The Linker switch is discussed in the Linker Chapter, Chapter 4.

Please refer to the discussion later in this chapter about the stack implementation for a complete discussion of the tradeoffs involved in stack location.

7.2.2.11 –0 & –1 Switches

These switches, which use the numerals zero and one, prevent the deletion of intermediate files produced by the preprocessor and Compiler. This may be useful for debugging.

The –0 switch results in the file *filename.p0* being left in place by the preprocessor.

The –1 switch results in the file *filename.dsp* being left in place by the Compiler. Files with the .DSP extension are ADSP-2101 source code for input to the Assembler. You may wish to inspect this file before assembling it. If you intend to optimize the output of the C Compiler, you must have this file.

7.2.3 Preprocessor Commands

The C preprocessor supports the complete ANSI draft standard set of options. Two directives with implementation-specific aspects are described here.

7.2.3.1 #pragma Directive

The #*pragma* directive is used in the C system. This is an implementation-dependent directive which is used in our implementation for passing inline assembly code. Inline code may be necessary for sections of your program that require features not directly supported in C, such as interrupt handling, the use of circular buffers, or optimization of certain computations. This directive must be used as shown below:

#pragma ADSP2100 (Toggles on/off processing of inline code)

For a complete example, see the section "Assembly Language Interface" in this chapter. Here is a simple example:

```
int someval = 8;
foo()
{
#pragma ADSP2100
SE = DM(someval_);          (ADSP-2101 assembly code)
#pragma ADSP2100
}
```

The preprocessor simply passes unchanged the text between the two directives on to the Compiler. The Compiler, in turn, removes the directive lines and passes the source code directly into its output file with no checking. The preprocessor need not be invoked to use the #*pragma* directive.

This example also shows the ADSP-2101 source code naming convention used by the C Compiler. An identifier used in C source appears in ADSP-2101 source with an underscore appended: *someval* becomes *someval_*.

7.2.3.2 #include Directive

Files may also be included by specifying a pathname when the Compiler is invoked, using the *I=pathname* switch.

The include function operates just as in other C environments. The search path for the files to include can be set using the ADII environment variable (which is a function of your computer's operating system, not the C Compiler). The Compiler first searches in the current directory; if the files cannot be located there, the path specified by ADII is searched. (For include directives using the form *filename*, the current directory is not searched.)

To define the ADII environment variable, execute a statement similar to the following examples, substituting the actual pathnames for your system where the dummy names are shown in italics below. The semicolon separates individual search paths. The final slash must be present. Do not include extra spaces.

IBM-PC Example:
SET ADII=*root**subdir**subdir*\;*root**nextsubdir**nextsubdir*\;

Unix (Sun) Example:
setenv ADII "/*root*/*subdir*/INCLUDE/;/*root*/*nextsub*/INCLUDE/;"

The maximum number of directories that can be specified with ADII is twenty. If ADII has not been defined in the system environment, the search terminates immediately after searching the current directory.

7.2.4 Linker Requirements

The Linker has a number of features controlled by command line switches given when the Linker is invoked. Complete information is given in Chapter 4 of this manual. Here we specify only the Linker switches that must be used with modules generated via the C Compiler.

The Linker –c switch must be used, and causes two things to happen. First, the Linker creates the artificial symbol

`____top_of_ram` (four leading underscores)

which is assigned the value of the highest available address in data memory (or program memory, see the discussion of the –pmstack switch below). Second, the Linker searches for and links in the C run time header, which is an assembly language file (filename *run_hdr*) provided with the Cross-Software System. The `____top_of_ram` symbol is used by the run time header to locate and initialize the stack.

The environment variable ADIRTH must be equated to a pathname identifying the directory which contains the run time header. This path is searched by the Linker; the run time header must be located and linked because it is used when running compiled C code. The pathname is a function of your operating system, and is determined by where you store the *run_hdr* file.

The Linker –pmstack switch, used in conjunction with the –c switch, forces the `____top_of_ram` symbol into program memory. So, if you

use the –pmstack switch with the C Compiler, you must use the –pmstack switch with the Linker as well.

7.2.5 Run Time Header

The C system includes the source file, *run_hdr.dsp*, and the assembled module, *run_hdr.obj*. You must link this object file with your other modules to create a working system. (Refer to the previous section "Linker Requirements.")

The run time header sets up the registers used to control the stack and calls the main routine in your program. It is loaded at absolute address zero and includes interrupt service routine calls. You should examine the file for detailed information about what it does. This routine may be altered as needed.

7.3 RUN TIME MODEL

The C language run time model is a stack-oriented machine, using the stack for parameter passing and local and temporary storage. The ADSP-2101 is a dual memory processor, with two data address generators (DAGs). DAG2 is used for stack addressing because it can address both program and data memory. DAG2 includes the length (L), index (I) and modify (M) registers four through seven, e.g. I4-I7. Specific register usage is given below.

You need to understand how the C Compiler maps C onto the ADSP-2101 architecture in order to create assembly language routines that interface to C programs.

The stack may be located in either program or data memory space. Because of certain constraints in the ADSP-2100 family architecture, locating the stack in data memory is usually more efficient. See the section on "Programming Hints" below.

7.3.1 Stack Implementation

The stack is implemented as a 16-bit wide push down structure, growing from high memory. There is a special variable identifying the top of memory which is evaluated by the Linker. (Remember, you must use Linker release 2.0 or later with the C Compiler.)

The default is to locate the stack in data memory unless program memory is specified with a Compiler switch (–pmstack). The Linker extracts the top of memory address from the .ACH Architecture Description file created by the System Builder module of the Cross-Software.

The stack is managed by a frame pointer and a stack pointer. Figure 7.2 shows the configuration of the stack during a typical call. The discussion of reserved registers, below, describes the use of the stack and frame pointer registers. Because the registers in the DAGs are 14-bit registers, the largest object size that can be put on the stack is 8K words.

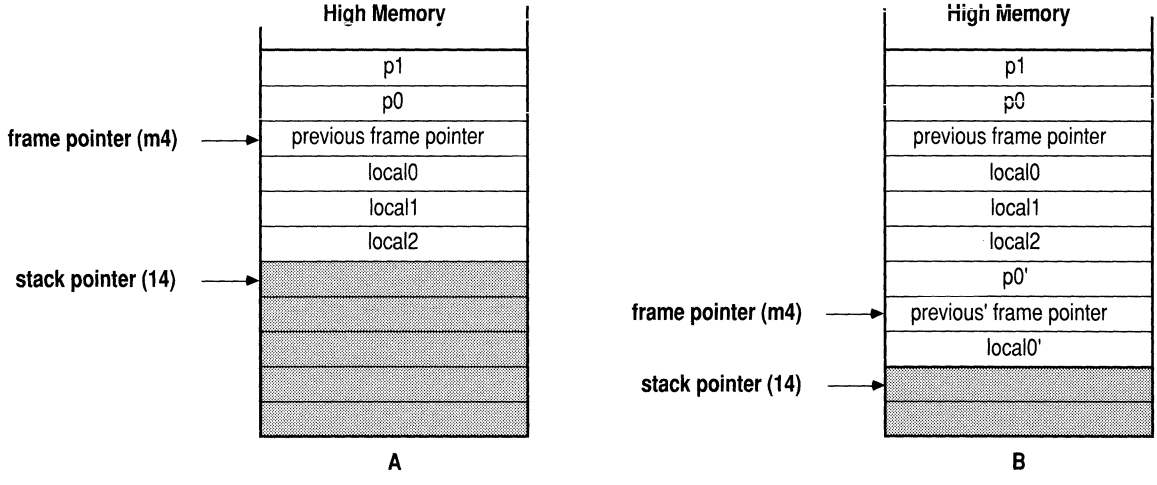


Figure 7.2 Stack Implementation in ADSP-2101 Memory Space

7.3.2 Register Use Limits

There are two types of register use limits. One group of registers is used by any code created by the C Compiler and must be restored exactly by any assembly language operations. These are *reserved/system registers* as listed below in Table 7.2. The second group consists of registers that might be used by C code to store data values. These *restricted/data registers* may be used by assembly language routines and may be restored selectively, based on a careful study of the register usage in the calling or context code module. They are listed in Table 7.3.

Reserved/system register	*Use*
M4	frame pointer
M5, M1	must contain +1
M7, M3	must contain –1
M2	must be zero
L0–L7	must be zero; circular buffers not supported in C.
I4	stack pointer
M6, M0, SI, SE	used as scratchpad registers

Table 7.2 Reserved/System Registers

Restricted/data registers	*Use*
AY0, AY1, AR	data storage
MX0, MX1, MY0, MY1	data storage
MR0, MR1	data storage
SR0, SR1	data storage
I0–I3, I5, I7	data storage

Table 7.3 Restricted/Data Registers

The C language does not support the concept of circular buffers. If your inline code or assembly language routine uses circular buffers you must set them up, use them, and reset the L registers to zero before returning to C.

In general, you must save and restore any registers from either group if you use that register. In practice, you must save and restore any register from the reserved/system group and you may determine (by examining a merged listing file) which registers from the restricted/data group are used by earlier routines.

DAG register usage in the code generated by C programs is quite different from the ADSP-2101 source code environment. This is because all I registers are automatically post-modified by the M register with which they are used. In C-generated code, M4 always contains the current frame pointer value and I4 contains the stack pointer value.

Because the I registers are post-modified, an I register is not used as the frame pointer. Instead, M4 is used. Available I registers (I5-I7) are set up with an offset value from the frame pointer to access parameters and local variables on the stack. Consequently, the frame pointer register usage is virtually the opposite of the standard ADSP-2101 use. The M4 register

contains an address value and the Ix register contains the offset or modify value. This allows the M4 value to remain unchanged by the built-in post-modify operation.

Stack pointer maintenance uses I and M registers in a more conventional style. I4 always points to the *next* available location on the stack, again due to the post-modify behavior of the ADSP-2101 DAGs. M5 is used exclusively to increment the stack pointer by one, which is equivalent to popping one value (the stack grows down from high memory). M7 is used to decrement the stack pointer address, equivalent to pushing.

M6 is used in a variety of situations, primarily in conjunction with popping multiple locations off the stack. If, for example, there were ten local variables on the stack to be popped upon return, the M6 register would be loaded with the value ten and paired with the I4 register to modify it by this amount in a single instruction.

7.3.3 Interrupts

Interrupts are not directly supported in the C environment. They must be handled via the *#pragma ADSP2100* directive and hand coded. The run time header contains a default definition (just a return from interrupt, RTI, instruction) which you may change to conform to your interrupt structure; for example, calling a C function.

7.3.4 Data Types

The ADSP-2101 is a 16-bit machine with provisions for certain 32-bit operations. The arithmetic types supported directly are listed below. Note that *fract* is not a standard C type. All other data types are mapped onto these types.

Name	*Description*
int	16-bit twos complement value
long int	32-bit twos complement value
unsigned int	16-bit unsigned value
unsigned long int	32-bit unsigned value
fract	16-bit fractional value (1.15 format)
float	32-bit real

Table 7.4 ADSP-2101 C Compiler Arithmetic Types

The ADSP-2101 fractional type is described in the *ADSP-2101 User's Manual*. Briefly, it is a fixed point number with 15 bits of significance whose range is always between ±1. If you assign a *fract* value to a *float* you

get a true floating-point value in the same range. You cannot convert values between *fract* and *integer* types. All operations that are valid for *float* are valid for *fract*. Whenever *float* and *fract* are mixed in expressions, all *fract* are converted to *float* before evaluation.

The floating-point type, which is not a "native" data type for the processor, consists of a 16-bit mantissa and 16-bit exponent. The mantissa must be a twos complement fractional value. The exponent must be a twos complement integer value. The floating point value is equal to: mantissa x 2^{exp}. All floating point numbers must be fully normalized; there must be no sign-bit extension.

Decimal	*Exponent (hex)*	*Mantissa (hex)*
1.0	0001	4000
3.0	0002	6000
3.25	0002	6800
0.0	0000*	0000
–2.0	0001	8000

* Exponent of zero should be zero; in some instances the Compiler may pass zero *floats* with non-zero exponents.

The next table lists all of the standard C language arithmetic and data types, and shows which ADSP-2101 type is used to represent them.

Name	*Underlying Type*
int	int
long int	long int
short int	int
unsigned int	unsigned int
unsigned long int	unsigned long int
char	int
unsigned char	unsigned int
float	float
double	float
long double	float
fract	fract
unsigned fract	fract
long fract	fract
unsigned long fract	fract
short fract	fract

Table 7.5 C Language Types on ADSP-2100 family

7 C Compiler

7.3.5 Memory Usage

Sixteen-bit values, obviously, require one word of ADSP-2101 storage whether on the stack or not. Thirty-two-bit values require two words and are stored with the LSW in lower memory and the the MSW in higher memory.

For example, in a global scope, the C statement:

```
long l = 52;
```

produces the ADSP-2101 source:

```
.VAR/RAM/DM l_[2];
.GLOBAL l_;
.INIT l_: 52,0;
```

Note the use of the underscore

When a 32-bit quantity is pushed onto the stack, its MSW is pushed first, followed by its LSW. Since the stack grows from high memory, the storage order on the stack is the same as for the global variable.

Floating-point numbers are stored with the mantissa in low memory and the exponent in high memory. When pushed onto the stack, the exponent is pushed first, followed by the mantissa, resulting in the same memory relationship as for a global variable.

7.3.6 Storage Classes & Modifiers

The Compiler supports all standard storage classes, types and modifiers:

Classes	*Types*	*Modifiers*
auto	all, including void	const
extern		volatile
register		*plus these extensions*:
static		pm
typedef		dm
		ram
		rom

Register variables are implemented as specified in the standard, namely as hints about processor register use. Specific register use is not guaranteed. The modifiers *pm*, *dm*, *rom*, and *ram* are provided to identify the storage location as either program or data memory and either ROM or RAM. In practice, only the *pm* and the *rom* modifiers are needed, since variables not

on the stack are located in data memory by default and data memory is RAM by default. For example

```
int pm i;
```

defines an integer, *i*, located in program memory and

```
const int rom pm ii;
```

defines a constant, *ii*, to be located in program memory ROM. Note that the *const* modifier makes it a constant, not the *rom* modifier.

The –gpm switch (if given when the Compiler is invoked) takes precedence over the modifiers above.

7.3.7 Function Calling & Exit

When a function is called, the following sequence of events takes place:

Calling Function's Responsibility:

1. The arguments are pushed onto the stack in reverse order. The last argument is pushed first and the first argument in the list is the last one pushed onto the stack.

2. Call the function.

3. Modify the stack pointer to remove the arguments from the stack.

4. Pick up returned value (if any) from AX0 or AX0,AX1 registers. (see below)

At step 2, above, control is passed to the called function. Here is the sequence of events within the called function:

Called Function's Responsibility:

1. Push old frame pointer onto the stack.

2. Create local variables on the stack for use during function execution

3. Save reserved/system registers and any restricted/data registers that will be used.

4. Execute the function's computation, reading arguments from the stack as needed. Store the result in the appropriate (AX0, AX1) register.

5. Restore all saved registers.

6. Release stack space used by locals during execution and restore previous frame pointer.

7. Return control to calling function.

When a function returns, its values (as distinct from any changes in its calling arguments) should be in ALU registers as shown below.

Type of Value Returned	*Register Used*
All 16-bit values	AX0
32-bit integers	AX0 for LSW AX1 for MSW
floating-point	AX0 for exponent AX1 for mantissa

7.4 ASSEMBLY LANGUAGE INTERFACE SUMMARY

With an understanding of the restrictions discussed in this chapter and an understanding of the ADSP-2101 instruction set, writing assembly language routines that are called from C programs is not difficult.

7.4.1 Checklist of Prerequisites

The material in this chapter details all of things you need to understand to successfully interface an assembly language routine to a C program. Here is a checklist of the things you should know:

Topic	*Where Is It Covered?*
Register Restrictions	7.3.2
Stack Usage	7.3.1, 7.3.2 & 7.3.5
Passing Parameters	7.3.7
Function Entry & Exit	7.3.7
Returning Values	7.3.7

7.4.2 Assembly Language Interface Example

The example shows how a simple function in ADSP-2101 source code is defined to properly interface to the C environment.

```
int i,j,k;
main()
{
   k=add(i,j);
}

add(x,y)
{
#pragma ADSP2100
{  Function add (x,y)          }
{       int x,y;               }
{                              }
{  Returns:   z=x+y;           }

   dm(i4,m7)=ay0;              { save registers }
   dm(i4,m7)=ar;

   i6=1;                       { get first parameter }
   modify(i6,m4);
   ax0=dm(i6,m5);              { m5=1, i6 points to 2nd parameter }

   ay0=dm(i6,m5);              { get second parameter }
   ar=ax0+ay0;                 { perform addition }
   ax0=ar;                     { return 16-bit values in ax0 }

   i6=-1;
   modify(i6, m4);
   ay0=dm(i6,m7);              { restore registers }
   ar=dm(i6,m7);

#pragma ADSP2100
}
```

7 C Compiler

7.5 LANGUAGE EXTENSIONS

In addition to the *pm*, *dm*, *ram*, and *rom* modifiers, ADSP-210X C provides one new keyword: *fastswitch*.

Fastswitch is syntactically identical to the *switch* statement. Semantically, however, *fastswitch* assumes that there is no default case. It is the programmer's responsibility to ensure that all possible cases are explicitly provided for. *Fastswitch* exists because it makes use of the DO UNTIL looping capability of processor, while the normal *switch* statement cannot. In many instances, the use of *fastswitch* will result in significantly better performance.

Note that use of *fastswitch* may be dangerous; if you miss a possible case, your program may become stuck in a loop or create some other error. The tables created for both *switch* and *fastswitch* can be optionally stored in program memory or ROM; see the section on Compiler switches in this chapter.

7.6 PROGRAMMING HINTS

Because of the inherent conflicts between the nature of a high-level language like C and the specialized architecture of processors like the ADSP-2101, a number of hints for programming approaches are given in this section as an aid to using the C language system.

7.6.1 Location Of Variables

There are some restrictions on the location of pointers and the objects pointed to. Also, the way in which you declare variables in your C program has performance implications for the assembly code produced.

Pointers and the objects they point to must be in the same memory space (program or data). If they are not, you must explicitly cast the pointer every time it is used to point to something in the other memory space.

7.6.1.1 Globals in PM vs. Globals in DM

Static/global variables are located at fixed addresses in memory. However, accessing data memory is usually more efficient than accessing program memory because the ADSP-2101 cannot perform an immediate value write to program memory. Consequently, global variables declared with the *pm* modifier incur an overhead compared to variables located in data memory.

The example shows two versions of C source, one with default global placement in data memory and the other with explicit *pm* placement, followed by the assembly code produced by each.

C Source, Global Variable in Data Memory

```
int i;
m( );
{
   i=3;
}
```

ADSP-2101 Source, Global Variable in Data Memory

```
SI=3;
DM(i_)=SI;
```

C Source, Global Variable in Program Memory

```
int pm i;
m( );
{
   i=3;
}
```

ADSP-2101 Source, Global Variable in Program Memory

```
I5 = ^i_;          {point to label/address}
SI=3;              {load immediate to register}
PM(I5, M6)=SI;     {write to pm}
```

Figure 7.3 Global variable location: data memory vs. program memory

7.6.2 Location of Stack

ADSP-2101 processors cannot write an immediate value to program memory. Immediate values must be loaded into a register before they can be written to program memory. Consequently, locating the stack in program memory incurs an overhead penalty of at least one additional instruction cycle for each stack access.

The example shows the assembly source operations required to set up the function call in each memory.

C Source, Function Call

```
foo(1, 2, 3);
```

ADSP-2101 Source, Stack in Data Memory

```
DM(I4,M7)=3;
DM(I4,M7)=2;
DM(I4,M7)=1;
CALL foo;
```

ADSP-2101 Source, Stack in Program Memory

```
AX0=3;
PM(I4,M7)=AX0;
AX0=2;
PM(I4,M7)=AX0;
AX0=1;
PM(I4,M7)=AX0;
CALL foo;
```

Figure 7.4 Stack location: effect of data memory vs. program memory

7.7 ERROR MESSAGES

The Compiler produces error messages of three basic types: preprocessing errors, corrected syntax errors, and user errors. Assembler errors may occur as well if the Assembler is automatically invoked. The Compiler can also produce a message about Compiler errors, although you should never see such an error in practice.

Preprocessor errors have the format:

%PPERROR *pp error number* *line number* *filename*
<—— error message ——>

PP error number is the preprocessor error number. *Line number* is the source code line number where the error occurred. *Filename* is the name of the file being compiled. The next line shows the actual error message. If

any preprocessor errors occur, the Compiler itself is not run. Here is an example of a preprocessor error report:

```
%PPERROR  [1]  line 23   filename.c
         Argument count error
```

The rest of the error messages take the following form:

%CC – *filename,* *line number*: *error type*

line of code where error was detected
----------^-------- *error message*

Filename is the name of the file in which the error occurred. *Line number* specifies the line where the error was detected by the Compiler. *Error type* is one of the following:

Corrected Syntax Error
User Error
Compiler Error

The next line displayed is the actual line of C source code where the error is detected. The last line contains a pointer showing where the error is located in the line of code, and, following that, the specific *error message* itself.

For example, if you fail to type a semicolon at the end of line 7 of your source code, you would see an error such as:

```
%CC - dsp_sys.c, line 8: Corrected Syntax Error

          p = 9;
       ----^---  Inserted ;
```

(The missing semicolon is not actually detected until the beginning of the following line.)

