DSP32C PC/AT USER GUIDE

Release 9/1/92

TABLE OF CONTENTS

INTRODUCTION

The AT&T DSP32C chip is a powerful numerical engine featuring the latest advances in submicon CMOS chip technology. With its 25 Mflop single cycle multiply/accumulate, and 32 bit data and 24 bit address buses, it brings performance rivaling many super computers to the PC environment. We hope you enjoy using your system and find your software and hardware development to be fast and easy.

Symmetric Research offers three boards featuring the DSP32C, allowing you to select the best combination of price, total on board memory, and number of cpus for your applications. Furthermore, because the hardware interface to the PC is essentially the same for each board, most software runs on all systems with no changes. Throughout this manual, and the supplied software, the three boards are referred to as:

Board	Number of cpus	On board memory	Parallel port	Serial ports
DSP_400	1	3 Mbyte	1 32 bit 12.5 Mhz	1 DSP32C
DSP_MOD	1	8 Mbyte	1 32 bit 12.5 Mhz	1 DSP32C
DSP_MUL	up to 4	1 Mbyte per cpu	none	1 DSP32C per cpu

where the DSP_400 board uses 400 mil wide 128Kx8 SRAM memory chips, and the DSP_MOD and DSP_MUL boards use 256Kx32 SRAM memory modules. Each board is equipped with a 16 bit PC/AT interface, and can have all of its on board memory transparently accessed by the PC while the DSP32C is executing.

This manual describes the hardware and software tools for the SR DSP32C boards. With the supplied software you have everything you need to develop either stand alone or integrated PC/AT DSP32C applications. Included are an assembler, monitor debugger, C callable math library, many graded examples to help you get started, and all the source code for the entire system. Using these tools you should be able to modify and develop the system to meet your needs.

INSTALLATION

HARDWARE

Hardware installation of the DSP32C coprocessor board is simple. Basically, find an available 16 bit slot in your PC and plug it in. The exact steps we recommend are as follows:

- Turn your machine off.
- Remove the lid and find an available slot.
- Avoid static by touching the DSP32C conductive bag and the PC at the same time. This makes sure you, the board, and the PC are all at the same potential.
- Pull the board from the bag and observe the switches and jumpers on it. Their default settings are fine for now.
- Plug the board in, put a screw in the bracket, and turn the power on. Your machine should boot as it normally does. Turn it off, and put the lid back on.

Once the board is installed, you will probably want to immediately run some software to verify its proper operation. To do this, first install the system software as described below, and then run some of the .exe programs in the \srdsp\examples directory. The simpler ones, like inner.exe, display text output, while others do full VGA graphics. Some of the more complicated programs may not work if there is not enough memory installed, whereas many of the simpler ones will run even with no on board memory.

The DSP32C coprocessor board requires very little address space in your PC, and it is unlikely that there will be any address conflicts with other hardware installed in your system. However, if there is, you can change the PC/IO base addresses the board and software uses. Changing the base addresses is covered in more detail under the CHANGING THE PC/IO BASE ADDRESS subtitle below. Additional information about the DIP switches and jumpers can also be found in the CIRCUITS chapter, while more information about the PC/IO mapping can be found in the MEMORY MAPPING chapter.

SOFTWARE

The system software is supplied on a single floppy disk in compressed PKZIP format. You should run the installation install.bat file on the supplied disk to install it. The installation batch file will automatically run the PKZIP decompression utility, and create the default \srdsp directory structure for the software on your hard disk. The directories that will be created on your disk are:

Directory\File	Description
\srdsp\bin\dspasm.exe	DSPASM assembler executable
\srdsp\bin\dspmon.exe	DSPMON monitor debugger executable
\srdsp\lib\cx_tools.lib	DSPTOOLS system utilities C library
\srdsp\lib\cx_math.lib	DSPMATH C/Fortran math library
\srdsp\lib\cx_graph.lib	GRAPHICS C library
\srdsp\include\dsptools.h	DSPTOOLS C include file
\srdsp\include\dspmath.h	DSPMATH C include file
\srdsp\include\dspmath.for	DSPMATH FORTRAN include file
\srdsp\include\graphics.h	GRAPHICS C include file
\srdsp\dspasm\ ...	DSPASM source code
\srdsp\dsptools\ ...	DSPTOOLS source code
\srdsp\dspmath\ ...	DSPMATH source code
\srdsp\graphics\ ...	GRAPHICS source code