7 C Compiler

If you attempt to use a variable which has not been declared, an error such as the following is reported:

```
%CC - dsp_sys.c, line 14: User Error

              k = 2;
       -------^---  k_ not defined in this scope
```

7.7.1 Corrected Syntax Errors

The Compiler has the capability to detect and correct most syntax errors, allowing the program to compile properly. It does not, however, make the corrections in your C source code file. You should take note of the errors and make the corrections in the source file yourself.

A syntax error is one in which the C source does not conform to ANSI draft standard C. However, the Compiler does not support, look for, or warn about any "old" style C syntax. For example the C statement

```
x =- 8;
```

does not mean

```
x = x - 8;
```

as it did in older versions of the language. It simply means that the variable *x* is assigned the value –8.

Because of the free form nature of the C language, syntax errors may not be detected until after the line on which they occur. For example,

```
1  while (i) {
2        j=i+j;
3        p=i+p*2;
4        foo(2,3,4)
5  }
```

gives rise to the error report:

```
%CC - filename.c, line 5: Corrected Syntax Error

                  }
       ---------^---  Inserted ;
```

In fact, the error is on line four, where the semicolon was left off.

If the syntax error or errors are too extensive to be corrected by the Compiler, a user error is reported and compiling is aborted.

7.7.2 User Errors

User errors flag improper usage of various kinds. Even with proper syntax it is possible to misuse a variable, use an undefined variable or violate the language in some other way. The error messages appearing in user errors are typically self-explanatory.

7.7.3 Compiler Errors

If you see this type of message, the Compiler has detected an internal error. Please make a note of the message displayed and contact Analog Devices Digital Signal Processing Division, Applications Engineering Group. See the copyright page of this manual for information on contacting Analog Devices.

7.7.4 Exit Codes

The Compiler returns an exit code to the operating system when it terminates. This code can be examined to determine whether or not to continue processing, such as when a batch file is used to automatically invoke the Assembler or Linker after compilation.

Three exit codes are defined for the C Compiler:

Exit Code	*Meaning*
0	No errors encountered
1	Errors encountered
4	Corrected syntax errors occurred

PROM Splitter 8

8.1 INTRODUCTION

The ADSP-2101 PROM Splitter extracts the address information and the contents of the ROM portion of the Memory Image file (.EXE) and formats the extracted images for uploading to PROM burners.

The PROM Splitter creates output files for program, data, and boot memory. Three usable files are created for PM to organize the PROMs in word addresses corresponding to three-byte instructions. Two usable files are created for DM to organize any data PROMs in terms of two-byte data words. One usable file is created for BM, which is physically byte-wide (although organized internally in vertical groups of four bytes per word address).

Both program and data memory can also be optionally output as a single stream of bytes for vertical rather than horizontal grouping of words in the PROMs. The PROM Splitter can format the PROM image files in Motorola S Record and Intel Hex Record. For one-byte wide files the Motorola S2 format is supported.

8.2 RUNNING THE PROM SPLITTER

To invoke the PROM Splitter from the host system, the command form is:

SPL21 *imagefile outfile* –format –mem_area

Imagefile is the main file name of the memory image file output of the Linker. The .EXE extension is always appended; therefore, the file must have this extension or the PROM Splitter cannot find it.

Outfile is the main file name of the PROM image files. Different names should be used for the program, data, and boot memory files to avoid accidentally overwriting the previous run.

There are two software switches: –format and –mem_area.

8 PROM Splitter

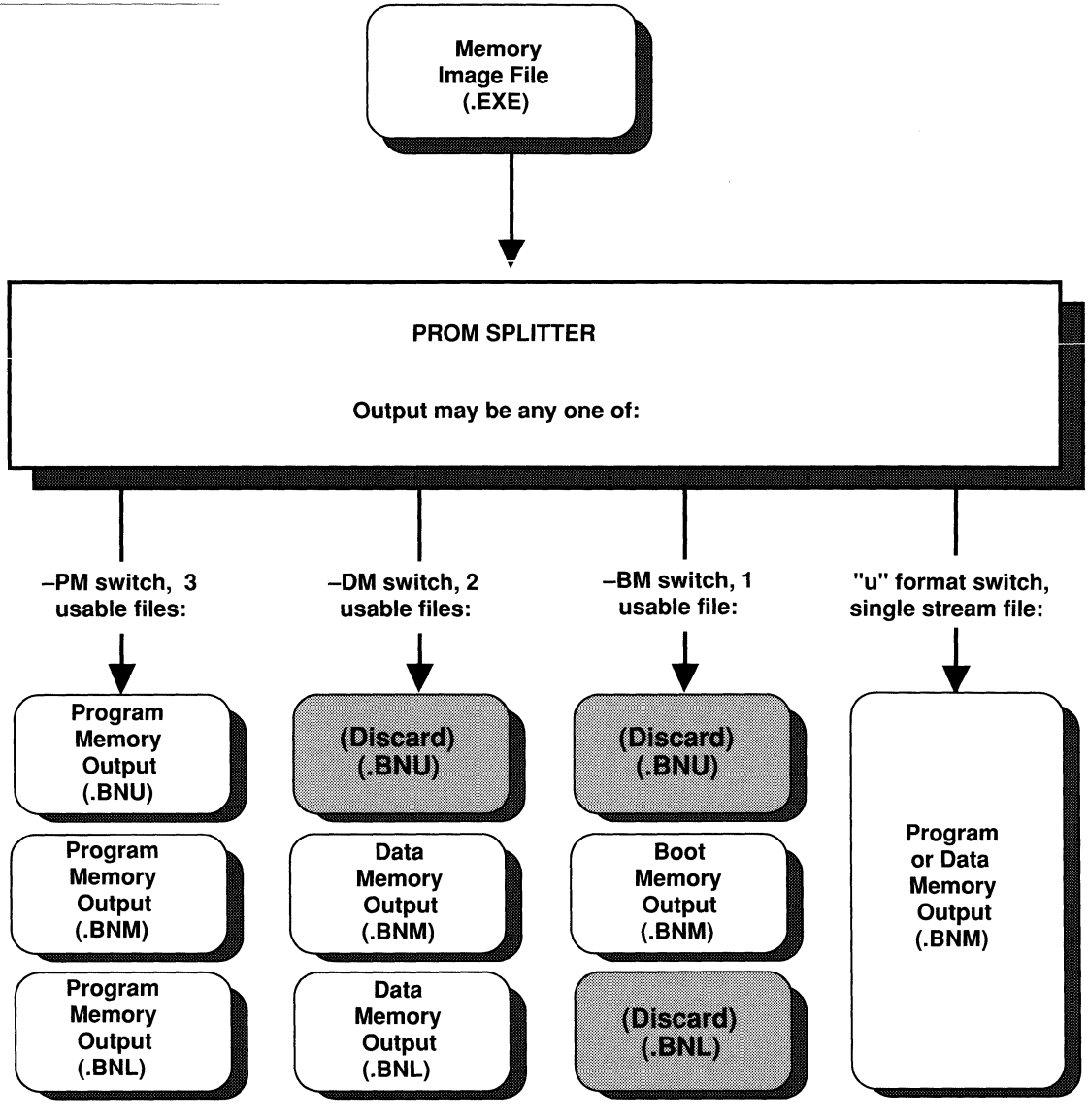


Figure 8.1 PROM Splitter I/O

–Format specifies the outfile PROM image record format and can be –s for Motorola S Record or –i for Intel Hex Record. For program and data memory only, the format may also be –us, –us2 or –ui for a single byte stream output file, as explained below. If no format is specified with this switch, the default is to Intel Hex.

–Mem_area is set to be either –pm for program memory, –dm for data memory, or –bm for boot memory, and specifies which memory space the PROM image files generated are for. To create the PROM image files for each, run the PROM Splitter three times, specifying –pm, –dm, and –bm once each, as in:

SPL21 *fir_sys pmburn* –i –pm

SPL21 *fir_sys dmburn* –i –dm

SPL21 *fir_sys bootburn* –i –bm

Again, remember that different *outfile* names must be specified for the –pm, –dm, and –bm runs. Otherwise the output files from one run may be overwritten by the output of successive runs.

As many ADSP-2101 systems do not use any PROM-based program or data memory, the PROM Splitter is usually run only to create image files for boot memory PROMs. Thus the only invocation command needed in most cases is of the form:

SPL21 *fir_sys bootburn* –bm

(Format defaults to –i, Intel Hex Record).

8.3 PROM SPLITTER OUTPUT

The PROM Splitter generates three PROM image files when the –pm is set. These PROM image files take *outfile* as their root name and use the .BNU extension for the upper byte file, .BNM for the middle byte file, and .BNL for the lower byte file. When the –dm switch is set, the PROM Splitter generates two usable PROM image files. These files take *outfile* as their root name and use the .BNM extension for the upper byte file and .BNL for the lower byte file. The .BNU file is created but should be discarded.

When the –bm switch is set, one usable PROM image file is created which contains all pages of boot memory, and which is named *outfile*.BNM. The

8 PROM Splitter

.BNU and .BNL files are created but can be deleted. Boot memory is always byte-wide and its contents are organized in vertical groups of four bytes (one word). The byte stream output from the PROM Splitter is arranged in such groups, with each group being filled from high-order byte to low-order byte of the word. The high-order byte is located at the lowest byte address of the four-byte group.

The 32-bit words of boot memory consist of a 24-bit instruction padded with an extraneous fourth byte (0xFF); these pad bytes are ignored except for the first one of each page. The pad byte of the first word of each page is the page length for that page of boot memory. This page length is calculated by the PROM Splitter and inserted into the boot memory image file at these locations. For page 0 this value is located at PROM byte address 0x0003.

$$\textit{Page length} = \frac{\text{Number of 24-bit instructions}}{8} - 1$$

(The number of instructions must be rounded up to a multiple of eight.)

For example, a page length of zero indicates eight instruction words, residing in thirty-two sequential bytes. The maximum page length value of 255 indicates 2048 instruction words. (See the *ADSP-2101 User's Manual* for more information on memory interfacing.)

Each boot page must contain a number of words which is a multiple of eight; the PROM Splitter adds extraneous words (0xFFFFFFFF) to the end of the page to assure this. For example, if a page has only 4 instructions, 4 extraneous words are added by the PROM Splitter.

The PROM Splitter can also generate a single image file for the u formats. This is possible only for program and data memory (not boot memory). The byte stream output from the PROM Splitter is arranged with the most significant byte for each location preceding the less significant byte(s), from the lower address to the higher address. The 24-bit words in program memory require a sequence of three bytes (1, 2, 3, 1, 2, 3, etc.), while the 16-bit words of data memory require a sequence of two bytes (1, 2, 1, 2, etc.). Absolute address information is lost in this format.

To create the byte-wide u format image, prefix a "u" to the format specified. The format switch now becomes –us, –us2 or –ui for Motorola S, Motorola S2 and Intel formats, respectively. The PROM image file takes *outfile* as its root name and .BNM as the extension.

Instruction Set Reference 9

9.1 OVERVIEW

This chapter is a complete reference for the instruction set of the ADSP-2101 microcomputer. It is organized by instruction group and, within each group, by individual instruction. The groups and individual instructions are in this order:

ALU
- Add / Add with Carry
- Subtract X-Y / Subtract X-Y with Borrow
- Subtract Y-X / Subtract Y-X with Borrow
- AND, OR, Exclusive OR
- Pass / Clear
- Negate
- NOT
- Absolute Value
- Increment
- Decrement
- Divide

MAC
- Multiply
- Multiply / Accumulate
- Multiply / Subtract
- Clear
- Transfer MR
- Conditional MR Saturation

SHIFTER
- Arithmetic Shift
- Logical Shift
- Normalize
- Derive Exponent
- Block Exponent Adjust
- Arithmetic Shift Immediate
- Logical Shift Immediate

MOVE
- Register Move
- Load Register Immediate
- Data Memory Read (Direct Address)
- Data Memory Read (Indirect Address)
- Program Memory Read (Indirect Address)
- Data Memory Write (Direct Address)
- Data Memory Write (Indirect Address)
- Program Memory Write (Indirect Address)

PROGRAM FLOW
- JUMP
- CALL
- JUMP or CALL on Flag In Pin
- Modify Flag Out Pin
- Return from Subroutine
- Return from Interrupt
- Do Until
- IDLE

MISC
- Stack Control
- Mode Control
- Modify Address Register
- NOP

MULTIFUNCTION
- ALU/MAC/SHIFT operation with Memory Read
- ALU/MAC/SHIFT operation with Data Register Move
- ALU/MAC/SHIFT operation with Memory Write
- Data & Program Memory Read
- ALU/MAC operation w/ Data & Program Memory Read

A page heading identifies the instruction group and individual instruction name.

9 Instruction Set Reference

9.2 CYCLE TIME NOTES

All ADSP-2101 instructions can execute in a single clock period. (The term cycle is used to denote both instruction and clock cycle for the ADSP-2101 for this reason.) Consequently, there is no cycle time information given in the description of any individual instruction. There are, however, some conditions under which an instruction cannot be completed in a single cycle, or an extra cycle is inserted between instructions.

9.2.1 ADSP-2101 Extra Cycle Conditions

There are two conditions that require one or more extra clock cycles to complete an instruction for the ADSP-2101: external memory wait states and multiple off-chip memory accesses.

Memory wait states are programmable, as described in the *ADSP-2101 User's Manual*. From one to seven extra clock cycles may be added to any external data or program memory access.

Because the address and data buses are multiplexed off-chip, the ADSP-2101 can execute one off-chip memory access per instruction with no penalty (other than any wait states inserted, as above). When multiple accesses of off-chip memory are required for one instruction, extra cycles are required. For example, to fetch both an instruction and a data memory value from off-chip requires one extra cycle.

While the two cases described above require extra clock cycles during the execution of an instruction, two other processor operations cause the insertion of extra cycles between instructions. The two operations are the autobuffering feature of serial communications and interrupt handling.

If the autobuffering feature of serial communications is being used to transfer individual data words to or from data memory, then one memory access (requiring 1-8 clock cycles) is "stolen" for each transfer. The timing of this transfer is a function of the rate of serial communication, and is not under direct program control. The stolen memory access occurs only between complete instructions, never between multiple cycles required to complete an instruction. For example, the serial communications controller waits until a data memory access with wait states is complete before "stealing" the cycle(s) it needs.

When an interrupt occurs during the execution of an instruction, the instruction is not completed. A NOP cycle is inserted instead, and the

address of the aborted instruction is stored as the return address for the main program. This NOP, which is followed by a jump to the interrupt-handling routine, is the extra cycle in this case.

9.3 INSTRUCTION SYNTAX NOTATION

The following notation is used in the syntax discussions:

Square Brackets [ ]	Anything within square brackets is an optional part of the instruction statement.
Parallel Lines | |	Lists of parameters enclosed by parallel vertical lines require the choice of one parameter from among the operands listed. If the parallel lines are within brackets, then that operand is optional for that instruction.
CAPITAL LETTERS	denote a literal in the program statement. These are instruction words, register names and operand selections to be coded as shown.
parameters	are shown in small letters and denote an operand in the instruction for which there are numerous choices. For example, the parameter *yop* might have as its choices in the actual instruction: MY0, MY1 or MF. These choices are shown for each instruction.
<exp>	in Shift Immediate instructions, stands for any constant, used as the exponent or shift value.
<data>	stands for any constant or an identifier referenced by the '%' and '^' operators.
<addr>	denotes an immediate value of an address to be coded in the instruction. "Addr" may be either an immediate value or a LABEL. Immediate values may be expressed in binary, octal, hexadecimal or decimal format. If no explicit format designator is used, the default is decimal.

9 Instruction Set Reference

immediate values — may be any constant in any of the formats shown in Appendix E, examples below. Decimal is the default format.

binary	B#10110101010	(prefix B#)
octal	017747	(prefix 0)
hexadecimal	0x7F4B or H#7F4B	(prefix 0x or H#)
decimal	1949	(no prefix)

9.3.1 Punctuation & Multifunction Instructions

All instructions terminate with a semicolon. A comma separates the clauses of a multifunction instruction but does not terminate it. For example, the statements below in Example A create one instruction (defined to execute in one instruction cycle). Example B creates two instructions, requiring two separate instruction cycles.

Example A, One multifunction instruction:

```
AX0 = DM(I0, M0),                    (comma )
AY0 = PM(I4, M4);
```

Example B, Two separate instructions:

```
AX0 = DM(I0, M0);                    (semicolon)
AY0 = PM(I4, M4);
```

9.3.2 Syntax Notation Example

Here is an example of one instruction, the ALU Add/Add with Carry instruction:

[IF cond] | AR / AF | = xop + | yop / C / yop + C | ;

Below this is a list of the permissible *conds*, *xops* and *yops*. The conditional clause is enclosed in square brackets indicating that it is optional.

The destination register for the add operation must be either AR or AF. They are within vertical parallel lines, and one of them must be chosen since there are no brackets (which would signal an optional operand).

Similarly, the *yop* term may consist of a Y operand, the carry bit literal, or the sum of both. One of the three expressions must be used.

9.3.3 Status Notation

The following notation is used in the discussion of the effect each instruction has on the status word(s):

*	An asterisk indicates a bit in the status word that is changed by the execution of this instruction.
–	A dash indicates that this bit is not affected by the instruction.
0 or 1	Indicates that a bit is unconditionally cleared or set by the instruction.

For example, the status word ASTAT is shown below:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	0	–	–

In this example, the MV bit is affected. It may be set or cleared. An explanation of the conditions leading to each outcome always follows this chart.

The AV bit is always cleared in this example.

All other bits are unaffected.

For a complete definition of the status register bits, refer to *ADSP-2101 User's Manual*.

9.3.4 Instruction Word Notation

At the end of each individual instruction definition, the 24-bit format of the assembled opcode is shown. In general, this section uses the same parameter identifiers, such as *yop*, as the instruction syntax itself.

0 and 1	denote specific bits in the instruction word
|	vertical bars are separators between fields or bits, and are added only for clarity.

9 Instruction Set Reference

Here is an example of the instruction word format for the ALU instruction Add / Add with Carry:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					Yop		Xop			0	0	0	0	COND			

This is followed by an explanation of the meaning of each parameter. Z, for example, selects the destination register (AR or AF). AMF denotes the ALU or MAC function. A list of the two different opcodes for Add and Add with Carry follow this section. Finally, the terms *Yop*, *Xop* and COND have the same meaning as in the higher level syntax of the instruction.

Appendix A is a listing of opcodes for each instruction type. This appendix also defines the bit patterns of each parameter field.

Syntax: [IF cond] | AR / AF | = xop | + yop / + C / + yop + C | ;

Permissible xops		*Permissible yops*	*Permissible conds*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: IF EQ AR = AX0 + AY0 + C;

Description: Test the optional condition and, if true, perform the specified addition. If false then perform a no-operation. Omitting the condition performs the addition unconditionally. The addition operation adds the first source operand to the second source operand along with the ALU carry bit, AC, (if designated by the "+C" notation), using binary addition. The result is stored in the destination location. The operands are contained in the data registers specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an arithmetic overflow occurs. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14	13 12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation, in this case:
AMF = 10010 for xop + yop + C operation
AMF = 10011 for xop + yop
Note that xop + C is a special case of xop + yop + C with yop = 0

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

9 ALU
SUBTRACT X-Y / SUBTRACT X-Y with BORROW

Syntax: [IF cond] | AR / AF | = xop | – yop / – yop + C–1 | ;

Permissible xops		*Permissible yops*	*Permissible conds*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: IF GE AR = AX0 – AY0;

Description: Test the optional condition and, if true, then perform the specified subtraction. If the condition is not true then perform a no-operation. Omitting the condition performs the subtraction unconditionally. The subtraction operation subtracts the second source operand from the first source operand, and optionally adds the ALU Carry bit (AC) minus 1 (H#0001), and stores the result in the destination location. The (C-1) quantity effectively implements a borrow capability for multiprecision subtractions. The operands are contained in the data registers specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	-	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an arithmetic overflow occurs. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC operation, Instruction type 9:

23	22	21	20	19	18	17 16 15 14	13 12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 10110 for xop - yop + C - 1 operation.
AMF = 10111 for xop - yop operation.

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

SUBTRACT Y-X / SUBTRACT Y-X with BORROW

Syntax: [IF cond] | AR / AF | = yop − | xop / xop + C − 1 | ;

Permissible xops		*Permissible yops*	*Permissible conds*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: IF GT AR = AY0 − AX0 + C − 1;

Description: Test the optional condition and, if true, then perform the specified subtraction. If the condition is not true then perform a no-operation. Omitting the condition performs the subtraction unconditionally. The subtraction operation subtracts the second source operand from the first source operand, optionally adds the ALU Carry bit (AC) minus 1 (H#0001), and stores the result in the destination location. The (C-1) quantity effectively implements a borrow capability for multiprecision subtractions. The operands are contained in the data registers specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an arithmetic overflow occurs. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14	13 12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 11010 for yop - xop + C - 1 operation.
AMF = 11001 for yop - xop operation.

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

9 ALU AND, OR, XOR

Syntax: [IF cond] | AR / AF | = xop | AND / OR / XOR | yop ;

Permissible xops		*Permissible yops*	*Permissible conds*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: AR = AX0 XOR AY0;

Description: Test the optional condition and if true, then perform the specified bitwise logical operation (logical AND, Inclusive OR, or EXCLUSIVE OR). If the condition is not true then perform a no-operation. Omitting the condition performs the logical operation unconditionally. The operands are contained in the data registers specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Always cleared.
AC Always cleared.

Instruction Format:
Conditional ALU/MAC Operation, Instruction Type 9:

23 22 21 20 19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0 0 1 0 0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 11100 for AND operation.
AMF = 11101 for OR operation.
AMF = 11110 for XOR operation.

Z: Destination register
Yop: Y operand
Xop: X operand
COND: condition

Syntax: [IF cond] | AR / AF | = PASS | xop / yop | ;

Permissible xops		*Permissible yops*	*Permissible conds*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1	0, 1	GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: IF GE AR = PASS AY0;

Description: Test the optional condition and if true, pass the source operand unmodified through the ALU block and store in the destination location. If the condition is not true perform a no-operation. Omitting the condition performs the PASS unconditionally. The source operand is contained in the data registers specified in the instruction.

The PASS instruction performs the transfer to the AR register and affects the status flag; this instruction is different from a register move operation which does not affect any status flags. PASS 0 is the best method of clearing AR; it can also be done in a multifunction instruction in conjunction with memory reads and writes. The 1 argument is H#0001.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Always cleared.
AC Always cleared.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 10000 for PASS yop operation.
AMF = 10011 for PASS xop operation.
Note that PASS xop is a special case of xop + yop, with yop specified as 0.
Z: Destination register Yop: Y operand
Xop: X operand COND: condition

9 ALU NEGATE

Syntax: [IF cond] | AR / AF | = – | xop / yop | ;

Permissible xops		*Permissible yops*	*Permissible conds*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: IF LT AR = – AY0;

Description: Test the optional condition and if true, then NEGATE the source operand and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the NEGATE operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if operand = H#8000. Cleared otherwise.
AC Always cleared.

Instruction Format:
Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 10101 for - yop operation.
AMF = 11001 for - xop operation
Note that –xop is a special case of yop –xop, with yop specified to be 0.

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

Syntax: [IF cond] | AR / AF | = NOT | xop / yop | ;

Permissible xops		*Permissible yops*	*Permissible conds*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1	0	GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: IF NE AF = NOT AX0;

Description: Test the optional condition and if true, then perform the logical complement (ones complement) of the source operand and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the complement operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Always cleared.
AC Always cleared.

Instruction Format:
Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 10100 for NOT yop operation.
AMF = 11011 for NOT xop operation.

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

9 ALU ABSOLUTE VALUE

Syntax: [IF cond] | AR / AF | = ABS xop ;

Permissible xops		*Permissible conds*		
AX0	MR2	EQ	LE	AC
AX1	MR1	NE	NEG	NOT AC
AR	MR0	GT	POS	MV
	SR1	GE	AV	NOT MV
	SR0	LT	NOT AV	NOT CE

Example: IF NEG AF = ABS AX0 ;

Description: Test the optional condition and, if true, then take the absolute value of the source operand and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the absolute value operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0	
		SS	MV	AQ	AS	AC	AV	AN	AZ
		-	-	-	*	0	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if *xop* is H#8000. Cleared otherwise.
AV Set if *xop* is H#8000. Cleared otherwise.
AC Always cleared.
AS Set if the source operand is negative. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	0 0	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 11111 for ABS xop operation.

Z: Destination register
Xop: X operand
COND: condition

Syntax: [IF cond] | AR / AF | = yop + 1 ;

Permissible yops	*Permissible conds*		
AY0	EQ	LE	AC
AY1	NE	NEG	NOT AC
AF	GT	POS	MV
	GE	AV	NOT MV
	LT	NOT AV	NOT CE

Example: IF GT AF = AF + 1;

Description: Test the optional condition and if true, then increment the source operand by H#0001 and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the increment operation unconditionally. The source operand is contained in the data register specified in the instruction. This operation enables setting AR or AF to + 1 by selecting yop = 0.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an overflow is generated. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 10001 for yop + 1 operation.
Note that the xop field is ignored for the increment operation.

Z: Destination register
Yop: Y operand
Xop: X operand
COND: condition

9 ALU DECREMENT

Syntax: [IF cond] | AR / AF | = yop – 1 ;

Permissible yops	*Permissible conds*		
AY0	EQ	LE	AC
AY1	NE	NEG	NOT AC
AF	GT	POS	MV
	GE	AV	NOT MV
	LT	NOT AV	NOT CE

Example: IF EQ AR = AY1 – 1 ;

Description: Test the optional condition and if true, then decrement the source operand by H#0001 and store in the destination location. If the condition is not true then perform a no-operation.Omitting the condition performs the decrement operation unconditionally. The source operand is contained in the data register specified in the instruction. This operation enables setting AR or AF to –1 by selecting yop = 0.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an overflow is generated. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:
Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 11000 for yop – 1 operation.
Note that the xop field is ignored for the decrement operation.

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

Syntax:

```
DIVS yop , xop ;
DIVQ xop ;
```

Permissible xops		*Permissible yops*
AX0	MR2	AY1
AX1	MR1	AF
AR	MR0	
	SR1	
	SR0	

Description: These instructions implement *yop* / *xop*. There are two divide primitives, DIVS and DIVQ. A single precision divide, with a 32-bit numerator and a 16-bit denominator, yielding a 16-bit quotient, executes in 16 cycles. Higher precision divides are also possible.

The division can be either signed or unsigned, but both the numerator and denominator must be the same; both signed or unsigned. The programmer sets up the divide by sorting the upper half of the numerator in any permissible *yop* (AY1 or AF), the lower half of the numerator in AY0, and the denominator in any permissible *xop*. The divide operation is then executed with the divide primitives, DIVS and DIVQ. Repeated execution of DIVQ implements a non-restoring conditional add-subtract division algorithm. At the conclusion of the divide operation the quotient will be in AY0.

To implement a signed divide, first execute the DIVS instruction once, which computes the sign of the quotient. Then execute the DIVQ instruction for as many times as there are bits remaining in the quotient (e.g., for a signed, single-precision divide, execute DIVS once and DIVQ 15 times).

To implement an unsigned divide, first place the upper half of the numerator in AF, then initialize the AQ bit to zero by manually clearing it in the Arithmetic Status Register, ASTAT. This indicates that the sign of the quotient is positive. Then execute the DIVQ instruction for as many times as there are bits in the quotient (e.g., for an unsigned single-precision divide, execute DIVQ 16 times).

The quotient bit generated on each execution of DIVS and DIVQ is the AQ bit which is written to the ASTAT register at the end of each cycle. The final remainder produced by this algorithm (and left over in the AF register) is not valid and must be corrected if it is needed. For more information, consult Appendix B, "Division Exceptions," in your *User's Manual*.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	*	–	*	*	*	*

AQ — Loaded with the bit value equal to the AQ bit computed on each cycle from execution of the DIVS or DIVQ instruction.

AC
AV
AN
AZ — These bits may change during execution of DIVS or DIVQ instruction; however, the bit values are meaningless and should be ignored.

Instruction Format:

DIVQ, Instruction Type 23:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	0	0	0	1	0	Xop			0	0	0	0	0	0	0	0

DIVS, Instruction Type 24:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	0	Yop		Xop			0	0	0	0	0	0	0	0

Xop: X operand Yop: Y operand

Syntax: [IF cond] | MR / MF | = xop * yop | (SS) / (SU) / (US) / (UU) / (RND) | ;

Permissible xops		*Permissible yops*	*Permissible conds*		
MX0	AR	MY0	EQ	LE	AC
MX1	SR1	MY1	NE	NEG	NOT AC
MR2	SR0	MF	GT	POS	MV
MR1			GE	AV	NOT MV
MR0			LT	NOT AV	NOT CE

Example: IF EQ MR = MX0 * MF (UU);

Description: Test the optional condition and, if true, then multiply the two source operands and store in the destination location. If the condition is not true perform a no-operation. Omitting the condition performs the multiplication unconditionally. The operands are contained in the data registers specified in the instruction. When MF is the destination operand, only bits 31-16 of the product are stored in MF.

The data format selection field following the two operands specifies whether each respective operand is in Signed (S) or Unsigned (U) format. The *xop* is specified first and *yop* is second. There is no default; one of the data formats must be specified.

If RND (Round) is specified, the MAC multiplies the two source operands, rounds the result to the most significant 24 bits (or rounds bits 31-16 to 16 bits if there is no overflow from the multiply), and stores the result in the destination location. The two multiplication operands *xop* and *yop* are considered to be in twos complement format. All rounding is unbiased. For a discussion of unbiased rounding, see "Rounding Mode" in the "Multiplier/Accumulator" section of the *ADSP-2101 User's Manual*.

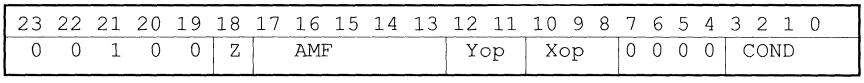
Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:



23	22	21	20	19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*	*Y-Operand*
0 0 1 0 0	xop * yop	(SS)	Signed	Signed
0 0 1 0 1	xop * yop	(SU)	Signed	Unsigned
0 0 1 1 0	xop * yop	(US)	Unsigned	Signed
0 0 1 1 1	xop * yop	(UU)	Unsigned	Unsigned
0 0 0 0 1	xop * yop	(RND)	Signed	Signed

Z:	Destination register	Yop:	Y operand
Xop:	X operand	COND:	condition

Syntax: [IF cond] | MR / MF | = MR + xop * yop | (SS) / (SU) / (US) / (UU) / (RND) | ;

Permissible xops		*Permissible yops*	*Permissible conds*		
MX0	AR	MY0	EQ	LE	AC
MX1	SR1	MY1	NE	NEG	NOT AC
MR2	SR0	MF	GT	POS	MV
MR1			GE	AV	NOT MV
MR0			LT	NOT AV	NOT CE

Example: IF GE MR = MR + MX0 * MY1 (SS) ;

Description: Test the optional condition and, if true, then multiply the two source operands, add the product to the present contents of the MR register, and store the result in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the multiply / accumulate unconditionally. The operands are contained in the data registers specified in the instruction. When MF is the destination operand, only bits 31-16 of the 40-bit result are stored in MF.

The data format selection field to the right of the two operands specifies whether each respective operand is in signed (S) or unsigned (U) format. The X operand is specified first and Y operand is second. There is no default. A data format must be specified.

If RND (Round) is specified, the MAC multiplies the two source operands, adds the product to the current contents of the MR register, rounds the result to the most significant 24 bits (or rounds bits 31-16 to the nearest 16 bits if there is no overflow from the multiply/accumulate), and stores the result in the destination location. The two multiplication operands *xop* and *yop* are considered to be in signed twos complement format. All rounding is unbiased. For a discussion of unbiased rounding, see "Rounding Mode" in the "Multiplier/Accumulator" section of the *ADSP-2101 User's Manual*.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*	*Y-Operand*
0 1 0 0 0	MR+xop * yop	(SS)	Signed	Signed
0 1 0 0 1	MR+xop * yop	(SU)	Signed	Unsigned
0 1 0 1 0	MR+xop * yop	(US)	Unsigned	Signed
0 1 0 1 1	MR+xop * yop	(UU)	Unsigned	Unsigned
0 0 0 1 0	MR+xop * yop	(RND)	Signed	Signed

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

Syntax: [IF cond] | MR / MF | = MR – xop * yop | (SS) / (SU) / (US) / (UU) / (RND) | ;

Permissible xops		*Permissible yops*	*Permissible conds*		
MX0	AR	MY0	EQ	LE	AC
MX1	SR1	MY1	NE	NEG	NOT AC
MR2	SR0	MF	GT	POS	MV
MR1			GE	AV	NOT MV
MR0			LT	NOT AV	NOT CE

Example: IF LT MR = MR – MX1 * MY0 (SU) ;

Description: Test the optional condition and, if true, then multiply the two source operands, subtract the product from the present contents of the MR register, and store the result in the destination location. If the condition is not true perform a no-operation. Omitting the condition performs the multiply/subtract unconditionally. The operands are contained in the data registers specified in the instruction. When MF is the destination operand, only bits 16-31 of the 40-bit result are stored in MF.

The data format selection field to the right of the two operands specifies whether each respective operand is in signed (S) or unsigned (U) format. The X operand is specified first and Y operand is second. There is no default; a data format must be specified.

If RND (Round) is specified, the MAC multiplies the two source operands, subtracts the product from the current contents of the MR register, rounds the result to the most significant 24 bits (or rounds bits 31-16 to 16 bits if there is no overflow from the multiply/accumulate), and stores the result in the destination location. The two multiplication operands *xop* and *yop* are considered to be in signed twos complement format. All rounding is unbiased. For a discussion of unbiased rounding, see "Rounding Mode" in the "Multiplier/Accumulator" section of the *ADSP-2101 User's Manual*.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of the upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*	*Y-Operand*
0 1 1 0 0	MR–xop * yop	(SS)	Signed	Signed
0 1 1 0 1	MR–xop * yop	(SU)	Signed	Unsigned
0 1 1 1 0	MR–xop * yop	(US)	Unsigned	Signed
0 1 1 1 1	MR–xop * yop	(UU)	Unsigned	Unsigned
0 0 0 1 1	MR–xop * yop	(RND)	Signed	Signed

Z:	Destination register	Yop:	Y operand
Xop:	X operand	COND:	condition

Syntax: [IF cond] | MR | = 0 ;
| MF |

Permissible conds

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF GT MR = 0;

Description: Test the optional condition and, if true, then set the specified register to zero. If the condition is not true perform a no-operation. Omitting the condition performs the clear unconditionally. The entire 40-bit MR or 16-bit MF register is cleared to zero.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	0	–	–	–	–	–	–

MV Always cleared.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					1	1	0	0	0	0	0	0	0	COND			

AMF: Specifies the ALU or MAC Operation. In this case, AMF = 00100 for clear operation.

Z: Destination register COND: condition

9 MAC TRANSFER MR

Syntax: [IF cond] | MR / MF | = MR [(RND)] ;

Permissible conds

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF EQ MF = MR (RND);

Description: Test the optional condition and, if true, then perform the MR transfer according to the description below. If the condition is not true then perform a no-operation. Omitting the condition performs the transfer unconditionally.

This transfer operation actually performs a multiply/accumulate, specifying *yop = 0* as a multiplicand and adding the zero product to the contents of MR. The MR register may be optionally rounded at the boundary between bits 15 and 16 of the result by specifying the RND option. If MF is specified as the destination, bits 31-16 of the result are stored in MF. If MR is the destination, the entire 40-bit result is stored in MR.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14 13	12 11	10 9 8 7 6 5	4 3 2 1 0
0	0	1	0	0	Z	AMF	1 1	Xop	0 0 0 0	COND

AMF: Specifies the ALU or MAC Operation. In this case,
AMF = 01000 designates MR + xop*yop (SS). The *yop* is selected to be zero; *xop* is ignored.

Z: Destination register
Xop: X operand
COND: condition

Syntax: IF MV SAT MR ;

Description: Test the MV (MAC Overflow) bit in the Arithmetic Status Register (ASTAT), and if set, then saturate the lower-order 32 bits of the 40-bit MR register; if the MV is not set then perform a no-operation.

Saturation of MR is executed with this instruction for one cycle only; MAC saturation is not a continuous mode that is enabled or disabled. The saturation instruction is intended to be used at the completion of a series of multiply/accumulate operations so that temporary overflows do not cause the accumulator to saturate.

The saturation result depends on the state of MV and on the sign of MR (the MSB of MR2). The possible results after execution of the saturation instruction are shown in the table below.

MV	*MSB of MR2*	*MR contents after saturation*
0	0	No change
0	1	No change
1	0	00000000 0111111111111111 1111111111111111
1	1	11111111 1000000000000000 0000000000000000

Status Generated: No status bits affected.

Instruction Format:

Saturate MR operation, Instruction Type 25:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

9 SHIFTER ARITHMETIC SHIFT

Syntax: [IF cond] SR = [SR OR] ASHIFT xop | (HI) (LO) | ;

Permissible xops		*Permissible conds*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF LT SR = SR OR ASHIFT SI (LO);

Description: Test the optional condition and, if true, then perform the designated arithmetic shift. If the condition is not true then perform a no-operation. Omitting the condition performs the shift unconditionally. The operation arithmetically shifts the bits of the operand by the amount and direction specified in the Shift Code from the SE register. Positive Shift Codes cause a left shift (upshift) and negative Codes cause a right shift (downshift).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option.

For ASHIFT with a positive Shift Code (i.e. positive value in SE), the operand is shifted left; with a negative Shift Code (i.e. negative value in SE), the operand is shifted right. The number of positions shifted is the count in the Shift Code. The 32-bit output field is sign-extended to the left (the MSB of the input is replicated to the left), and the output is zero-filled from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped. Bits shifted out of the low order bit in the destination field (SR_0) are dropped.

To shift a double precision number, the same Shift Code is used for both halves of the number. On the first cycle, the upper half of the number is shifted using the HI and PASS options; then on the second cycle, the lower half of the number is shifted using the LO and OR options.

Status Generated: None affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

SF	*Shifter Function*
0 1 0 0	ASHIFT (HI, PASS)
0 1 0 1	ASHIFT (HI, OR)
0 1 1 0	ASHIFT (LO, PASS)
0 1 1 1	ASHIFT (LO, OR)

Xop: shifter operand COND: condition

9 SHIFTER LOGICAL SHIFT

Syntax: [IF cond] SR = [SR OR] LSHIFT xop | (HI) / (LO) | ;

Permissible xops		*Permissible conds*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF GE SR = LSHIFT SI (HI) ;

Description: Test the optional condition and, if true, then perform the designated logical shift. If the condition is not true then perform a no-operation. Omitting the condition performs the shift unconditionally. The operation logically shifts the bits of the operand by the amount and direction specified in the Shift Code from the SE register. Positive Shift Codes cause a left shift (upshift) and negative Codes cause a right shift (downshift).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option.

For LSHIFT with a positive Shift Code, the operand is shifted left; the numbers of positions shifted is the count in the Shift Code. The 32-bit output field is zero-filled from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR31) are dropped.

For LSHIFT with a negative Shift Code, the operand is shifted right; the number of positions shifted is the count in the Shift Code. The 32-bit output field is zero-filled from the left. Bits shifted out of the low order bit in the destination field (SR0) are dropped.

To shift a double precision number, the same Shift Code is used for both halves of the number. On the first cycle, the upper half of the number is shifted using the HI and PASS options; then on the second cycle, the lower half of the number is shifted using the LO and OR options.

Status Generated: None affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

SF	*Shifter Function*
0 0 0 0	LSHIFT (HI, PASS)
0 0 0 1	LSHIFT (HI, OR)
0 0 1 0	LSHIFT (LO, PASS)
0 0 1 1	LSHIFT (LO, OR)

Xop: shifter operand COND: condition

9 SHIFTER NORMALIZE

Syntax: [IF cond] SR = [SR OR] NORM xop | (HI) / (LO) | ;

Permissible xops		*Permissible conds*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: SR = NORM SI (HI) ;

Description: Test the optional condition and, if true, then perform the designated normalization. If the condition is not true then perform a no-operation. Omitting the condition performs the normalize unconditionally. The operation arithmetically shifts the input operand to eliminate all but one of the sign bits. The amount of the shift comes from the SE register. The SE register may be loaded with the proper Shift Code to eliminate the redundant sign bits by using the Derive Exponent instruction; the Shift Code loaded will be the negative of the quantity: (the number of sign bits minus one).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option. When the LO reference is selected, the 32-bit output field is zero-filled to the left. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped.

The 32-bit output field is zero-filled from the right. If the exponent of an overflowed ALU result was derived with the HIX modifier, the 32-bit output field is filled from left with the ALU Carry (AC) bit in the Arithmetic Status Register (ASTAT) during a NORM(HI) operation. In this case (SE = 1 from the exponent detection on the overflowed ALU value) a downshift occurs.

To normalize a double precision number, the same Shift Code is used for both halves of the number. On the first cycle, the upper half of the number is shifted using the HI and PASS options; then on the second cycle, the lower half of the number is shifted using the LO and OR options.

Status Generated: None affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

SF	*Shifter Function*
1 0 0 0	NORM (HI, PASS)
1 0 0 1	NORM (HI, OR)
1 0 1 0	NORM (LO, PASS)
1 0 1 1	NORM (LO, OR)

Xop: shifter operand COND: condition

9 SHIFTER DERIVE EXPONENT

Syntax: [IF cond] SE = EXP xop | (HI) / (LO) / (HIX) | ;

Permissible xops		*Permissible conds*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF GT SE = EXP MR1 (HI) ;

Description: Test the optional condition and, if true, perform the designated exponent operation. If the condition is not true then perform a no-operation. Omitting the condition performs the exponent operation unconditionally.

The EXP operation derives the effective exponent of the input operand to prepare for the normalization operation (NORM). EXP supplies the source operand to the exponent detector, which generates a Shift Code from the number of leading sign bits in the input operand. The Shift Code, stored in SE at the completion of the EXP operation, is the effective exponent of the input value. The Shift Code depends on which exponent detector mode is used (HI, HIX, LO).

In the HI mode, the input is interpreted as a single precision signed number, or as the upper half of a double precision signed number. The exponent detector counts the number of leading sign bits in the source operand and stores the resulting Shift Code in SE. The Shift Code will equal the negative of the number of redundant sign bits in the input.

In the HIX mode, the input is interpreted as the result of an add or subtract which may have overflowed. HIX is intended to handle shifting and normalization of results from ALU operations. The HIX mode examines the ALU Overflow bit (AV) in the Arithmetic Status Register: if AV is set, then the effective exponent of the input is +1 (indicating that an ALU overflow occurred before the EXP operation), and +1 is stored in SE. If AV is not set, then HIX performs exactly the same operations as the HI mode.

In the LO mode, the input is interpreted as the lower half of a double precision number. In performing the EXP operation on a double precision number, the higher half of the number must first be processed with EXP in the HI or HIX mode, and then the lower half can be processed with EXP in the LO mode. If the upper half contained a non-sign bit, then the correct Shift Code was generated in the HI or HIX operation and that is the code that is stored in SE. If, however, the upper half was all sign bits, then EXP in the LO mode totals the number of leading sign bits in the double precision word and stores the resulting Shift Code in SE.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	–	–	–	–	–	–	–

SS — Set by the MSB of the input for an EXP operation in the HI or HIX mode with AV = 0. Set by the MSB inverted in the HIX mode with AV = 1. Not affected by operations in the LO mode.