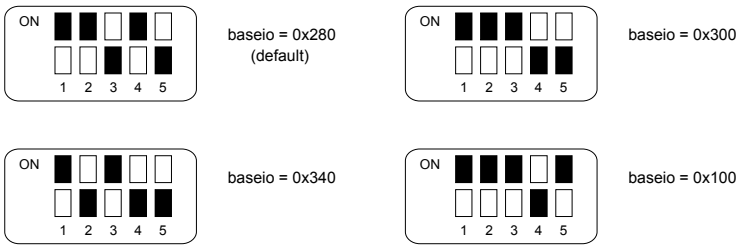
After installation, you should also set up your autoexec.bat file to include the \srdsp\bin directory on your DOS search path, and your C compiler to automatically search the \srdsp\include and \srdsp\lib directories for compiling and linking. Unless you have a compelling reason to do so, we recommend using the default \srdsp directory structure, and *avoid* creating multiple copies of the executables and libraries. Having only a single copy of the system software on your disk makes maintaining your software much easier.

Finally, under the \srdsp root directory, there are several batch files and readme distribution notes. The software as supplied with the board has been built with either Microsoft QuickC with Assembler v2.5 or Borland C++ v3, as indicated on the disk label. The batch files mkquick.bat, mkturbo.bat, and mkbrlnd.bat will rebuild the entire system from scratch with the respective tools.

CHANGING THE PC/IO BASE ADDRESS DIP SWITCHES

While there usually aren't address conflicts with the DSP32C coprocessor boards, it can happen. There have been some collisions reported with network cards, particularly Lantastic. If you suspect an address conflict is occurring, it is easy to change the DSP32C board addresses to see if the trouble goes away. Of course the best solution is know for certain what addresses are available, and select one without guessing.

To work at PC/IO base addresses other than the default, the DIP switches on the board *and* the system software must be changed. The software must know what address the board is at in order to properly communicate with it. Some common alternative settings for the DIP switches are covered in the following diagrams. All three of the SR DSP32C boards, DSP_400, DSP_MOD, and DSP_MUL, have a 5 position DIP switch that selects the base address for the DSP32C PIO registers on the board. Some possible settings are:



For the multiple DSP32C board, DSP_MUL, there is also an 8 position dip switch. This selects the base address of the cpu selection register. This register selects which DSP32C PIO is connected to the PC bus at the baseio base address. Some settings for this switch are:

Additional information about the DIP switches can be found in the circuit diagrams for your board in the CIRCUITS chapter.

Once the DIP switches have been set, you also need to set up your software to use the non default address. There are two ways to do this. The first is to set environment variables with the DOS set command:

set DSP32C_BASEIO=xxx
set DSP32C_BASESEL=xxx

where xxx are three hex digits as indicated on the dip switch diagrams above. Baseio corresponds to the address of the 5 position DIP switch, while basesel corresponds to the base address of the 8 position DIP switch on DSP_MUL boards. For example, to use a baseio address of 0x300, issue the DOS command set DSP32C_BASEIO=300. You will probably want to put these statements in your DOS autoexec.bat file. Spaces are allowed around the = sign, but there should not be any leading 0x in front of the three hex digits.

The software base addresses can also be set under program control. This method is especially useful when multiple boards are being run in one PC. By changing the baseio and basesel global variables in the DSPTOOLS library, your programs can select the board being

communicated with at runtime. These variables are covered in more detail in the DSPTOOLS chapter.

MEMORY MAPPING

This chapter covers the memory mapping of the coprocessor board from both the PC/AT's and the DSP32C's points of view. An accurate knowledge of the board's resources is necessary to program it, and both assembler and high level language programmers should be aware of these details.