Instruction Format:

Conditional Shift Operation, Instruction Type 16:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

SF	*Shifter Function*
1100	EXP (HI)
1101	EXP (HIX)
1110	EXP (LO)

Xop: shifter operand COND: condition

9 SHIFTER BLOCK EXPONENT ADJUST

Syntax: [IF cond] SB = EXPADJ xop ;

Permissible xops		*Permissible conds*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF GT SB = EXPADJ SI ;

Description: Test the optional condition and, if true, perform the designated exponent operation. If the condition is not true then perform a no-operation. Omitting the condition performs the exponent operation unconditionally. The Block Exponent Adjust operation, when performed on a series of numbers, derives the effective exponent of the number largest in magnitude. This exponent can then be associated with all of the numbers in a block floating point representation.

The Block Exponent Adjust circuitry applies the input operand to the exponent detector to derive its effective exponent. The input must be a signed twos complement number. The exponent detector operates in HI mode (see the EXP instruction, above).

At the start of a block, the SB register should be initialized to –16 to set SB to its minimum value. On each execution of the EXPADJ instruction, the effective exponent of each operand is compared to the current contents of the SB register. If the new exponent is greater than the current SB value, it is written to the SB register, updating it. Therefore, at the end of the block, the SB register will contain the largest exponent found. EXPADJ is only an inspection operation; no actual shifting takes place since the true exponent is not known until all the numbers in the block have been checked. However, the numbers can be shifted at a later time after the true exponent has been derived.

Extended (overflowed) numbers and the lower halves of double precision numbers can not be processed with the Block Exponent Adjust instruction.

Status Generated: Not affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

SF = 1111.

Xop: shifter operand COND: condition

9 SHIFTER
ARITHMETIC SHIFT IMMEDIATE

Syntax: SR = [SR OR] ASHIFT xop BY <exp> | (HI) / (LO) | ;

Permissible xops		*<exp>*
SI	MR0	Any constant
SR1	MR1	
SR0	MR2	
AR		

Example: SR = SR OR ASHIFT SR0 BY 3 (LO);

Description: Arithmetically shift the bits of the operand by the amount and direction specified by the constant in the exponent field. Positive Shift Codes cause a left shift (upshift) and negative Codes cause a right shift (downshift).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option.

For ASHIFT with a positive Shift Code (i.e. positive value in SE), the operand is shifted left; with a negative Shift Code (i.e. negative value in SE), the operand is shifted right. The 32-bit output field is sign-extended to the left (the MSB of the input is replicated to the left), and the output is zero-filled from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_1) are dropped. Bits shifted out of the low order bit in the destination field (SR_0) are dropped.

To shift a double precision number, the same Shift Code is used for both parts of the number. On the first cycle, the upper half is shifted using the HI and PASS options. Next the lower half is shifted using the LO and OR options.

Status Generated: None affected.

Instruction Format:
Shift Immediate Operation, Instruction Type 15:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	SF				Xop			<data>							

SF	*Shifter Function*
0 1 0 0	ASHIFT (HI, PASS)
0 1 0 1	ASHIFT (HI, OR)
0 1 1 0	ASHIFT (LO, PASS)
0 1 1 1	ASHIFT (LO, OR)

Xop: Shifter Operand <data>: 8-bit signed shift value

9 SHIFTER LOGICAL SHIFT IMMEDIATE

Syntax: SR = [SR OR] LSHIFT xop BY <exp> | (HI) / (LO) | ;

Permissible xops		*<exp>*
SI	MR0	Any constant
SR1	MR1	
SR0	MR2	
AR		

Exampie: SR = LSHIFT SR1 BY – 6 (HI) ;

Description: Logically shifts the bits of the operand by the amount and direction specified by the constant in the exponent field. Positive Shift Codes cause a left shift (upshift) and negative Codes cause a right shift (downshift).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option.

For LSHIFT with a positive value, the operand is shifted left; the numbers of positions shifted is the count in the shift value. The 32-bit output field is zero-filled to the left and from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_{n}) are dropped.

For LSHIFT with a negative value, the operand is shifted right; the number of positions shifted is the count in the shift value. The 32-bit output field is zero-filled from the left and to the right. Bits shifted out of the low order bit are dropped.

Status Generated: None affected.

Instruction Format:
Shift Immediate Operation, Instruction Type 15:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	SF				Xop			<data>							

SF	*Shifter Function*
0 0 0 0	LSHIFT (HI, PASS)
0 0 0 1	LSHIFT (HI, OR)
0 0 1 0	LSHIFT (LO, PASS)
0 0 1 1	LSHIFT (LO, OR)

Xop: Shifter Operand <data>: 8-bit signed shift value

Syntax: reg = reg ;

Permissible registers

AX0	MX0	SI	SB	CNTR
AX1	MX1	SE	PX	OWRCNTR (*write only*)
AY0	MY0	SR1	ASTAT	RX0
AY1	MY1	SR0	MSTAT	RX1
AR	MR2	I0-I7	SSTAT (*read only*)	TX0
	MR1	M0-M7	IMASK	TX1
	MR0	L0-L7	ICNTL	IFC (*write only*)

Example: I7 = AR;

Description: Move the contents of the source to the destination location. The contents of the source are always right-justified in the destination location after the move.

When transferring a smaller register to a larger register (e.g., an 8-bit register to a 16-bit register), the value stored in the destination is either sign-extended to the left if the source is a signed value, or zero-filled to the left if the source is an unsigned value. The unsigned registers which (when used as the source) cause the value stored in the destination to be zero-filled to the left are: I0 through I7, L0 through L7, CNTR, PX, ASTAT, MSTAT, SSTAT, IMASK, and ICNTL. All other registers cause sign-extension to the left.

When transferring a larger register to a smaller register (e.g., a 16-bit register to a 14-bit register), the value stored in the destination is right-justified (bit 0 maps to bit 0) and the higher-order bits are dropped.

Note that whenever MR1 is loaded with data, it is sign-extended into MR2.

Status Generated: None affected.

Instruction Format:
Internal Data Move, Instruction Type 17:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	0	0	DST RGP		SRC RGP		DEST REG				SOURCE REG			

SRC RGP (Source Register Group) and SOURCE REG (Source Register) select the source register according to the Register Selection Table (see Appendix A).

DST RGP (Destination Register Group) and DEST REG (Destination Register) select the destination register according to the Register Selection Table (see Appendix A).

Syntax: reg = <data> ;

data: <constant>
'%' <identifier>
'^' <identifier>

Permissible registers
dregs *(16-bit data load)* regs *(maximum 14-bit data load)*

AX0	MX0	SI	SB	CNTR
AX1	MX1	SE	PX	OWRCNTR (*write only*)
AY0	MY0	SR1	ASTAT	RX0
AY1	MY1	SR0	MSTAT	RX1
AR	MR2		IMASK	TX0
	MR1		ICNTL	TX1
	MR0		I0-I7	IFC (*write only*)
	M0-M7			
	L0-L7			

Example:
```
I0 = ^data_buffer;
L0=%data_buffer;
```

Description: Move the data value specified to the destination location. The data may be a constant, or any identifier referenced with the "length of" (%) or "pointer to" (^) operators. The data value is contained in the instruction word, with 16 bits for data register loads and up to 14 bits for other register loads. The value is always right-justified in the destination location after the load (bit 0 maps to bit 0). When a value of length less than the length of the destination is moved, it is sign-extended to the left to fill the destination width.

Note that whenever MR1 is loaded with data, it is sign-extended into MR2.

The RX and TX registers may be loaded with a maximum of 14 bits of data, although the registers themselves are 16 bits wide.

Status Generated: None affected.

Instruction Format :
Load Data Register Immediate, Instruction Type 6:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	DATA																DREG			

DATA contains the immediate value to be loaded into the Data Register destination location. The data is right-justified in the field, so the value loaded into an N-bit destination register is contained in the lower-order N bits of the DATA field.

DREG selects the destination Data Register for the immediate data value. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

Load Non-Data Register Immediate Instruction Type 7:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	RGP				DATA												REG			

DATA contains the immediate value to be loaded into the Non-Data Register destination location. The data is right-justified in the field, so the value loaded into an N-bit destination register is contained in the lower-order N bits of the DATA field.

RGP (Register Group) and REG (Register) select the destination register according to the Register Selection Table (see Appendix A).

DATA MEMORY READ (Direct Address)

Syntax: reg = DM (<addr>) ;

Permissible registers

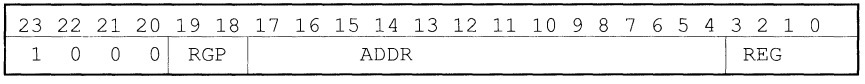
AX0	MX0	SI	SB	CNTR
AX1	MX1	SE	PX	OWRCNTR (*write only*)
AY0	MY0	SR1	ASTAT	RX0
AY1	MY1	SR0	MSTAT	RX1
AR	MR2	I0-I7		TX0
	MR1	M0-M7	IMASK	TX1
	MR0	L0-L7	ICNTL	IFC (*write only*)

Example: SI = DM(*ad_port0*);

Description: The Read instruction moves the contents of the data memory location to the destination register. The addressing mode is direct addressing (designated by an immediate address value or by a label). The data memory address is stored directly in the instruction word as a full 14-bit field. The contents of the source are always right-justified in the destination register after the read (bit 0 maps to bit 0).

Status Generated: None affected.

Instruction Format:
Data Memory Read (Direct Address), Instruction Type 3:



ADDR contains the direct address to the source location in Data Memory.

RGP (Register Group) and REG (Register) select the destination register according to the Register Selection Table (see Appendix A).

9 MOVE
DATA MEMORY READ (Indirect Address)

Syntax: dreg = DM (I0 | I1 | I2 | I3 , M0 | M1 | M2 | M3) ;
I4 | I5 | I6 | I7 , M4 | M5 | M6 | M7

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR1
AY1	MY1	SR0
AR	MR2	
	MR1	
	MR0	

Example: AY0 = DM (I3, M1);

Description: The Data Memory Read Indirect instruction moves the contents of the data memory location to the destination register. The addressing mode is register indirect with post-modify. The contents of the source are always right-justified in the destination register after the read (bit 0 maps to bit 0).

Status Generated: None affected.

Instruction Format:
ALU / MAC Operation with Data Memory Read, Instruction Type 4:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	G	0	0	AMF					0	0	0	0	0	DREG				I		M	

AMF specifies the ALU or MAC operation to be performed in parallel with the Data Memory Read. In this case, AMF = 00000, indicating a No-Op for the ALU / MAC function.

DREG selects the destination Data Register . One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

G specifies which Data Address Generator the I and M registers are selected from. These registers must be from the same DAG as separated by the gray bar above. I specifies the indirect address pointer (I register). M specifies the modify register (M register).

Syntax: dreg = PM (I4 / I5 / I6 / I7 , M4 / M5 / M6 / M7) ;

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR1
AY1	MY1	SR0
AR	MR2	
	MR1	
	MR0	

Example: MX1 = PM (I6, M5);

Description: The Program Memory Read Indirect instruction moves the contents of the program memory location to the destination register. The addressing mode is register indirect with post-modify. The 16 most significant bits of the Program Memory Data bus (PMD23-8) are loaded into the destination register, with bit PMD8 lining up with bit 0 of the destination register (right-justification). If the destination register is less than 16 bits wide, the most significant bits are dropped. Bits PMD7-0 are always loaded into the PX register. You may ignore these bits or read them out on a subsequent cycle.

Status Generated: None affected

Instruction Format:
ALU / MAC Operation with Program Memory Read, Instruction Type 5:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	AMF					0	0	0	0	0	DREG				I		M	

AMF specifies the ALU or MAC operation to be performed in parallel with the Data Memory Read. In this case, AMF = 00000, indicating a No-Op for the ALU / MAC function

DREG selects the destination Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

I specifies the indirect address pointer (I register). M specifies the modify register (M register).

9 MOVE
DATA MEMORY WRITE (Direct Address)

Syntax: DM (<addr>) = reg ;

Permissible registers

AX0	MX0	SI	SB	CNTR
AX1	MX1	SE	PX	RX0
AY0	MY0	SR1	ASTAT	RX1
AY1	MY1	SR0	MSTAT	TX0
AR	MR2	I0-I7	SSTAT (read only)	TX1
	MR1	M0-M7	IMASK	
	MR0	L0-L7	ICNTL	

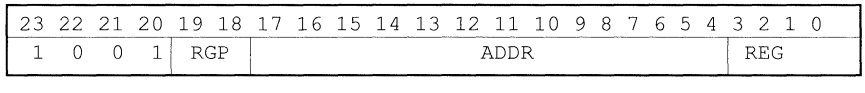
Example: DM (*cntl_port0*) = AR;

Description: Moves the contents of the source register to the data memory location specified in the instruction word. The addressing mode is direct addressing (designated by an immediate address value or by a label). The data memory address is stored directly in the instruction word as a full 14-bit field. Whenever a register less than 16 bits in length is written to memory, the value written is either sign-extended to the left if the source is a signed value, or zero-filled to the left if the source is an unsigned value. The unsigned registers which are zero-filled to the left are: I0 through I7, L0 through L7, CNTR, PX, ASTAT, MSTAT, SSTAT, IMASK, and ICNTL. All other registers are sign-extended to the left.

The contents of the source are always right-justified in the destination location after the write (bit 0 maps to bit 0).

Status Generated: None affected.

Instruction Format:
Data Memory Read (Direct Address), Instruction Type 3:



ADDR contains the direct address of the destination location in Data Memory.

RGP (Register Group) and REG (Register) select the destination register according to the Register Selection Table (see Appendix A).

Syntax: DM (I0 | I1 | I2 | I3 , M0 | M1 | M2 | M3) = dreg | <data> ;
I4 | I5 | I6 | I7 , M4 | M5 | M6 | M7

data: <constant>
'%' <identifier>
'^' <identifier>

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR1
AY1	MY1	SR0
AR	MR2	
	MR1	
	MR0	

Example: DM (I2, M0) = MR1;

Description: The Data Memory Write Indirect instruction moves the contents of the source to the data memory location specified in the instruction word. The immediate data may be a constant, or any identifier referenced with the "length of" (%) or "pointer to" (^) operators.

The addressing mode is register indirect with post-modify. When a register of less than 16 bits is written to memory, the value written is sign-extended to form a 16-bit value. The contents of the source are always right-justified in the destination location after the write (bit 0 maps to bit 0).

Status Generated: None affected.

9 MOVE
DATA MEMORY WRITE (Indirect Address)

Instruction Format:
ALU / MAC Operation with Data Memory Write, Instruction Type 4:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	G	1	0	AMF					0	0	0	0	0	DREG				I		M	

Data Memory Write, Immediate Data, Instruction Type 2:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	G	Data																I		M	

AMF specifies the ALU or MAC operation to be performed in parallel with the Data Memory Write. In this case, AMF = 00000, indicating a No-Op for the ALU / MAC function.

Data represents the actual 16-bit value.

DREG selects the source Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

G specifies which Data Address Generator the I and M registers are selected from. These registers must be from the same DAG as separated by the gray bar in the Syntax description above. I specifies the indirect address pointer (I register). M specifies the modify register (M register).

PROGRAM MEMORY WRITE (Indirect Address)

Syntax: PM (I4 | I5 | I6 | I7 , M4 | M5 | M6 | M7) = dreg ;

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR1
AY1	MY1	SR0
AR	MR2	
	MR1	
	MR0	

Example: PM (I6, M5) = AR;

Description: The Program Memory Write Indirect instruction moves the contents of the source to the program memory location specified in the instruction word. The addressing mode is register indirect with post-modify. The 16 most significant bits of the Program Memory Data bus (PMD23-8) are loaded from the source register, with bit PMD8 aligned with bit 0 of the source register (right justification). The 8 least significant bits of the Program Memory Data bus (PMD7-0) are loaded from the PX register. Whenever a source register of length less than 16 bits is written to memory, the value written is sign-extended to form a 16-bit value.

Status Generated: None affected.

Instruction Format:
ALU / MAC Operation with Program Memory Write, Instruction Type 5 (see Appendix A), as shown below:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	AMF					0	0	0	0	0	DREG				I		M	

AMF specifies the ALU or MAC operation to be performed in parallel with the Program Memory Write. In this case, AMF = 00000, indicating a No-Op for the ALU / MAC function.

DREG selects the source Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

I specifies the indirect address pointer (I register). M specifies the modify register (M register).

9 PROGRAM FLOW
JUMP

Syntax: [IF cond] JUMP | (I4) | (I5) | (I6) | (I7) | <addr> | ;

Permissible conds

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF NOT CE JUMP *top_loop*;

Description: Test the optional condition and, if true, then perform the specified jump. If the condition is not true then perform a no-operation. Omitting the condition performs the jump unconditionally. The JUMP instruction causes program execution to continue at the effective address specified by the instruction. The addressing modes available for the JUMP instruction are direct or register indirect.

If JUMP is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled. Consult your *User's Manual*.

For direct addressing (using an immediate address value or a label), the program address is stored directly in the instruction word as a full 14-bit field. For register indirect jumps, the selected I register provides the address; it is not post-modified in this case.

Status Generated: None affected.

Instruction Field:
Conditional JUMP Direct Instruction Type 10:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	ADDR														COND			

Conditional JUMP Indirect Instruction Type 19:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	I		0	0	COND			

I specifies the I register (Indirect Address Pointer).

ADDR: immediate jump address COND: condition

Syntax: [IF cond] CALL | (I4) / (I5) / (I6) / (I7) / <addr> | ;

Permissible conds

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF AV CALL *scale_down;*

Description: Test the optional condition and, if true, then perform the specified call. If the condition is not true then perform a no-operation. Omitting the condition performs the call unconditionally. The CALL instruction is intended for calling subroutines. CALL pushes the PC stack with the return address and causes program execution to continue at the effective address specified by the instruction. The addressing modes available for the CALL instruction are direct or register indirect.

If CALL is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled. Consult your *User's Manual.*

For direct addressing (using an immediate address value or a label), the program address is stored directly in the instruction word as a full 14-bit field. For register indirect jumps, the selected I register provides the address; it is not post-modified in this case.

Status Generated: None affected.

Instruction Field:
Conditional JUMP Direct Instruction Type 10:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	ADDR														COND			

Conditional JUMP Indirect Instruction Type 19:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	I		0	1	COND			

I specifies the I register (Indirect Address Pointer).

ADDR: immediate jump address COND: condition

9 PROGRAM FLOW
JUMP or CALL ON FLAG IN PIN

Syntax: IF | FLAG_IN / NOT FLAG_IN | | JUMP / CALL | | <addr> | ;

Example: IF FLAG_IN JUMP *service_proc_three;*

Description: Test the condition of the FI pin of the ADSP-2101 and, if set to one, perform the specified jump or call. If FI is zero then perform a no-operation. Omitting the flag in condition reduces the instruction to a standard ADSP-2101 JUMP or CALL instruction.

The JUMP instruction causes program execution to continue at the address specified by the instruction. The addressing mode for the JUMP on FI must be direct.

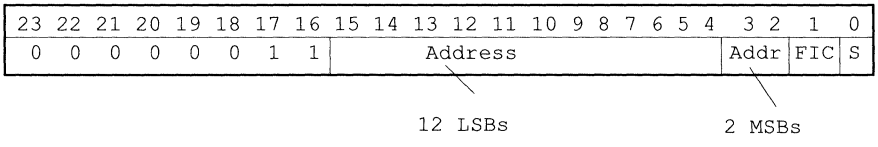
The CALL instruction is intended for calling subroutines. CALL pushes the PC stack with the return address and causes program execution to continue at the address specified by the instruction. The addressing mode for the CALL on FI must be direct.

If JUMP or CALL is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled. Consult your *User's Manual*.

For direct addressing (using an immediate address value or a label), the program address is stored directly in the instruction word as a full 14-bit field.

Status Generated: None affected.

Instruction Field:
Conditional JUMP or CALL on Flag In Direct Instruction Type 27:



S specifies JUMP (0) or CALL (1) FIC: Latched state of FI pin

Syntax: [IF cond] | SET / RESET / TOGGLE | FLAG_OUT ;

Example: IF MV RESET FLAG_OUT;

Description: Evaluate the optional condition and if true, set to one, reset to zero, or toggle the state of the FO pin of the ADSP-2101. Otherwise perform a no-operation and continue with the next instruction. Omitting the condition performs the operation unconditionally. Although this instruction does not directly alter the flow of your program, it is provided to signal external devices.

Status Generated: None affected.

Instruction Field:
Flag Out Mode Control Instruction Type 28:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	FO			COND		

FO: Operation to perform on FO pin COND: Condition code

9 PROGRAM FLOW
RTS

Syntax: [IF cond] RTS ;

Permissible conds

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF LE RTS ;

Description: Test the optional condition and, if true, then perform the specified return. If the condition is not true then perform a no-operation. Omitting the condition performs the return unconditionally. RTS executes a program return from a subroutine. The address on top of the PC stack is popped and is used as the return address. The PC stack is the only stack popped.

If RTS is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled. Consult the *User's Manual*.

Status Generated: None affected.

Instruction Field:
Conditional Return, Instruction Type 20:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	COND			

COND: condition

Syntax: [IF cond] RTI ;

Permissible conds

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF MV RTI ;

Description: Test the optional condition and, if true, then perform the specified return. If the condition is not true then perform a no-operation. Omitting the condition performs the return unconditionally. RTI executes a program return from an interrupt service routine. The address on top of the PC stack is popped and is used as the return address. The value on top of the status stack is also popped, and is loaded into the arithmetic status (ASTAT), mode status (MSTAT) and the interrupt mask (IMASK) registers.

If RTI is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled. Consult the *User's Manual*.

Status Generated: None affected.

Instruction Field:
Conditional Return, Instruction Type 20:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	COND			

COND: condition

9 PROGRAM FLOW DO UNTIL

Syntax: DO <addr> [UNTIL term] ;

Permissible terms

EQ	NE	GT	GE	LT	FOREVER
LE	NEG	POS	AV	NOT AV	
AC	NOT AC	MV	NOT MV	CE	

Example: DO *loop_label* UNTIL CE ;

Description: DO UNTIL sets up looping circuitry for zero-overhead looping. The program loop begins at the program instruction immediately following the DO instruction, ends at the address designated in the instruction and repeats execution until the specified condition is met (if a condition is specified) or repeats in an infinite loop (if no condition is specified). The condition is tested during execution of the last instruction in the loop, the status having been generated upon completion of the previous instruction. The address (<addr>) of the last instruction in the loop is stored directly in the instruction word.

When the DO instruction is executed, the address of the last instruction is pushed onto the loop stack along with the termination condition and the current program counter value plus 1 is pushed onto the PC stack.

Any nesting of DO loops continues the process of pushing the loop and PC stacks, up to the limit of the loop stack size (4 levels of loop nesting) or of the PC stack size (16 levels for subroutines plus interrupts plus loops). With either or both the loop or PC stacks full, a further attempt to perform the DO instruction will set the appropriate stack overflow bit and will perform a no-operation.

Status Generated:

ASTAT: Not affected.

SSTAT:

7	6	5	4	3	2	1	0
LSO	LSE	SSO	SSE	CSO	CSE	PSO	PSE
*	0	–	–	–	–	*	0

LSO Loop Stack Overflow: set if the loop stack overflows; otherwise not affected.

LSE Loop Stack Empty: always cleared (indicating loop stack not empty).

PSO — PC Stack Overflow: set if the PC stack overflows; otherwise not affected.

PSE — PC Stack Empty: always cleared (indicating PC stack not empty).

Instruction Format:

Do Until, Instruction Type 11:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	Addr														TERM			

ADDR specifies the address of the last instruction in the loop. In the Instruction Syntax, this field may be a program label or an immediate address value.

TERM specifies the termination condition, as shown below.

COND	*Syntax*	*Condition Tested*
0 0 0 0	NE	Not Equal to Zero
0 0 0 1	EQ	Equal Zero
0 0 1 0	LE	Less Than or Equal to Zero
0 0 1 1	GT	Greater Than Zero
0 1 0 0	GE	Greater Than or Equal to Zero
0 1 0 1	LT	Less Than Zero
0 1 1 0	NOT AV	Not ALU Overflow
0 1 1 1	AV	ALU Overflow
1 0 0 0	NOT AC	Not ALU Carry
1 0 0 1	AC	ALU Carry
1 0 1 0	POS	X Input Sign Positive
1 0 1 1	NEG	X Input Sign Negative
1 1 0 0	NOT MV	Not MAC Overflow
1 1 0 1	MV	MAC Overflow
1 1 1 0	CE	Counter Expired
1 1 1 1	FOREVER	Always

IDLE

Syntax: IDLE ;

Description: On an ADSP-2101 processor, IDLE loops indefinitely in a low-power state, waiting for interrupts. When an interrupt occurs it is serviced and execution continues with the instruction following IDLE. Typically this next instruction will be a JUMP back to IDLE, implementing a low-power standby loop. (Note the restrictions on JUMP as the last instruction in a DO UNTIL loop, detailed under the JUMP instruction earlier in this section.)

The serial port autobuffering operation continues during IDLE.

Status Generated: None affected.

Instruction Field:
Idle Instruction Type 31:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Syntax: [PUSH | STS] [, POP CNTR] [, POP PC] [, POP LOOP] ;
POP

Example: POP CNTR, POP PC, POP LOOP;

Description: Stack Control pushes or pops the designated stack(s). The entire instruction executes in one cycle regardless of how many stacks are specified.

The PUSH STS (Push Status Stack) instruction increments the status stack pointer by one to point to the next available status stack location; and pushes the arithmetic status (ASTAT), mode status (MSTAT), and interrupt mask register (IMASK) onto the processor's status stack. Note that the PUSH STS operation is executed automatically whenever an interrupt service routine is entered.

Any POP pops the value on the top of the designated stack and decrements the same stack pointer to point to the next lowest location in the stack. POP STS causes the arithmetic status (ASTAT), mode status (MSTAT), and interrupt mask (IMASK) to be popped into these same registers. This also happens automatically whenever a return from interrupt (RTI) is executed.

POP CNTR causes the counter stack to be popped into the down counter. When the loop stack or PC stack is popped (with POP LOOP or POP PC, respectively), the information is lost. Returning from an interrupt (RTI) also pops the PC counter automatically.

Status Generated:

SSTAT:

7	6	5	4	3	2	1	0
LSO	LSE	SSO	SSE	CSO	CSE	PSO	PSE
–	*	*	*	–	*	–	*

PSE — PC Stack Empty: cleared if a pop results in an empty program counter stack; set otherwise.

CSE — Counter Stack Empty: cleared if a pop results in an empty counter stack; set otherwise.

SSE — Status Stack Empty: for PUSH STS, this bit is always cleared (indicating status stack not empty). For POP STS, SSE is cleared if the pop results in an empty status stack; set otherwise.

SSO — Status Stack Overflow: for PUSH STS set if the status stack overflows; otherwise not affected.

LSE — Loop Stack Empty: cleared if a pop results in an empty loop stack; set otherwise.

Note that once any Stack Overflow occurs, the corresponding stack overflow bit is set in SSTAT, and this bit stays set indicating there has been loss of information. Once set, the stack overflow bit can only be cleared by resetting the processor.

Instruction Format:

Stack Control, Instruction Type 26:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Pp	Lp	Cp	Spp	

Pp: PC Stack Control
Cp: Counter Stack Control
Lp: Loop Stack Control
Spp: Status Stack Control

Syntax:

ENA DIS		BIT_REV AV_LATCH AR_SAT SEC_REG G_MODE M_MODE TIMER	[, ...] ;

Example: ENA AR_SAT, ENA M_MODE;

Description: Enables (ENA) or disables (DIS) the designated processor mode. The corresponding mode status bit in the mode status register (MSTAT) is set for ENAble mode and cleared for DISable mode. At RESET, MSTAT is set to zero, meaning that all modes are disabled. Any number of modes can be changed in one cycle with this instruction. Each ENA or DIS clause must be separated by a comma from the next.

MSTAT Bits:

0	SEC_REG	Alternate Register Data Bank
1	BIT_REV	Bit-Reverse Mode on Address Generator #1
2	AV_LATCH	ALU Overflow Status Latch Mode
3	AR_SAT	ALU AR Register Saturation Mode
4	M_MODE	MAC Result Placement Mode
5	TIMER	Timer Enable
6	G_MODE	Enables GO Mode

The data register bank select bit (SEC_REG) determines which set of data registers is currently active (0 = primary, 1 = secondary).

The bit-reverse mode bit (BIT_REV), when set to 1, causes addresses generated by Data Address Generator #1 to be output in bit reversed order.

The ALU overflow latch mode bit (AV_LATCH), when set to 1, causes the AV bit in the arithmetic status register to stay set once an ALU overflow occurs. In this mode, if an ALU overflow occurs, the AV bit will be set and will remain set even if subsequent ALU operations do not generate overflows. The AV bit can only be cleared by writing a zero into it directly over the DMD bus.

The AR saturation mode bit, (AR_SAT), when set to 1, causes the AR register to saturate if an ALU operation causes an overflow, as described in the ALU section of this document.

The ADSP-2101 MAC result placement mode (M_MODE) determines whether or not the left shift is made between the multiplier product and the MR register.

Setting the timer enable bit on the ADSP-2101 starts the timer decrementing logic. Clearing it halts the timer.

The "Go" mode allows the ADSP-2101 to continue executing instructions during a bus grant. In the microprocessor ADSP-2100 access to external memory was essential for fetching instructions and/or data. In the microcomputer ADSP-2101 this is often not true. The Go mode allows the processor to run; only if an external memory access is required does the processor halt, waiting for the bus to be released.

Instruction Format:

Mode Control, Instruction Type 18:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	TI		MM		AS		OL		BR		SR		GM		0	0

TI: Timer Enable
AS: AR Saturation Mode Control
BR: Bit Reverse Mode Control
GM: Go Mode
MM: Multiplier Placement
OL: ALU Overflow Latch Mode Control
SR: Secondary Register Bank Mode

Syntax: MODIFY (I0 | I1 | I2 | I3 , M0 | M1 | M2 | M3);

I4 | I5 | I6 | I7 , M4 | M5 | M6 | M7

Example: MODIFY (I1, M1);

Description: Add the selected M register (Mn) to the selected I register (Im), then process the modified address through the modulus logic with buffer length as determined by the L register corresponding to the selected I register (Lm), and store the resulting address pointer calculation in the selected I register. The I register is modified as if an indexed memory address were taking place, but no actual memory data transfer occurs.

The selection of the I and M registers is constrained to registers within the same Data Address Generator: selection of I0-I3 in Data Address Generator #1 constrains selection of the M registers to M0-M3. Similarly, selection of I4-I7 constrains the M registers to M4-M7.

Status Generated: None affected.

Instruction Format:
Modify Address Register, Instruction Type 21:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	G	I		M	

G specifies which Data Address Generator is selected. The I and M registers specified must be from the same DAG, separated by the gray bar above. I specifies the I register (depends on which DAG is selected by the G bit). M specifies the M register (depends on which DAG is selected by the G bit).

9 MISC
NOP

Syntax: NOP ;

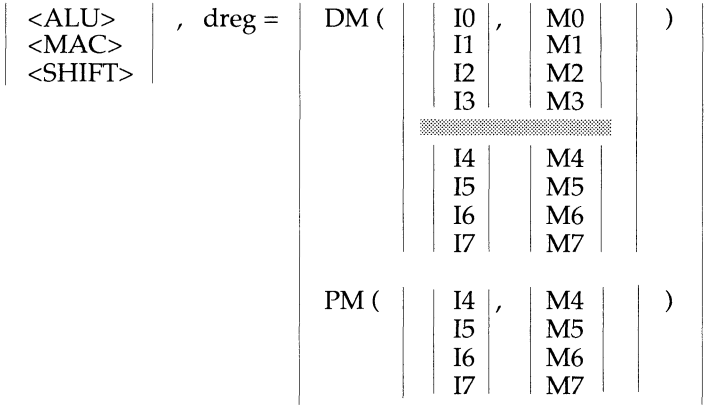
Description: No operation occurs for one cycle. Execution continues with the instruction following the NOP instruction.

Status Generated: None affected.

Instruction Format:
No operation, Instruction Type 30 (see Appendix A), as shown below:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Syntax:



Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR0
AY1	MY1	SR1
AR	MR0	
	MR1	
	MR2	

Description: Perform the designated arithmetic operation and data transfer. The read operation moves the contents of the source to the destination register. The addressing mode when combining an arithmetic operation with a memory read is register indirect with post-modify. The contents of the source are always right-justified in the destination register.

The computation must be unconditional. All ALU, MAC and Shifter operations are permitted except Shift Immediate and ALU DIVS and DIVQ instructions.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (computation first, memory read second) is intended to imply this. In fact, you may code this instruction with the order of clauses reversed. The Assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the Assembler (–s switch) the warning is not issued.

Because of the read-first, write-second characteristic of the processor, using the same register as source in one clause and a destination in the other is legal. The register supplies the value present at the beginning of the cycle and is written with the new value at the end of the cycle. Using the same register as a destination in both clauses, however, produces an indeterminate result and should not be done. The Assembler issues a warning unless semantics checking is turned off. Regardless of whether or not the warning is produced, however, this practice is not supported.

For example,

(1) AR = AX0 + AY0, AX0 = DM (I0, M0);

is a legal version of this multifunction instruction and is not flagged by the Assembler. Reversing the order of clauses, as in

(2) AX0 = DM (I0, M0) , AR = AX0 + AY0;

results in an Assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the data memory value is loaded into AX0 and then used in the computation, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

The following, therefore, is illegal and not supported, even though Assembler semantics checking produces only a warning:

(3) AR = AX0 + AY0, AR = DM (I0, M0); *Illegal!*

Status Generated: All status bits are affected in the same way as for the single function versions of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV	Set if the accumulated product overflows the lower-order 32 bits of the MR register. Cleared otherwise.

<SHIFT> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	-	-	-	-	-	-	-

SS	Affected only when executing the EXP operation; set if the source operand is negative. Cleared if the number is positive.

9 MULTIFUNCTION COMPUTATION with MEMORY READ

Instruction Format:

ALU/MAC operation with Data Memory Read, Instruction Type 4:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	G	0	Z	AMF					Yop		Xop			Dreg				I		M	

ALU/MAC operation with Program Memory Read, Instruction Type 5:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	Z	AMF					Yop		Xop			Dreg				I		M	

Shift operation with Data Memory Read, Instruction Type 12:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	G	0			SF					Xop				Dreg		I	M

Shift operation with Program Memory Read, Instruction Type 13:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	0			SF					Xop				Dreg		I	M

Z:	Result register	Dreg:	Destination register
SF:	Shifter operation	AMF:	ALU/MAC operation
Yop:	Y operand	Xop:	X operand
G:	Data Address Generator	I:	Indirect address register
M:	Modify register		

Syntax:

<ALU> <MAC> <SHIFT>	, dreg = dreg ;	

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR0
AY1	MY1	SR1
AR	MR0	
	MR1	
	MR2	

Description: Perform the designated arithmetic operation and data transfer. The contents of the source are always right-justified in the destination register after the read.

The computation must be unconditional. All ALU, MAC and Shifter operations are permitted except Shift Immediate and ALU DIVS and DIVQ instructions.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (computation first, register transfer second) is intended to imply this. In fact, you may code this instruction with the order of clauses reversed. The Assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the Assembler (–s switch) the warning is not issued.

Because of the read-first, write-second characteristic of the processor, using the same register as source in one clause and a destination in the other is legal. The register supplies the value present at the beginning of the cycle and is written with the new value at the end of the cycle. Using the same register as a destination in both clauses, however, produces an indeterminate result and should not be done. The Assembler issues a warning unless semantics checking is turned off. Regardless of whether or not the warning is produced, however, this practice is not supported.

For example,

(1) AR = AX0 + AY0, AX0 = MR1;

is a legal version of this multifunction instruction and is not flagged by the Assembler. Reversing the order of clauses, as in

(2) AX0 = MR1, AR = AX0 + AY0;

results in an Assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the MR1 register value is loaded into AX0 and then AX0 is used in the computation, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

The following, therefore, is illegal and not supported, even though Assembler semantics checking produces only a warning:

(3) AR = AX0 + AY0, AR = MR1; *Illegal!*

Status Generated: All status bits are affected in the same way as for the single function versions of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV — Set if the accumulated product overflows the lower-order 32 bits of the MR register. Cleared otherwise.

<SHIFT> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	-	-	-	-	-	-	-

SS — Affected only when executing the EXP operation; set if the source operand is negative. Cleared if the number is positive.

Instruction Format:

ALU/MAC operation with Data Register Move, Instruction Type 8:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Z	AMF					Yop		Xop			Dreg				Dreg			
													destination							source			

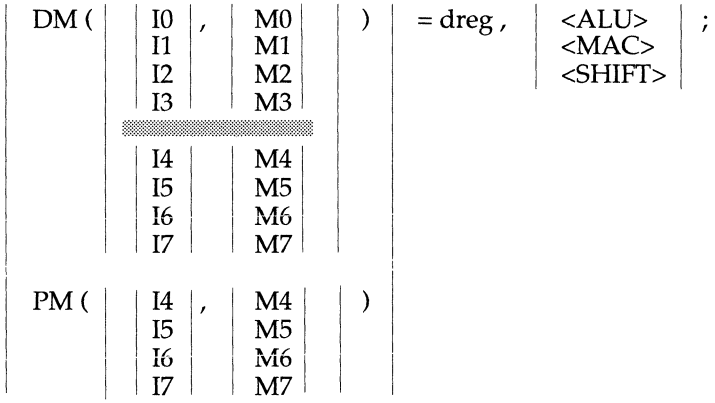
Shift operation with Data Register Move, Instruction Type 14:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0			SF		Xop			Dreg				Dreg			
													destination							source			

Z:	Result register	Dreg:	Destination register
SF:	Shifter operation	AMF:	ALU/MAC operation
Yop:	Y operand	Xop:	X operand

9 MULTIFUNCTION COMPUTATION with MEMORY WRITE

Syntax:



Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR0
AY1	MY1	SR1
AR	MR0	
	MR1	
	MR2	

Description: Perform the designated arithmetic operation and data transfer. The write operation moves the contents of the source to the specified memory location. The addressing mode when combining an arithmetic operation with a memory write is register indirect with post-modify. The contents of the source are always right-justified in the destination register.

The computation must be unconditional. All ALU, MAC and Shifter operations are permitted except Shift Immediate and ALU DIVS and DIVQ instructions.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (memory write first, computation second) is intended to imply this. In fact, you may code this instruction with the order of clauses reversed. The Assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the Assembler (–s switch) the warning is not issued.

Because of the read-first, write-second characteristic of the processor, using the same register as destination in one clause and a source in the other is legal. The register supplies the value present at the beginning of the cycle and is written with the new value at the end of the cycle.

For example,

(1) DM (I0, M0) = AR, AR = AX0 + AY0;

is a legal version of this multifunction instruction and is not flagged by the Assembler. Reversing the order of clauses, as in

(2) AR = AX0 + AY0, DM (I0, M0) = AR;

results in an Assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the result of the computation in AR is then written to memory, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

Status Generated: All status bits are affected in the same way as for the single function versions of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV — Set if the accumulated product overflows the lower-order 32 bits of the MR register. Cleared otherwise.

<SHIFT> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	-	-	-	-	-	-	-

SS — Affected only when executing the EXP operation; set if the source operand is negative. Cleared if the number is positive.

Instruction Format:
ALU/MAC operation with Data Memory Write, Instruction Type 4:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	G	1	Z	AMF					Yop		Xop			Dreg				I		M	

ALU/MAC operation with Program Memory Write, Instruction Type 5:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	Z	AMF					Yop		Xop			Dreg				I		M	

Shift operation with Data Memory Write, Instruction Type 12:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	G	1	SF				Xop			Dreg				I		M	

Shift operation with Program Memory Write, Instruction Type 13:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	1	SF				Xop			Dreg				I		M	

Z:	Result register	Dreg:	Destination register
SF:	Shifter operation	AMF:	ALU/MAC operation
Yop:	Y operand	Xop:	X operand
I:	Indirect address register	M:	Modify register

G: Data Address Generator; I & M registers must be from the same DAG, as separated by the gray bar in the Syntax description.

9 MULTIFUNCTION DATA & PROGRAM MEMORY READ

Syntax:

AX0 AX1 MX0 MX1	= DM (	I0 I1 I2 I3	,	M0 M1 M2 M3	),	AY0 AY1 MY0 MY1	= PM (	I4 I5 I6 I7	,	M4 M5 M6 M7	);

Description: Perform the designated memory reads, one from data memory and one from program memory. Each read operation moves the contents of the memory location to the destination register. For this double data fetch, the destinations for data memory reads are the X registers in the ALU and the MAC, and the destinations for program memory reads are the Y registers. The addressing mode for this memory read is indirect with post-modify. The contents of the source are always right-justified in the destination register.

For information on extra cycle conditions, refer to the Instruction Set Overview at the beginning of this chapter.

Status Generated: No status bits are affected.

Instruction Format:
ALU/MAC with Data & Program Memory Read, Instruction Type 1:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	PD			DD				AMF			0	0	0	0	0	PM I		PM M		DM I		DM M

AMF specifies the ALU or MAC function. In this case, AMF = 00000, designating a no-operation for the ALU or MAC function.

PD: Program Destination register
DD: Data Destination register
AMF: ALU/MAC operation
I: Indirect address register
M: Modify register

ALU / MAC with DATA & PROGRAM MEMORY READ

Syntax:

<ALU>	,	AX0	= DM (	I0	,	M0	),	AY0	= PM (	I4	,	M4	);
<MAC>		AX1		I1		M1		AY1		I5		M5	
		MX0		I2		M2		MY0		I6		M6	
		MX1		I3		M3		MY1		I7		M7	

Description: This instruction combines an ALU or a MAC operation with a data memory read and a program memory read. The read operations move the contents of the memory location to the destination register. For this double data fetch, the destinations for data memory reads are the X registers in the ALU and the MAC, and the destinations for program memory reads are the Y registers. The addressing mode is register indirect with post-modify. The contents of the source are always right-justified in the destination register after the read.

The computation must be unconditional. All ALU and MAC operations are permitted except the DIVS and DIVQ instructions. The results of the computation must be written into the R register of the computational unit; ALU results to AR, MAC results to MR.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (computation first, memory reads second) is intended to imply this. In fact, you may code this instruction with the order of clauses altered. The Assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the Assembler (–s switch) the warning is not issued.

The same data register may be used as a source for the arithmetic operation and as a destination for the memory read. The register supplies the value present at the beginning of the cycle and is written with the value from memory at the end of the cycle.