PC/AT MEMORY MAPPING

The PC/AT bus communicates with the coprocessor boards through the PIO registers on the DSP32C. These registers are mapped into the I/O space of the PC/AT, and occupy 32 contiguous byte locations, where the base address for the PIO registers is selected with the 5 position DIP switch on the board. This switch specifies the top 5 bits of a 10 bit PC/IO space address. The normal default setting for baseio from the factory is 0x280, although you can use other addresses as covered in the INSTALLATION chapter. For the multiple cpu board, DSP_MUL, the basic PIO interface is augmented with an on board register that controls which cpu has its PIO connected to the PC bus. The DSP_MUL selection register will be described in more detail later in this section.

All the SR DSP32C coprocessor boards have been designed to use the 16 bit mode of the DSP32C PIO, and *all* transfers with the PIO use the full 16 bit PC bus. Word wide 16 bit IO transfers on the PC must be aligned on even PC/IO space addresses, and the boards have been designed to multiply the PIO register offsets listed on page 6-9 of the AT&T DSP32C Information manual by 2 to enforce this word alignment. Specifically, the PC/IO space offsets from baseio are:

OFFSET from PC I/O baseio address	PIO register
0x00	PAR
0x04	PDR
0x08	EMR
0x0C	ESR
0x0E	PCR
0x10	PIR
0x16	PARE
0x18	PDR

The most heavily used PIO registers are PCR, PAR, PDR, and PIR. PCR controls the DSP32C's execution, allowing for reset and other configuration options. The pair of registers PAR and PDR allow for reads and writes to the DSP32C memory space. First PAR is set with the address to read or write to, and then PDR is used as a data register to pump data in or out. The DSP32C automatically increments the PAR address for moving arrays on and off the board. The PIR register can be used for communicating a word to the PC, without using DMA cycles like the PDR register does. See the AT&T DSP32C Information manual for complete details on each of these registers.

C callable functions for communicating with these registers are provided in the DSPTOOLS library, and provide the easiest way to use the PIO. In particular, the get_32C and put_32C functions take care of transferring data through the PAR and PDR registers. The functions have been programmed in assembler for maximum performance, and use the 80286 insw and outsw instructions. See the DSPTOOLS and EXAMPLES chapters for how to use them. Users wanting to program the PIO registers directly, can use the DSPTOOLS source code as a guide.

For the DSP_MUL multiple cpu board, the basic interface to the DSP32C PIO has been augmented with a selection register to control which cpu has its PIO connected to the PC bus at baseio. This 8 bit register has its own 8 position DIP switch, so you can select its PC/IO address separately from the baseio address of the PIO. The factory default for the basesel address is 0x2A0, although you can run at other addresses as covered in the INSTALLATION chapter. Only the lowest two bits of the selection register are significant, and should be poked with the binary number of the selected cpu. Once the selection register has been poked, all other communication with the DSP_MUL board is just as it is for the DSP_400 and DSP_MOD boards. PC programs can poke the DSP_MUL selection register at any time without affecting the execution of any particular cpu.

Finally, it is possible to run multiple coprocessor boards in a single PC. To do so requires setting the DIP switches to different base addresses, so each board can have its own IO space locations. After setting the switches, the DSPTOOLS library has global variables, baseio and basesel that set the base addresses the library software will use. The values of these variables should be changed appropriately at run time to communicate with the desired board.

DSP32C MEMORY MAP

The DSP32C has a 24 bit 16 Mbyte address bus. Within this byte addressed space, the chip can communicate with the on chip DSP32C internal ram blocks, the external on board SRAMS, and the parallel port on the top edge of the DSP_400 and DSP_MOD boards. The DSP32C does not have any segmentation or separate program and address spaces. Any instruction can address any location in the byte addressed space. However, the DSP32C *does* have multiple distinct data buses connecting the on chip internal ram blocks and the external

SRAM memory, and generally performance can be improved by using these buses optimally. Please refer to the AT&T DSP32C Information manual for more details about the internal bussing.