For example,

(1) MR=MR+MX0*MY0(UU), MX0=DM(I0, M0), MY0=PM(I4,M4);

is a legal version of this multifunction instruction and is not flagged by the Assembler. Changing the order of clauses, as in

(2) MX0=DM(I0, M0), MY0=PM(I4,M4), MR=MR+MX0*MY0(UU);

results in an Assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the data memory value is loaded into MX0 and MY0 and subsequently used in the computation, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

Status Generated: All status bits are affected in the same way as for the single operation version of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV	Set if the accumulated product overflows the lower-order 32-bits of the MR register. Cleared otherwise.

Instruction Format:
ALU/MAC with Data and Program Memory Read, Instruction Type 1:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	PD		DD		AMF					Yop			Xop			PMI		PMM		DMI		DMM

PD: Program Destination register
DD: Data Destination register
AMF: ALU/MAC operation
M: Modify register
Yop: Y operand
Xop: X operand
I: Indirect address register

Instruction Coding A

A.1 OPCODES

Here is a summary of the complete instruction set of the ADSP-2101. Following the list of types and codes shown immediately below is a key to the abbreviations used. Any instruction codes not shown are reserved for future use.

Type 1: ALU / MAC with Data & Program Memory Read

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	PD		DD		AMF					Yop		Xop			PM I		PM M		DM I		DM M	

Type 2: Data Memory Write (Immediate Data)

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	G	DATA																I		M	

Type 3: Read /Write Data Memory (Immediate Address)

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	D	RGP		ADDR														REG			

Type 4: ALU / MAC with Data Memory Read / Write

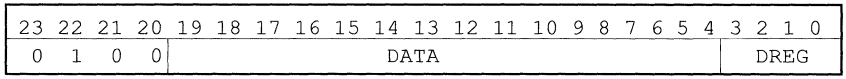
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	G	D	Z	AMF					Yop		Xop			DREG				I		M	

Type 5: ALU / MAC with Program Memory Read / Write

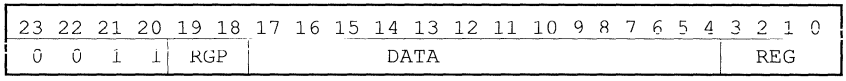
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	D	Z	AMF					Yop		Xop			DREG				I		M	

A Instruction Coding

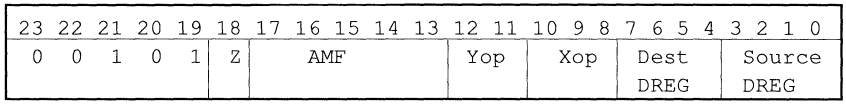
Type 6: Load Data Register Immediate



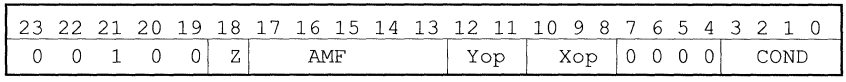
Type 7: Load Non-Data Register Immediate



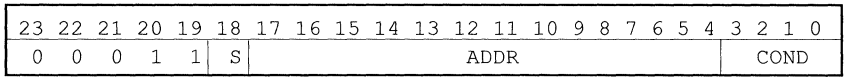
Type 8: ALU / MAC with Internal Data Register Move



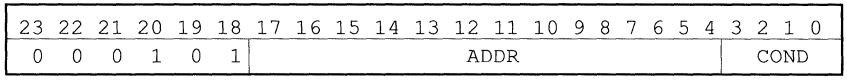
Type 9: Conditional ALU / MAC



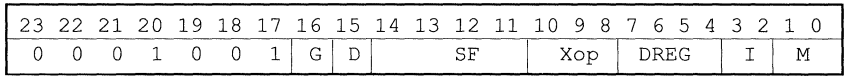
Type 10: Conditional Jump (Immediate Address)



Type 11: Do Until



Type 12: Shift with Data Memory Read / Write



Instruction Coding A

Type 13: Shift with Program Memory Read / Write

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	D	SF				Xop			DREG				I		M	

Type 14: Shift with Internal Data Register Move

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	SF				Xop			Dest REG				Source REG			

Type 15: Shift Immediate

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	SF				Xop			exponent							

Type 16: Conditional Shift

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

Type 17: Internal Data Move

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	0	0	DST RGP		SRC RGP		Dest REG				Source REG			

Type 18: Mode Control

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	TI		MM		AS		OL		BR		SR		GM		0	0

Explanation of these codes can be found together alphabetically under "Mode Control" in the next section.

A Instruction Coding

Type 19: Conditional Jump (Indirect Address)

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	I		0	S	COND			

Type 20: Conditional Return

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	T	COND			

Type 21: Modify Address Register

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	G	I		M	

Type 22: Reserved

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Type 23: DIVQ

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	0	0	0	1	0	Xop			0	0	0	0	0	0	0	0

Type 24: DIVS

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	0	Yop		Xop			0	0	0	0	0	0	0	0

Type 25: Saturate MR

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction Coding A

Type 26: Stack Control

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	PP	LP	CP	SPP	

Type 27: Call or Jump on Flag In

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	Address												Addr		FIC	S

12 LSBs 2 MSBs

Type 28: Flag Out Mode Control

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	FO		COND			

Type 29: Reserved

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Type 30: No Operation

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Type 31: Idle

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A Instruction Coding

A.2 ABBREVIATION CODING

AMF ALU / MAC Function codes

0 0 0 0 0	No operation		

MAC Function codes

0 0 0 0 1	X * Y	(RND)	
0 0 0 1 0	MR + X * Y	(RND)	
0 0 0 1 1	MR – X * Y	(RND)	
0 0 1 0 0	X * Y	(SS)	Clear when y = 0
0 0 1 0 1	X * Y	(SU)	
0 0 1 1 0	X * Y	(US)	
0 0 1 1 1	X * Y	(UU)	
0 1 0 0 0	MR + X * Y	(SS)	
0 1 0 0 1	MR + X * Y	(SU)	
0 1 0 1 0	MR + X * Y	(US)	
0 1 0 1 1	MR + X * Y	(UU)	
0 1 1 0 0	MR – X * Y	(SS)	
0 1 1 0 1	MR – X * Y	(SU)	
0 1 1 1 0	MR – X * Y	(US)	
0 1 1 1 1	MR – X * Y	(UU)	

ALU Function codes

1 0 0 0 0	Y	Clear when y = 0
1 0 0 0 1	Y + 1	
1 0 0 1 0	X + Y + C	
1 0 0 1 1	X + Y	X when y = 0
1 0 1 0 0	NOT Y	
1 0 1 0 1	– Y	
1 0 1 1 0	X – Y + C – 1	
1 0 1 1 1	X – Y	
1 1 0 0 0	Y – 1	
1 1 0 0 1	Y – X	– X when y = 0
1 1 0 1 0	Y – X + C – 1	

1 1 0 1 1	NOT X
1 1 1 0 0	X AND Y
1 1 1 0 1	X OR Y
1 1 1 1 0	X XOR Y
1 1 1 1 1	ABS X

COND Status Condition codes

0 0 0 0	Equal	EQ
0 0 0 1	Not equal	NE
0 0 1 0	Greater than	GT
0 0 1 1	Less than or equal	LE
0 1 0 0	Less than	LT
0 1 0 1	Greater than or equal	GE
0 1 1 0	ALU Overflow	AV
0 1 1 1	NOT ALU Overflow	NOT AV
1 0 0 0	ALU Carry	AC
1 0 0 1	Not ALU Carry	NOT AC
1 0 1 0	X input sign negative	NEG
1 0 1 1	X input sign positive	POS
1 1 0 0	MAC Overflow	MV
1 1 0 1	Not MAC Overflow	NOT MV
1 1 1 0	Not counter expired	NOT CE
1 1 1 1	Always	FOREVER

CP Counter Stack Pop codes

0	No change
1	Pop

D Memory Access Direction codes

0	Read
1	Write

A Instruction Coding

DD Double Data Fetch Data Memory Destination codes

0 0	AX0
0 1	AX1
1 0	MX0
1 1	MX1

DREG Data Register codes

0 0 0 0	AX0
0 0 0 1	AX1
0 0 1 0	MX0
0 0 1 1	MX1
0 1 0 0	AY0
0 1 0 1	AY1
0 1 1 0	MY0
0 1 1 1	MY1
1 0 0 0	SI
1 0 0 1	SE
1 0 1 0	AR
1 0 1 1	MR0
1 1 0 0	MR1
1 1 0 1	MR2
1 1 1 0	SR0
1 1 1 1	SR1

FIC FI condition code

1	latched FI is 1	FLAG_IN
0	latched FI is 0	NOT FLAG_IN

Instruction Coding A

FO Mode Control codes for Flag Out pin

FO:	Set, reset, or toggle the output Flag.
0 0	No change
0 1	Toggle
1 0	Reset
1 1	Set

G Data Address Generator codes

0	DAG1
1	DAG2

I Index Register codes

G =	0	1
0 0	I0	I4
0 1	I1	I5
1 0	I2	I6
1 1	I3	I7

LP Loop Stack Pop codes

0	No Change
1	Pop

M Modify Register codes

G =	0	1
0 0	M0	M4
0 1	M1	M5
1 0	M2	M6
1 1	M3	M7

A Instruction Coding

Mode Control codes

SR:	Secondary register bank
BR:	Bit-reverse mode
OL:	ALU overflow latch mode
AS:	AR register saturate mode
MM:	Alternate Multiplier placement mode
GM:	GOMode; enable means go if possible
TI:	Timer enable

0 0	No change
0 1	No change
1 0	Deactivate
1 1	Activate

PD Double Data Fetch Program Memory Destination codes

0 0	AY0
0 1	AY1
1 0	MY0
1 1	MY1

PP PC Stack Pop codes

0	No Change
1	Pop

Instruction Coding A

REG Register codes

RGP =	00	01	10	11
0 0 0 0	AX0	I0	I4	ASTAT
0 0 0 1	AX1	I1	I5	MSTAT
0 0 1 0	MX0	I2	I6	SSTAT
0 0 1 1	MX1	I3	I7	IMASK
0 1 0 0	AY0	M0	M4	ICNTL
0 1 0 1	AY1	M1	M5	CNTR
0 1 1 0	MY0	M2	M6	SB
0 1 1 1	MY1	M3	M7	PX
1 0 0 0	SI	L0	L4	RX0
1 0 0 1	SE	L1	L5	TX0
1 0 1 0	AR	L2	L6	RX1
1 0 1 1	MR0	L3	L7	TX1
1 1 0 0	MR1	–	–	IFC (write only)
1 1 0 1	MR2	–	–	OWRCNTR (write only)
1 1 1 0	SR0	–	–	–
1 1 1 1	SR1	–	–	–

S Jump Type codes

0	Jump
1	Call

SF Shifter Function codes

0 0 0 0	LSHIFT	(HI, PASS)
0 0 0 1	LSHIFT	(HI, OR)
0 0 1 0	LSHIFT	(LO, PASS)
0 0 1 1	LSHIFT	(LO, OR)
0 1 0 0	ASHIFT	(HI, PASS)
0 1 0 1	ASHIFT	(HI, OR)
0 1 1 0	ASHIFT	(LO, PASS)
0 1 1 1	ASHIFT	(LO, OR)
1 0 0 0	NORM	(HI, PASS)

A Instruction Coding

1 0 0 1	NORM	(HI, OR)
1 0 1 0	NORM	(LO, PASS)
1 0 1 1	NORM	(LO, OR)
1 1 0 0	EXP	(HI)
1 1 0 1	EXP	(HIX)
1 1 1 0	EXP	(LO)
1 1 1 1	Derive Block Exponent	

SPP Status Stack Push/Pop codes

0 0	No change
0 1	No change
1 0	Push
1 1	Pop

T Return Type codes

0	Return from Subroutine
1	Return from Interrupt

X X Operand codes

0 0 0	X0 (SI for shifter)
0 0 1	X1 (invalid for shifter)
0 1 0	AR
0 1 1	MR0
1 0 0	MR1
1 0 1	MR2
1 1 0	SR0
1 1 1	SR1

Y Y Operand codes

0 0	Y0
0 1	Y1
1 0	F (feedback register)
1 1	zero

Z ALU/MAC Result Register codes

0	Result register
1	Feedback register

File Formats B

B.1 DATA FILES (.DAT)

The .DAT file format is used for data buffer initialization in source code, simulated parallel and serial port data flow, and saving or loading simulated memory. The format is generally the same for all uses, with some restrictions as detailed in this section. The .DAT extension is a DOS convention only and is not required by the Linker or Simulator.

These files contain text only: the characters for hexadecimal, decimal, and binary data. Any standard text editor can be used to create the files.

B.1.1 Assembler Buffer Initialization Files

The .INIT Assembler directive names an external file from which to initialize a data buffer. This file is created by the user and specified in source code with the .INIT directive. The data file should be located in the directory from which the Linker is invoked, or the path specifying the directory which does contain the file must be given in the .INIT directive. The Linker reads this file and loads the buffers via the .EXE memory image file. The file can be of any length.

B.1.1.1 Integer Data

The standard format of this file is a single four- or six-character hexadecimal number per line of input (carriage returns are ignored). If a file for DM contains six characters per line, however, the four most significant digits of each number are used and the other two are ignored.

Files for PM are typically six characters per line. If a line in the file contains only four characters, the number is left-justified and zero-filled to the right.

For example, if your data file contained these lines:

```
002222
2222
220001
2211
```

The contents if loaded in DM would be:

```
0022
2222
2200
2211
```

The contents if loaded in PM would be:

```
002222
222200      (zero-filled)
220001
221100      (zero-filled)
```

B.1.1.2 Non-Integer Data

Buffer initialization files can contain decimal numbers with fractional components. These non-integer (or floating point) values are treated by the Linker as integers; the fractional component is ignored (not rounded). For example, the data value

```
5.862
```

is loaded in memory as

```
5
```

B.1.1.3 Comments

Comments can be entered on each line in a buffer initialization file, anywhere to the right of the data. The comments do not need to be enclosed in brackets; anything other than data is ignored by the Linker.

B.1.2 Simulator Data Files

The Simulator requires external files to simulate input and output data for memory-mapped I/O ports, to simulate receive and transmit data for serial ports, and to store or load simulated processor memory with the D and E commands.

B.1.2.1 I/O Port Data

You must create this file to provide input data to parallel ports in DM or PM. The file format is one four-character hexadecimal number per line for DM-mapped ports, and one six-character hexadecimal number for PM-mapped ports. Carriage returns are ignored by the Simulator.

If data is written out to the port, the Simulator opens a file to store the output data.

B.1.2.2 SPORT Data

A file must also be created to provide simulated (receive) serial data to the serial ports. The input file must consist entirely of ones, zeros, and carriage returns. A single line in the file may have any number of ones and zeros; carriage returns may be used to terminate each line and make the file more readable. The Simulator reads the ASCII characters and ignores carriage returns.

The Simulator opens a file to store output (transmit) data.

B.1.2.3 Simulated Memory Data

Portions of PM or DM can be saved to or loaded from an external file by the Simulator with the use of the D (dump memory) and E (enter memory) commands. The file format for data to be loaded with the E command is the same as that for I/O port data, described above; four-character hex numbers for DM and six-character numbers for PM. The files saved to with the D command are useful for intermediate storage of simulated PM or DM contents to be reloaded by the Simulator later or input to other software applications.

B.2 MEMORY IMAGE FILE (.EXE)

The PM/DM/BM Memory Image File is created by the Linker. This file contains the memory images of the system that the user has created using the development system. Memory images include assembled and resolved opcodes and initialized data buffers. The Memory Image File is used to upload executable code to the ADSP-2101 Simulator, Emulator and PROM Splitter.

The first three characters of this file are "<ESC> <ESC> i". These characters tell the Emulator that the following code is upload information for PM, DM, and BM.

Each "kernel" of information that can be uploaded into a particular area of memory is headed by one of the following: @PA, @PO, @DA, @DO, @BO. The P indicates program memory; D indicates data memory; B indicates boot memory; O indicates ROM; A indicates RAM.

B File Formats

Following this header is a four-character hexadecimal address where upload starts. Only areas of memory that need to be loaded will have kernels associated with them. Following this is the series of words to be uploaded sequentially from the start address. These words are six characters for PM and BM, and four characters for DM, all hexadecimal.

Each kernel is terminated by *#nnnnnn...*, which is a dummy value separating kernels. Kernels can occur in any order.

This file is closed by "<ESC> <ESC> o" to tell the Emulator that the upload is complete. <ESC> sequences are ignored by the Simulator and the PROM Splitter.

An example .EXE Image File is shown below:

```
<ESC> <ESC>i
@PO
0004
1C007F
1C05BF
0A000F
#123123123123
@DA
0000
2D40
0000
#123123123123
@BO
0000
03242A
025B26
01921C
#123123123123
@PO
6000
7FFFFF
7FFD88
80009E
#123123123123
@BO
0800
0A000F
FD887F
```

```
025B26
#123123123123
@DA
0182
0040
#123123123123
@DA
0183
0000
#123123123123
<ESC> <ESC>o
```

Notice that the boot memory words in this file are only 24 bits wide, instead of 32 bits wide as in the boot memory PROMS. The Linker does not generate 32-bit boot memory words; this is done by the PROM Splitter (which reads the .EXE file).

Acccordingly, the boot memory addresses in the .EXE file are word addresses, not PROM byte addresses. These word addresses are similar to program memory addresses; each address locates a 24-bit word of code or data.

The Linker views boot memory as an array of words, of length 16K, and divided into 8 pages of 2K each. The page number is embedded in the @BO address: 0000 is the start of page 0, 0800 is the start of page 1, 1000 is the start of page 2, etc.. To obtain the page number, divide the hex address by 2K and drop the remainder.

B.3 DEBUG SYMBOL TABLE FILE (.SYM)

The Debug Symbol Table File (.SYM) is created by the Linker. It lists all of the symbols and associated addresses encountered by the Linker. A separate list is generated for each module which includes all symbols that can be referenced by that module.

The first three characters of this file are "<ESC> <ESC> d". These characters tell the Emulator that the following code is the Debug Symbol information.

The "m directive" is used to specify the source code module which is the scope of a set of symbols. The "m directive" takes the form

_m*modulename* addr

where *modulename* is the name of the module and *addr* is the module base address, in hexadecimal. A p, d, or b# is appended to indicate placement in PM, DM, or BM. The boot page number is listed following the b, as in "b0".

On lines following this directive are the symbols which can be referenced within the context of the named module. With each symbol is the associated hex address and an indication of the memory to which it belongs (d, p, or b#). A ZZZZ in the address field indicates that the address for that symbol was never resolved (not all symbols have memory addresses associated with them).

The file is closed by "<ESC> <ESC> o" to tell the Emulator that the Debug Symbol File transmission is complete. <ESC> sequences are ignored by the Simulator.

The following is an example of a Debug Symbol Table File for a single module in program memory which is not booted:

```
<ESC> <ESC>d
_mFFT 0004p
REAL_OUTPUT 1000d
IMAGINARY_OUTPUT 2000d
MAGN 0100d
SIN_COEF 6000p
COS_COEF 6080p
FFT_START 0020p
BUTTERFLY_LOOP 0033p
GROUP_LOOP 0037p
STAGE_LOOP 0040p
<ESC> <ESC>o
```

The following example shows the .SYM file for this manual's example program (see Chapter 3, Assembler), consisting of two modules on boot page 0:

```
<ESC> <ESC>d
_mFIR_ROUTINE 000Fb0
FIR_START 000Fb0
CONVOLUTION 0014b0
COEFFICIENT 0000b0
DATA_BUFFER 3800d
_mMAIN_ROUTINE 0019b0
```

```
DATA_BUFFER 3800d
COEFFICIENT 0000b0
RESTARTER 0035b0
CLEAR_BUFFER 003Db0
WAIT 0052b0
FIR_START 000Fb0
<ESC> <ESC>o
```

B.4 PROM IMAGE FILES (.BNU, .BNM, .BNL)

PROM Image files are generated by the ADSP-2101 PROM Splitter. They are uploaded to a PROM burner to program the appropriate PROMs. One file is needed for each PROM chip to be programmed. The format of the files depends on the options specified in the PROM Splitter invocation command line. See Chapter 8.

There are three types of image files generated by the PROM Splitter: .BNU for the upper bytes of the 24-bit words, .BNM for the middle bytes, and .BNL for the lower bytes.

The files created can be in either Intel Hex I format or Motorola S format. An overview of these formats is given in the following sections.

B.4.1 Intel Format

The example files below show the Intel format for a program memory file (middle byte), a program memory single stream file, and a boot page memory file. Each line of the file is a data record with the exception of the last line, which is the end of file record. Larger files contain additional data records.

Program Memory .BNM sample file:

```
:0A0004003C4034343426142226 0850    data record
:00000401FB                         end of file record
```

B File Formats

This file format is obtained by using the –pm and –i switches; the file contains the middle byte of program memory words. The records are organized into the following fields:

```
:0A0004003C40343434261422260850
```

:	start character
0A	byte count in this record
0004	address of the first data byte in this record
00	record type (00)
3C	first data byte
08	last data byte
50	checksum: Twos complement negation of binary summation (least significant 8 bits) of preceding bytes, including byte count, address and data bytes.

```
:00000401FB
```

:	start character
00	byte count (zero in this record)
0004	address of the first byte in this record
01	record type (01 in this record)
FB	checksum

Program Memory .BNM Single Stream file:

```
:1E0000003C0055400080340000340014340008261 80F14 00C222E00F26300208000F36    data record
:00001F01E0                                                                   end of file record
```

This file format is obtained by using the –pm and –ui switches; the file contains all three bytes of program memory words. The fields are the same. Note that every third data byte (shown in bold) corresponds to the middle byte example above.

Boot Memory .BNM sample file:

:200000001111100**0A**000100FF001100FF011100FF111100FF100000FF110000FF111000FF33	data record
:20002000000000FF000100FF110000FF111000FF000100FF001100FF011100FF3C00E5FF50	data record
:200040000D0388FF680080FFE89800FF14014EFFE90000FF20400FFF050000FF0D0C9CFF33	data record
:200060000A001FFF18035FFF000000FF000000FF000000FF0A001FFF000000FF000000FFBC	data record
:20008000000000FF0A001FFF000000FF000000FF000000FF1800FFFF000000FF000000FF28	data record
:2000A000000000FF0A001FFF000000FF000000FF000000FF0A001FFF000000FF000000FFF6	data record
:2000C000000000FF0A001FFF000000FF000000FF000000FF3400F8FF3800F8FF340014FF5B	data record
:2000E000380014FF378000FF380000FF3C00F5FF1403DEFFA00000FF37FEF1FFA00004FF3D	data record
:20010000A00004FFA00004FFA00004FFA00004FFA00BF4FFA00034FFA69274FFA00004FF94	data record
:20012000A00004FFA00004FFA00004FFA00004FFA00004FFA00004FFA70004FFA10004FF9F	data record
:2000A0003C0004FF3C0183FF028000FF18052FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE5	data record
:00000001FF	end of file record

This file format is obtained by using the –bm and –i switches; the file contains the four bytes of boot memory words. The records have the same fields as specified above with the following additions:

- Every fourth data byte is a pad byte (FF) inserted to produce the 32-bit word width.
- The pad byte of the first data word (0A), at PROM byte address 0x0003, is the page length for this boot page. This byte is shown in bold above.
- Pad bytes are added to the last data record to make the number of words on the page a multiple of 8.

B File Formats

B.4.2 Motorola Format

The Motorola standard is quite similar. The following example shows an S-style .BNM file consisting of the middle byte of program memory. Each line of the file is a data record with the exception of the last line, which is the end of file record. Larger files contain additional data records.

Program Memory .BNM sample file:

```
S10D00043C4034343426142226084C      data record
S903000DEF                          end of file record
```

This file format is obtained by using the –pm and –s switches. The records are organized into the following fields:

```
S10D00043C4034343426142226084C

S1                                  start character
  0D                                byte count in this record
    0004                            address of the first byte in this
                                    record
        3C                          first data byte
                            08      last data byte
                              4C    checksum: One's complement of
                                    binary summation (least significant
                                    8 bits) of preceding bytes, including
                                    byte count, address and data bytes.

S903000DEF

S9                                  start character
  03                                byte count in this record
    000D                            address of the first byte in this
                                    record
        EF                          checksum
```

The Motorola format may also be used to create a single-stream file containing all three bytes of program memory; this is the S2 format. See the Chapter "PROM Splitter," for the complete set of options.

Host-Specific Requirements C

C.1 SYSTEM REQUIREMENTS

This appendix details the requirements (hardware and software) and restrictions for each of the environments in which you can operate the ADSP-2101 Cross-Software.

C.2 IBM PC AND COMPATIBLES

The Cross-Software for IBM PCs executes on all PC models. You must have a hard disk and it is recommended that you use at least an AT-class model. Installation of the Cross-Software on a PC requires the following:

- PC-DOS 3.0 or later
- 640KB memory
- The directive "FILES=25" in your CONFIG.SYS (configuration boot) file. This file must be in the root directory of the startup disk.

In addition, for graphics output when using the PL (plot memory) command in the Simulator, you must have:

- A color display system: IBM CGA, EGA, or VGA-type. Monochrome and Hercules-type displays will not work.

There are several operating restrictions that result from memory limitations on the PC:

- The size of the source file processed by the the Assembler may be limited by memory.
- The number and size of modules and number of symbols that can be handled by the Linker is limited by memory.
- Do not run any memory-resident programs like Sidekick or ProKey while you are using the Cross-Software.

C Host-Specific Requirements

C.3 SUN-3 WORKSTATION

The Sun-3 Cross-Software must be run under a version of Unix supporting virtual memory.

ANSI Standard C D

D.1 ANSI DRAFT STANDARD EXCEPTIONS

You can obtain a copy of the draft standard by writing to the Computer and Business Equipment Manufacturers Association at:

CBEMA
Suite 500
311 First Street N.W.
Washington, DC 20001-2178

The ADSP-21XX C Compiler and Preprocessor adhere to the current ANSI draft standard (X3J11) except as noted below.

D.1.1 Features Not Supported & Restrictions

1. Static and global variables are not automatically initialized to zero. The initial value of such variables is undefined.
2. Passing structures in a function call
3. Returning structures from a function call
4. Objects cannot exceed 8K bytes in size
5. Floating-point does not meet the precision specified

D.1.2 New Features and Extensions

1. The *fastswitch* keyword.
2. The *fract* numeric type.
3. The storage class modifiers *pm, dm, ram* and *rom.*

D ANSI Standard C

D.2 DIFFERENCES BETWEEN HOST VERSIONS

There are no specific differences in the ADSP-21XX C Compiler, although there may be differences in other Cross-Software modules' performance that affects the assembly or linking of code generated by the C Compiler. See also Appendix C.

Linker Operation E

E.1 INTRODUCTION

The ADSP-2101 boot memory space may contain up to eight individual boot pages. Software selection of the next page to be booted and software-forced booting, described in the Memory Interface chapter of the *ADSP-2101 User's Manual*, allow the processor to be rebooted under program control. This allows an application to be implemented in multiple boot pages, with execution branching from one page to another. This branching, of course, takes the form of software reboots.

This appendix will only be of interest to you if your application requires multiple boot pages of ADSP-2101 code.

E.2 RE-BOOTING UNDER PROGRAM CONTROL

Programs requiring multiple boot pages have the following characteristics and operation:

1. They boot first from page zero.

2. At some point in their execution they set the boot page select field (of the System Control Register) to select the next page needed.

3. With the next boot page selected, a software boot is forced by setting the boot force bit, loading that page into on-chip program memory. On-chip data memory is unchanged.

The memory-mapped control register System Control Register, located at internal data memory address 3FFF Hex, contains the BPAGE select field and BFORCE bit. This register must be written to under program control when re-booting is necessary.

E.3 SHARED DATA STRUCTURES

In order for data buffers to be shared by different boot pages, precautions must be taken to prevent the overwriting of the data when a new page is

booted. The STATIC qualifier, when used in a .VAR buffer declaration, instructs the Linker to prevent the buffer from being overwritten during a software re-boot. The Linker accomplishes this by different means for PM data buffers and for DM data buffers.

E.3.1 Data Buffers in Program Memory

Data buffers located in program memory are either relocatable or non-relocatable, depending on whether or not they are declared at a specific base address (with the ABS qualifier). If you wish to share a buffer between multiple boot pages, you may take one of two approaches. The first is to make the buffer relocatable by omitting the ABS specification and declaring the buffer as STATIC, allowing the Linker to perform the task for you. Alternatively, you may place the buffer yourself with the ABS qualifier; however, you must ensure that the buffer is never overwritten in on-chip PM during a re-boot.

To use the STATIC qualifier to create a data buffer to be shared, declare the buffer in the following manner in source code on boot page 0:

.VAR /PM/STATIC/ etc. *buffer_name*[length];
.GLOBAL *buffer_name*

If you were to declare a buffer called *dat1*, for example, boot page 0 would be as shown in Figure E.1. Notice that the Linker places *dat1* at the top of the 2K-long page. If the buffer contents are not initialized, the page length to be loaded extends only as far as the source code on the page.

Suppose that *dat1* is to be shared by boot pages 0, 1, and 2. It must be declared as above on page 0, and given the GLOBAL attribute so that it may referenced in other code modules on all three pages. These modules which require access to the buffer must use the .EXTERNAL directive in order to be able to reference it:

.EXTERNAL *dat1*;

If *dat1* is initialized on page 0, the memory images of pages 0, 1, and 2 would be as shown in Figure E.2, on page E-4. Because initialization data is contained in the buffer on page 0, the page length to be loaded is the full 2K. The Linker places filler bytes (FF Hex) in the region between the end of code and the start of the buffer.

Now suppose that execution of the source code on page 0 modifies the data in the buffer *dat1*; the new data must be passed to page 1 unaltered.

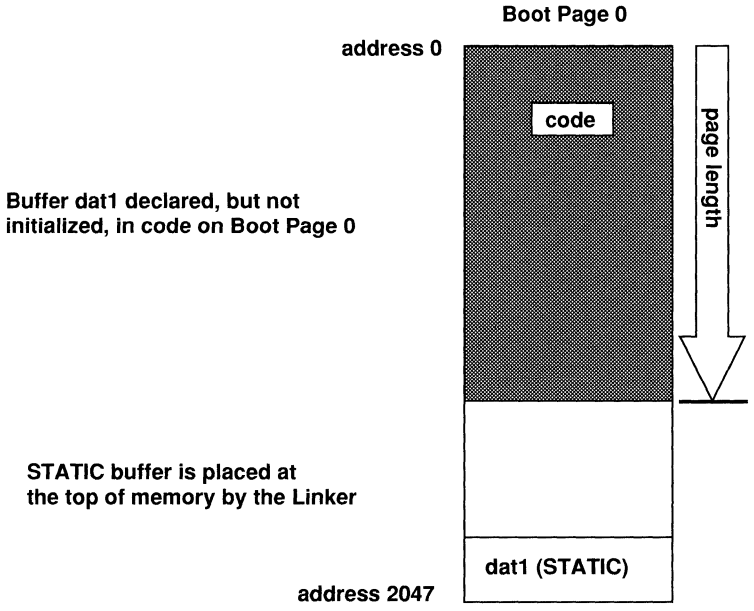


Figure E.1 STATIC Data Buffers in Boot Memory

When page 1 is booted, the buffer is *not overwritten* in on-chip PM because loading occurs *only up to the page length*. Booting of page 2 also leaves the buffer's contents undisturbed, as can be seen from its page length.

The Linker always places STATIC buffers at the top of memory on a boot page. If the code on the page is too long and overlaps the buffer space, a Linker error message is generated.

The alternative to using the STATIC qualifier to create a shared data buffer is to place the buffer in boot memory yourself with the ABS buffer address specification. The address must be chosen such that it is higher in memory than the largest boot page (page length). In this case you are doing the job of the Linker– placing the data buffer in memory and assuring that it is never overwritten during a boot page load.

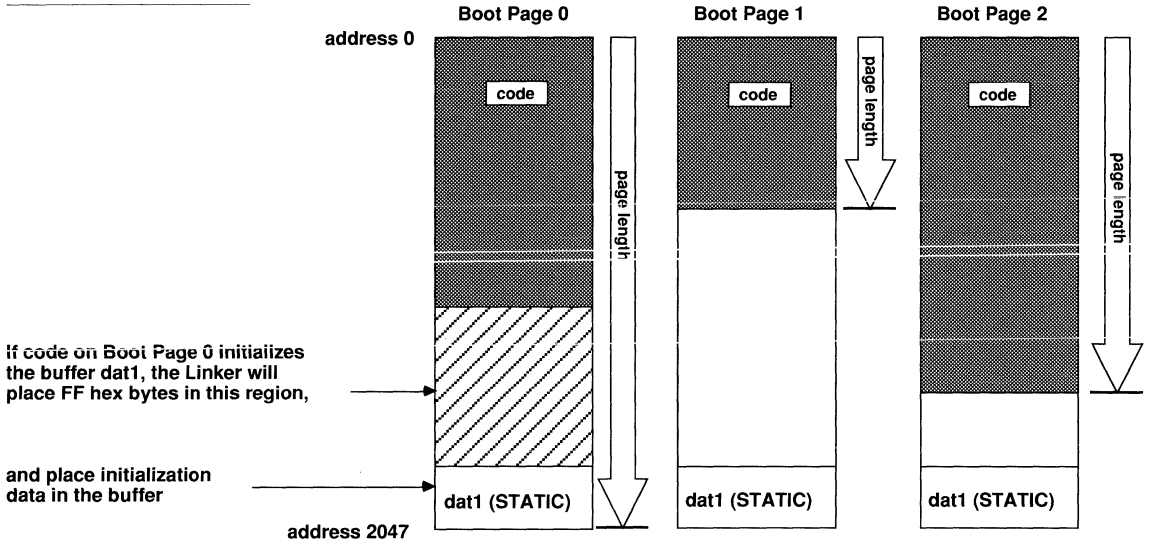


Figure E.2 Sharing STATIC Data Between Multiple Boot Pages

E.3.2 Data Buffers in Data Memory

In order to share a data buffer in data memory between multiple boot pages, the buffer must be relocatable, GLOBAL, and STATIC. To assure that the buffer data is preserved during software re-boots, the Linker follows the same procedure for the boot pages as it does when linking multiple code modules. See the Memory Allocation section in Chapter 4. In this case the Linker must scan over all boot pages and place any non-relocatable buffers before placing the STATIC buffer at a location where it will not be overwritten by any other page's data structure.

Again you may do the job of the Linker by omitting the STATIC qualifier and placing the data buffers of all boot pages in data memory. This task requires an exact determination of the complete layout of data memory which may prove difficult; allowing the Linker to do the work for you is usually easier.

E.4 SHARED SUBROUTINES

Any source code to be shared between multiple boot pages must actually be located on each of the pages. There are two ways to accomplish this: by repeating the BOOT=n qualifer in the module declaration, or by creating libraries and using the Linker's –p switch.

E.4.1 Repeating The BOOT Qualifier

If you write subroutine code modules, a copy of the module must be placed on each boot page which calls it. One way to accomplish this is by repeating the BOOT=n qualifier for each page necessary in the subroutine module declaration, as in:

.MODULE/RAM/BOOT=0/BOOT=1/BOOT=2 *calc_routine*;
.ENTRY *start_calc*;

In this case, the Linker now places a copy of the module *calc_routine* on boot pages 0, 1, and 2. Remember that the address label for the start of the subroutine code (*start_calc* in this example) must be declared as an ENTRY point in order to be visible to other code modules on the same page. These modules which call the subroutine must declare the entry point as EXTERNAL.

E.4.2 Libraries & –p Switch

If you have a large number of subroutines written for your ADSP-2101 system, it may be desirable to place the files in a directory of library routines. See the discussion of the the Linker –lib *directory* switch & ADIL variable in Chapter 4. Library routines are also used by the C Compiler when assembling C-generated source code.

When linking your complete system program, use the –p switch. The Linker inserts a copy of any subroutine called on a boot page into the memory image file for that page. Figure E.3, on the next page, illustrates the results of this process. To find the library of subroutines, the Linker searches directories specified by the –lib switch and ADIL variable.

E Linker Operation

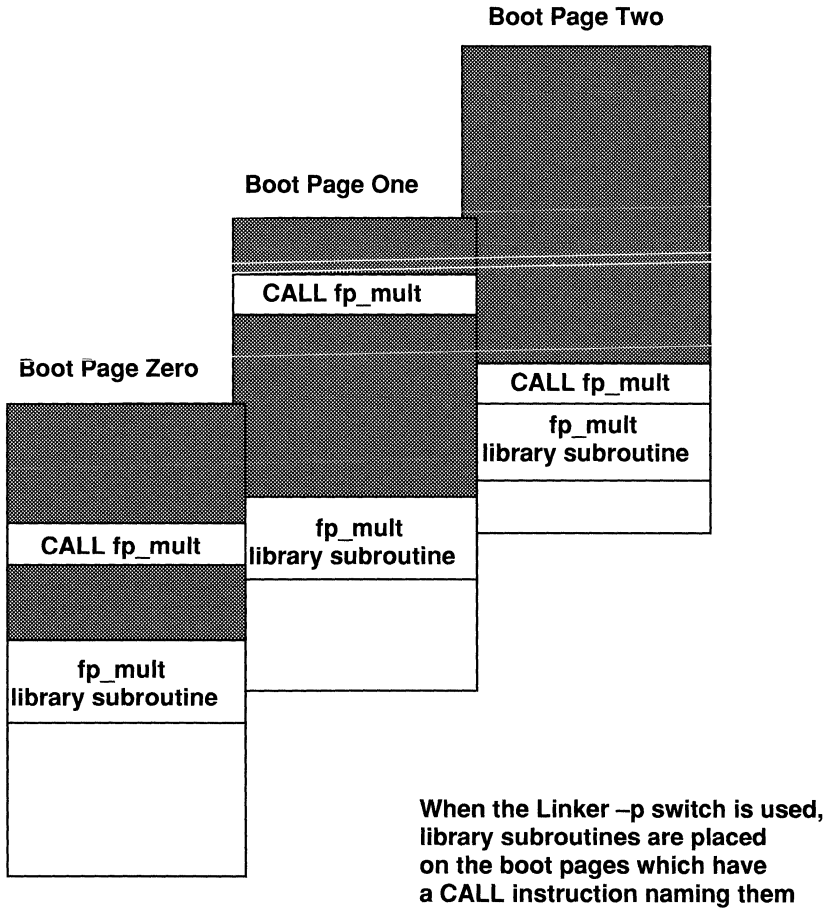


Figure E.3 Library Routines & Multiple Boot Pages

Error Messages F

F.1 INTRODUCTION

This appendix lists and provides a definition of all error messages generated by the ADSP-2101 Cross-Software modules other than the C Compiler. A listing is included for the System Builder, Assembler, Linker, and Simulator. The PROM Splitter displays no error messages. See Chapter 7 for a description of C Compiler error messages.

F.2 SYSTEM BUILDER ERRORS

The System Builder generates messages for syntax errors and system architecture definition errors. Syntax errors are errors in usage of the System Builder directives in the input file. Architecture definition errors are primarily errors in memory configuration, and may be either fatal or non-fatal. Error sources should be corrected in the input file.

Error message	*Explanation*
Boot memory segment crosses 16K boundary	Memory boundary limit exceeded
BOOT page must be on a 2K word boundary	Boot page segments do not accept the ABS modifier; the first address of a boot page is always equal to the boot page number * 2048
BOOT page cannot exceed 2K word page size	Boot page segment declared with length greater than 2048
Code/data– 24 required bits in memory width	Info only: 24-bit word width in PM
Data memory segment exceeds 16K boundary	Memory boundary limit exceeded

F Error Messages

Data memory segment overlaps reserved space	No segment may be declared in upper 1K of data memory
Divide by zero in expression	Arithmetic error
DM segment can have DATA attribute only	Segment declared in data memory may not have the CODE qualifier
DM segment cannot exceed physical address h#3fff	Segment declared in data memory address beyond 0x3FFF
Expecting a constant at symbol *symbol*	Numeric constant required in place of the named symbol
Expecting 'ABS' at symbol *symbol*	System-reserved keyword 'ABS' required in place of the named symbol
Expecting an identifier at symbol *symbol*	User-defined identifier required in place of the named symbol
Expecting 'CONST', 'ADSP2100', 'ADSP2101', 'PORT', 'SEG', 'ENDSYS', or 'MMAP' at symbol *symbol*	System-reserved keyword required in place of the named symbol
Expecting segment modifier at symbol *symbol*	Segment qualifier required in place of the named symbol
Expecting 'SYSTEM' at symbol *symbol*	'.SYSTEM' must be the first text read by the System Builder
Expecting '.' at symbol *symbol*	'.' required in place of the named symbol
Expecting ']' at symbol *symbol*	']' required in place of the named symbol
Expecting '=' at symbol *symbol*	'=' required in place of the named symbol
Expecting '/' at symbol *symbol*	'/' required in place of the named symbol

Error Messages F

Expecting ';' at symbol *symbol*	';' required in place of the named symbol
FATAL ERROR: Boundary error occurred	Memory boundary limit exceeded
FATAL ERROR: Impossible PM configuration	Program memory configuration not allowed
FATAL ERROR: Overlap occurred	Two or more declared segments overlap
FATAL ERROR: Unable to open *filename* for reading	System Builder cannot find .SYS file or cannot create .ACH; 1) the path/filename specified is incorrect or not allowed, 2) file does not exist, or 3) operating system condition
NOTICE: ? no PM found	No program memory segment was declared
Ports are not allowed in boot memory	I/O ports may only be placed in data or program memory
Problem mallocing enough memory	Not enough memory is available on host computer for System Builder to continue running
Program memory segment crosses 16K boundary	Program memory may not exceed 16K
Trying to redeclare symbol *symbol*	The named symbol is declared twice in the input file
Warning: Absolute address specified for boot page will be ignored.Segment address will be boot page * 2K	Boot page segments do not accept the ABS modifier; the first address of a boot page is always equal to the boot page number * 2048
Warning: Missing semicolon after ENDSYS directive	Semicolon must always terminate an instruction or directive

F Error Messages

You must specify memory segment address	ABS qualifier was omitted in a segment declaration
You must specify memory segment area	PM, DM, or BOOT qualifier was omitted in a segment declaration
You must specify memory segment type	ROM or RAM qualifier was omitted in a segment declaration

F.3 ASSEMBLER ERRORS

Error message	*Explanation*
Boot page out of range	Page number specified is invalid
Divide by zero in expression	Arithmetic error
Do labels cannot be external	DO loop cannot reference an external label
Do loop terminates do loop at symbol *symbol*	A DO instruction may not be the last instruction of a DO loop
Dreg of data memory access must be one of: AX0, AX1, MX0, MX1	Incorrect data register used in instruction
Dreg of program memory access must be one of: AY0, AY1, MY0, MY1	Incorrect data register used in instruction
Expecting a condition at symbol *symbol*	Instruction condition code required in place of the named symbol
Expecting a constant at symbol *symbol*	Numeric constant required in place of the named symbol
Expecting a data format at symbol *symbol*	'UU', 'SU', 'US', or 'SS' required in place of the named symbol (in an instruction)

Expecting an identifier at symbol *symbol*	User-defined identifier required in place of the named symbol
Expecting an index register at symbol *symbol*	Index register required in place of the named symbol (in an instruction)
Expecting an integer at symbol *symbol*	Immediate integer operand required in place of the named symbol (in an instruction)
Expecting AR as the destination of the alu operation	AR required in instruction
Expecting 'AV', 'AC', 'MV' or 'CE' at symbol *symbol*	Instruction condition code required in place of the named symbol
Expecting 'AX0', 'AX1', 'MX0', or 'MX1' for DM read	'AX0', 'AX1', 'MX0', or 'MX1' required in instruction
Expecting 'AY0', 'AY1', 'MY0', or 'MY1' for PM read	'AY0', 'AY1', 'MY0', or 'MY1' required in instruction
Expecting binary operator at symbol *symbol*	Logical (not bitwise) expression operator required in place of the named symbol
Expecting 'C' at symbol *symbol*	'C' (carry bit) required in place of the named symbol (in an instruction)
Expecting 'DM' at symbol *symbol*	'DM' required in place of the named symbol (in an instruction); immediate addresses must be in data memory
Expecting 'ENA' or 'DIS' at symbol *symbol*	'ENA' or 'DIS' required in place of the named symbol (in an instruction)
Expecting 'EXP' at symbol *symbol*	'EXP' required in place of the named symbol (in an instruction)

F Error Messages

Expecting 'EXPADJ' at symbol *symbol*	'EXPADJ' required in place of the named symbol (in an instruction)
Expecting 'HI', 'LO', or 'HIX' at symbol *symbol*	'HI', 'LO', or 'HIX' required in place of the named symbol (in an instruction)
Expecting 'HI' or 'LO' at symbol *symbol*	'HI' or 'LO' required in place of the named symbol (in an instruction)
Expecting 'I0', 'I1', 'I2', or 'I3' for DM index register	'I0', 'I1', 'I2', or 'I3' required in instruction
Expecting 'I0', 'I1', 'I2', or 'I3' at symbol *symbol*	'I0', 'I1', 'I2', or 'I3' required in place of the named symbol (in an instruction)
Expecting 'I4', 'I5', 'I6', or 'I7' at symbol *symbol*	'I4', 'I5', 'I6', or 'I7' required in place of the named symbol (in an instruction)
Expecting 'M0', 'M1', 'M2', or 'M3' at symbol *symbol*	M0–M3 must be used if I0–I3 are used (in an instruction)
Expecting 'M4', 'M5', 'M6', or 'M7' at symbol *symbol*	'M4', 'M5', 'M6', or 'M7' required in place of the named symbol (in an instruction)
Expecting MODULE directive at symbol *symbol*	The MODULE directive must be the first statement in any file which the Assembler reads
Expecting MODULE qualifier at symbol *symbol*	MODULE qualifier required in place of the named symbol
Expecting MR as the destination of the mac operation	MR required in instruction
Expecting 'MR' at symbol *symbol*	'MR' required in place of the named symbol (in an instruction)
Expecting 'OR' at symbol *symbol*	'OR' required in place of the named symbol (in an instruction)

Expecting 'PM' at symbol *symbol*	'PM' required in place of the named symbol (in an instruction)
Expecting 'PUSH' or 'POP' at symbol *symbol*	'PUSH' or 'POP' required in place of the named symbol (in an instruction)
Expecting 'RND' at symbol *symbol*	'RND' required in place of the named symbol (in an instruction)
Expecting segment name at *symbol*	Segment name required in place of the named symbol (in an instruction)
Expecting shift operand at symbol *symbol*	Shift operand required in place of the named symbol (in an instruction)
Expecting 'STS' at symbol *symbol*	'STS' required in place of the named symbol (in an instruction)
Expecting VAR qualifier at symbol *symbol*	VAR qualifier required in place of the named symbol
Expecting '1' at symbol *symbol*	'1' required in place of the named symbol (in an instruction)
Expecting '0' at symbol *symbol*	'0' required in place of the named symbol (in an instruction)
Expecting '0' or '1' at symbol *symbol*	'0' or '1' required in place of the named symbol (in an instruction)
Expecting ']' at symbol *symbol*	']' required in place of the named symbol
Expecting '–' at symbol *symbol*	'–' (subtract) required in place of the named symbol (in an instruction)
Expecting '*' at symbol *symbol*	'*' (multiply) required in place of the named symbol (in an instruction)

F Error Messages

Expecting '(' at symbol *symbol*	'(' required in place of the named symbol (in an instruction)
Expecting ',' at symbol *symbol*	',' required in place of the named symbol (in an instruction)
Expecting '=' at symbol *symbol*	'=' required in place of the named symbol (in an instruction)
Expecting ')' at symbol *symbol*	')' required in place of the named symbol (in an instruction)
Expecting ':' at symbol *symbol*	':' required in place of the named symbol (in an instruction)
Expecting ';' at symbol *symbol*	';' required in place of the named symbol (in an instruction)
FATAL ERROR: unable to open *filename* for reading	Assembler cannot find the named input file; 1) the path/filename specified is incorrect, or 2) file does not exist
FATAL ERROR: unable to open *filename* for writing	Assembler cannot create the named file due to an operating system condition, such as a bad filename or full disk
Illegal address operand *symbol*	Immediate operand required in place of the named symbol (in an instruction)
Illegal address specified	Address is in un-declared memory or is invalid
Illegal data specified	Data is invalid
Illegal do until term at symbol *symbol*	DO UNTIL instruction has an illegal termination condition
Illegal do until not term at symbol *symbol*	DO UNTIL NOT instruction has an illegal termination condition

Error Messages F

Illegal exponent *symbol*	Exponent is out of range (in an immediate shift instruction)
Illegal INIT value *symbol*	Buffer initialization value is invalid
Illegal length specified	Buffer length is invalid
Illegal length operator usage	'%' (length of) operator incorrectly used
Illegal lhs *'item'*	Item named is invalid when located to the left of equal sign (in an instruction)
Illegal mode operand *symbol*	Mode operand required in place of the named symbol (in an instruction)
Illegal multi-function	Incorrect instruction syntax used (illegal operands included)
Illegal offset specified	Buffer address offset is invalid
Illegal operand *symbol*	Instruction operand required in place of the named symbol
Illegal rhs *'item'*	Item named is invalid when located to the right of equal sign (in an instruction)
Illegal stack operand *symbol*	Stack operand required in place of the named symbol (in an instruction)
Illegal yop *symbol*	Instruction yop required in place of the named symbol
Illegal xop *symbol*	Instruction xop required in place of the named symbol
Multiple do's to the same address *symbol*	Illegal do loop

F Error Messages

Must use DAG0 for data memory access	Incorrect data address generator used in instruction
Problem mallocing enough memory	Memory allocation limit or restriction reached
Result register contention	Two or more operations executed in the same cycle use the same result register
Trying to redeclare *symbol* as a buffer	Named symbol is declared twice
Trying to redeclare *symbol* as an external	Named symbol is declared twice
Trying to redeclare *symbol* as a label	Named symbol is declared twice
Trying to redeclare *symbol* as a port	Named symbol is declared twice
Undefined do addr: *address*	Address specified in non-existent memory
Undefined symbol: *symbol*	Named symbol is not declared properly
Warning: Illegal register access inferred	Incorrect register usage

F.4 LINKER ERRORS

The Linker generates error messages, warning messages, and informational messages. Some errors allow the Linker to continue running in order to detect additional problems, but fatal errors halt the linking process. Both error types are reported to the display, as well as a limited number of warning and informational messages. Error sources must be corrected.

Several categories of Linker errors are detected. The most common messages point out memory allocation and symbol reference problems. These errors can result from insufficient memory declarations, improper

absolute address specifications, undefined symbol references, etc. Other messages result from operating system conditions or software failures during execution of either the Assembler or Linker.

Words shown below in italics are error message arguments, and specify the particular item involved with an error condition.

F.4.1 Operating System Errors

Error Message	*Explanation*
Assembler detected errors in module *modulename*	Assembly errors are flagged in the specified module; source code must be corrected and re-assembled
Can't create executable file *filename*	Linker cannot create .EXE file due to an operating system condition, such as a bad filename or full disk
Can't create list file *filename*	Linker cannot create .MAP file due to an operating system condition, such as a bad filename or full disk
Can't create symbol file *filename*	Linker cannot create .SYM file due to an operating system condition, such as a bad filename or full disk
Can't create temporary name	Linker is unable to complete the directory search: when searching a directory for files to link, the Linker must create a temporary file containing a list of the directory's contents; this error is seen when insufficient memory is available for the file on the host computer or if the list is not found

F Error Messages

Can't open architecture file *filename*	Linker can't find .ACH file; 1) default filename 210x not found, and no alternative specified with the –a switch, 2) the path/filename specified with –a switch is incorrect, or 3) file does not exist
Can't open buffer init file *filename*	Linker cannot find buffer initialization file; 1) the path/filename specified (in .INIT Assembler directive) is incorrect, or 2) file does not exist
Can't open code file *filename*	Linker cannot find the named .OBJ file to link; 1) the path/filename specified is incorrect, or 2) file does not exist
Can't open input list file *filename*	Text file named in –i switch is not found; 1) the path/filename specified is incorrect, or 2) file does not exist
Can't open library file *filename*	Linker cannot find the named .OBJ file to link; 1) the search path/directories specified (with ADIL environment variable or –lib switch) are incorrect, or 2) file does not exist
Can't open temp file *filename*	Linker cannot find temporary file it created during directory search
Errno: # Can't execute MSDOS command: *command*	Linker has given a DOS command which is not correctly executed; DOS error number listed is returned to Linker
Fail on fseek: *filename*	Linker is unable to complete a file or directory search
Failed request to alloc more memory	Not enough memory is available on host computer for Linker to continue running

F.4.2 Informational Messages

Error message	*Explanation*
Library (ies) searched: *directory, ...*	Linker is searching the listed directories for files to link
Trying to open library d*irectory*	Linker is looking for the named directory
Using library *director*	Linker is processing files from the named directory

F.4.3 Memory Allocation Errors

Error message	*Explanation*
(Warning) Bootable module *modulename* has been located at external address *address*	The module named is declared on a page of boot memory; however, not enough memory is available on the page to store this module, and the Linker has been forced to place it in off-chip memory (i.e. too much code and/or data has been declared on the boot page) (WARNING ONLY)
Can't place *symbol* of module *modulename*	The data buffer or code module named by *symbol* cannot be placed in memory (specific reasons to follow)
/ at address *address*	Object is declared at absolute address specified (with ABS qualifier), but cannot be placed there
/ contention at *address*	Two objects have been declared at the same absolute address (with ABS qualifier), or overlap each other at the specified address

F Error Messages

/ for boot page *page#*	Object is to be placed in boot memory on page number listed
/ no appropriate *pm,dm* ram available	Object to be placed is declared in PM RAM or DM RAM; this memory type does not exist in architecture
/ no appropriate *pm,dm* rom available	Object to be placed is declared in PM ROM or DM ROM; this memory type does not exist in architecture
/ not enough *pm,dm rom,ram* left	Not enough of the specified memory type is available to complete placement of object
/ (Warning) placement forced to be external	Object declared in internal DM or PM; Linker placing object in external DM or PM, however, due to shortage of on-chip memory space (WARNING ONLY)
/ requires # words of *pm,dm rom,ra*	Relocatable object has length (in words) shown, and is declared in memory type specified (INFORMATION ONLY)
/ requires # words of *pm,dm rom,ra* from a segment named *segname*	Object is relocatable within named segment, has length (in words) shown, and is declared in memory type specified (INFORMATION ONLY)
/ *symbol* of module *modulename,*	*Symbol* specifies the data buffer or code module being placed (INFORMATION ONLY)

F.4.4 Symbol Reference Errors

Error message	*Explanation*
Global *buffername* declared in modules *modulename* and *modulename* (maybe others)	A global buffer should only be declared in one module to prevent conflicting usage
Module name *modulename* duplicated	Two or more modules to be linked have the same name; a unique name must be given to each
Symbol of module *modulename* not linked	The data buffer or address label listed is declared as .EXTERNAL in this module, but is not declared as .GLOBAL or .ENTRY in another module
Symbol of *modulename* is bootable but ref'd as if not	The buffer or module named is stored in boot memory; a non-bootable program has referenced or called the bootable object, which may not have been booted yet
Symbol of *modulename* is not part of boot page *page#*	The buffer or subroutine (module) named is referenced on the boot page specified, but a copy is not located on the page (see Appendix E)
Symbol of *modulename* ref'd in *modulename* is not part of boot page *page#*	The global buffer or address label named by *symbol* is referenced (in the second module named) on the boot page specified, but a copy is not located on the page (see Appendix E)
Usage of *symbol* in module *modulename* implies it is in *pm,dm*, it is not	The object named is declared in PM or DM; it is referenced in the code module named as if to be found in the other memory space

F Error Messages

Usage of *symbol* in module *modulename* is inconsistant with declaration	The buffer name or address label shown is misused in code; for example, using a variable name as an entry point

F.4.5 Other Errors

Error message	*Explanation*
Boot image created (# kbytes) is larger than specified boot memory (# kbytes	The boot memory portion of .EXE file contains more bytes than the boot memory space (defined in .ACH file) can hold
(Warning) Initialization data for *buffername* in module *modulename* is longer than buffer	Initialization file contains more data than can be loaded into buffer (WARNING ONLY)
Link errors	Generic message displayed if any errors are detected
No boot memory for generated boota images found in *sysname* (*filename*)	Linker has processed modules which have the BOOT qualifier; however, no boot memory is declared in .ACH file
Offset on *buffername* in module *modulename* forces address out of range	An access of *buffername* is attempted with an offset added to the buffer base address; the offset is too large, and address(es) to be accessed does not exist
Sources too large *filename*	The named .OBJ file is too large for Linker to handle; the assembly source code file must be divided into smaller size files and re-assembled Note: This error will rarely be seen for the ADSP-2101 Linker
filename too big– breakup sources	Same as above

F.4.6 Software Errors

These errors indicate that either the Linker or Assembler is not operating properly. When a software failure is detected, the Linker aborts execution and specifies one of three error conditions. The initial message is always seen first.

If you see these messages, contact the Applications Engineering Group at Analog Devices, Digital Signal Processing Division. See the copyright page of this manual for telephone numbers to use.

Error message	*Explanation*
Calling broken software	Initial message; software failure detected in Linker or Assembler
Can't find *symbol* of module *modulename* in symbol table (it should be there!)	Linker has allocated memory for *symbol* (buffer, module, or address label) without error,and added *symbol* to the symbol table; when the Linker subsequently tries to assign an address, however, the symbol is not present in the symbol table (Linker failure)
New module search fail	Same as above error condition, but is generated only for module names not found in symbol table (Linker failure)
Opcode with bad unresolved field	Opcode generation mechanism of the Assembler is not operating correctly (Assembler failure)

F Error Messages

F.5 SIMULATOR ERRORS

F.5.1 General Errors

Error message	*Explanation*
Bad filename for redirect	File not found for command window input in redirect command: *<'filename'*
Cannot open Help files	ADIDOC environment variable not set correctly or help files not present
Expecting an integer	Non-integer value given
Files must be single quoted	Any file named in a command must be in single quotes
Invalid command	Incorrect syntax used in a command window command
Unable to open any more windows	Maximum of 10 windows open at one time

F.5.2 Defaults Errors

Error message	*Explanation*
Can't add any more paths	The total number/size of search paths allowed is limited (list shown in defaults window)
Can't change architecture information	Information is for display only
Can't replace current directory	Current directory is always searched
Unable to replace path	Search path to be added is too large

F.5.3 Expression Errors

Error message	*Explanation*
Divide by zero in expression	Arithmetic error
Expression table full	Maximum of 10 expressions allowed
Invalid expression	Invalid operator or operand used in expression command: *?* or *?+*
Unable to delete expression	Expression number given does not exist

F.5.4 Break Errors

Error message	*Explanation*
Break in PM only	Breakpoints and break ranges may not be set in boot or data memory
Invalid address	Address given in set breakpoint command is in undeclared memory segment: *B addr*
Invalid break address	No break is set at address (label) given or break number given does not exist in break delete command: *BD addr or number*
Invalid break expression number	Break expression number given in break delete command does not exist
Invalid break specification	Break number given is not an integer or address (label) given is undefined in break delete command: *BD addr or number*
Invalid range	Range given in set break range command is in undeclared memory segment: *BR range*

Only integer shifts are allowed in break expressions	Non-integer used in bitwise shift

F.5.5 Watch Errors

Error message	*Explanation*
Unable to delete watchpoint	No watchpoint or watch expression exists for number given in watch delete command: *WD number*
Watch expr table full	Maximum of 25 watch expressions allowed
Watchpoint table full	Maximum of 25 watchpoints allowed

F.5.6 Command Errors

Error message	*Explanation*
Invalid alias to delete	Symbol given in delete alias command does not exist in alias list: *JD symbol*
Invalid interrupt number	Interrupt number must be 0, 1, or 2 in interrupt command: *I int# min max*
Invalid range specification	Range given in dump memory command is in undeclared memory segment: *D addr or range*
Not a valid register	Register named in load register command must be a processor or Simulator-defined register: *R reg expr*

F.5.7 Plot Memory Errors

Error message	*Explanation*
Invalid ending DM address	Address given in plot memory command is in undeclared memory segment
Invalid ending PM address	Address given in plot memory command is in undeclared memory segment
Invalid increment value	Decimation value must be a positive integer in plot memory command: *PL range decimation*
Invalid plot size (max 640)	Range length/decimation must be less than or equal to 640 for plot memory command
Invalid starting DM address	Address given in plot memory command is in undeclared memory segment
Invalid starting PM address	Address given in plot memory command is in undeclared memory segment
PLT_FNC_WRN: undefined data contained in plot range	Location(s) in memory range to be plotted not loaded with data; warning only– Simulator will attempt to create plot

F.5.8 Port & SPORT Errors

Error message	*Explanation*
Encountered end of SPORT 0 input file	Execution halted when no further data available; warning only
Encountered end of SPORT 1 input file	Execution halted when no further data available; warning only

F Error Messages

Mult error: frame sync came too early	RFSDIV value too low in multichannel mode
Mult Receive enable register undefined	Both words of Multichannel Word Enable processor control register must be loaded
Mult Transmit enable register undefined	Both words of Multichannel Word Enable processor control register must be loaded
No input file for SPORT 0	No serial data received during program execution; data input file must be assigned in serial port command: *P SPORT# <'file'*
No input file for SPORT 1	No serial data received during program execution; data input file must be assigned in serial port command: *P SPORT# <'file'*
No output file for SPORT 0	Unable to transmit serial data during execution; data output file must be assigned in serial port command: *P SPORT# >file'*
No output file for SPORT 1	Unable to transmit serial data during execution; data output file must be assigned in serial port command: *P SPORT# >file'*
Not a valid serial port	Number given in open serial port command must be 0 or 1: *P SPORT#*
Ports allowed only in DM or PM	I/O ports may not be located in boot memory
Problem writing SPORT 0 output file	Execution halted; operating system unable to write serial transmit data to file

Problem writing SPORT 1 output file	Execution halted; operating system unable to write serial transmit data to file
Unable to open any more I/O ports	Maximum of 25 I/O ports allowed

F.5.9 Instruction & Program Load Errors

Error message	*Explanation*
Bad instruction	Not valid ADSP-2101 instruction
Can't find instr in DM	Instructions may only be searched for in program and boot memory with the find command: *F range expr*
Unable to assemble into DM	Instructions may only be loaded into program or boot memory with the assemble command: *A addr instr*
Writing RAM segment to ROM in DM	Warning only; pertains to load program or load rom command: *L 'file', LR 'file'*
Writing RAM segment to ROM in PM	Warning only; pertains to load program or load rom command: *L 'file', LR 'file'*
Writing ROM segment to RAM in DM	Warning only; pertains to load program or load rom command: *L 'file', LR 'file'*
Writing ROM segment to RAM in PM	Warning only; pertains to load program or load rom command: *L 'file', LR 'file'*

F Error Messages

F.5.10 Execution Errors

Error message	*Explanation*
Can't set SSTAT register	Stack status register is read-only
Error: M reg is greater than L reg	Modify value must be less than length of circular buffer
PC stack empty when POP occurred	No return address on PC stack when RTI or RTS instruction executed
SE undefined used in shift operation	Shifter exponent register is not loaded for shift instruction
Tried to read from non-existent memory	Source of read instruction executed is in undeclared memory segment
Tried to read from reserved memory	Only reads from processor control registers are allowed in upper 1K of data memory
Tried to write to non-existent memory	Destination of write instruction executed is in undeclared memory segment
Tried to write to reserved memory	Only writes to processor control registers are allowed in upper 1K of data memory
Tried to write to ROM	Destination of write instruction executed is in ROM
Undefined instruction executed	Program memory location accessed is not loaded
Undefined registers used in shifter	Data register is not loaded for shift instruction
Undefined used in ALU operation	Input register is not loaded for add or subtract instruction

Undefined used in DAG	Either L, I, or M register is not loaded prior to memory access
Undefined used in MAC operation	Input register is not loaded for multiply instruction
Undefined used in timer	All three timer registers must be set: TCOUNT, TPERIOD, TSCALE

F.5.11 Command Syntax Errors

Error message	*Explanation*
Usage: A address instruction Usage: I min max Usage: J symbol command Usage: K win line Usage: O addr [<file] [>file] Usage: P # [<file] [>file] Usage: PA line # start end Usage: PC Usage: PD line # Usage: PL range increment Usage: PP parameter value Usage: PR Usage: U (address | range | register) Usage: X symbol Usage: Y [<file] [>file] Usage: Z [<file] [>file]	Command entered incorrectly

Analog Devices
Digital Signal Processing Division
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
(617) 329-4700

E1344b-0.7-9/90