The DSP32C can run in one of several memory modes. These modes control what addresses the internal ram blocks reside at, and can be changed depending on whether the system will be booting from ROMS or downloaded from the PC. The boards have all been jumpered at the factory to use memory mode 5. Other modes can be used. See the CIRCUITS chapter of the manual for jumper settings, and page 2-5 of the AT&T DSP32C Information manual for details about the memory modes. Memory mode 5 has been selected as the default because it places some internal ram at location 0. This special location is where execution begins after a DSP32C reset, and by mapping it to internal ram the PC is always guaranteed to have some valid memory for the DSP32C to execute in.

Assuming the default memory mode 5, the mapping of the DSP32C address space is:

Locations	**Physical mapping**
0x000000 to 0x0007FF	On chip memory block RAM2
0x000800 to 0x000FFF	On chip memory block RAM0
0x001000 to 0xEFFFFF	On board (off chip) SRAM memory
0xF00000	Parallel port 0 (R0,W0)
0xF40000	Parallel port 1 (R1,W1)
0xF80000	Parallel port 2 (R2,W2)
0xFC0000	Parallel port 3 (R3,W3)
0xFFF800 to 0xFFFFFF	On chip memory block RAM1

where the amount of on board SRAM memory is determined by how many chips or modules your board is populated with, and the parallel ports are present only for the DSP_400 and DSP_MOD boards.

The transition from internal on chip memory at 0x000FFF to on board SRAM at 0x001000 is handled transparently by the chip. It will automatically take care of changing over from using the internal data buses to using the external data bus without any programming concerns. However, as noted above, if you are programming for absolute maximum performance, you will want to use the internal and external buses optimally and avoid all pipeline hits. Consult the AT&T DSP32C Information manual for more details about the pipeline. Program performance can often be improved by a factor of 2 by judiciously placing your program and data so the processor can exercise all the buses in parallel.

Wait states for external cycles on the DSP32C are under software control, and do not require any jumpers to be set on the board. In fact, the wait states can be changed at run time as appropriate for the device being addressed. This is sometimes a factor for example, when a slow device is connected to the parallel port. Just before the port is addressed, the DSP32C can be slowed down with a few wait states, and then programmed back to fewer wait states after the slow access. The DSP32C PCW register has the bits controlling the wait states. After a power up or reset, the chip automatically defaults to two wait states, so your DSP32C programs should begin with instructions to program PCW appropriately for the speed of on board SRAM memory you are using. Usually this will be 0 wait states, for which the PCW register should be programmed with the value 0. In terms of performance, each wait state adds one quarter of an instruction cycle, or 20ns. Thus each wait state only degrades the performance by 25%, and the timing can be closely tuned to match the performance of the parallel port device or on board SRAM memory being addressed. Internal cycles on the DSP32C always run at 0 wait states, regardless of the PCW setting. See the EXAMPLES chapter for more details about wait state programming.

On the DSP_400 and DSP_MOD boards the 32 bit parallel ports are connected to the 50 pin header on the top edge of the board. For the pin out of this header, see the CIRCUITS chapter of this manual. The header has four sets of read and write strobes, and reading or writing to the addresses for the parallel ports will strobe the respective R/W pins and transfer the 32 bit data on the data bus. Note that when the parallel port is not being accessed, the R/W strobes are not asserted, and the parallel port data bus buffers are tristated.

Finally, memory block RAM1 is used by the DSPMON monitor for breakpointing. When executing a breakpoint, breakpoint code must be downloaded to the DSP32C to do a register dump when the breakpoint occurs. The internal memory block RAM1 in high memory is used by the monitor for this code. You may use RAM1 in your programs. However, if you execute a breakpoint, your code or data located there may be overwritten. To see which addresses DSPMON uses for its breakpoint code, refer to the DSPMON source code files dspmon.c and dspmon.dsp.

DSPMON SYSTEM MONITOR

The DSPMON program is a software tool for monitoring and controlling the DSP32C coprocessor boards by hand. Its commands are quite similar to those of debug or symdeb for the 80x86 on the PC, and users familiar with those programs should find DSPMON easy to use. Using DSPMON is a good way to get familiar with the coprocessor board. No damage can occur from anything done within DSPMON.

To start DSPMON, make sure the \srdsp\bin directory is on your current path, and execute it from the DOS command line. It takes a number of command line options, controlling the video mode, the files that are downloaded and so forth. See the EXAMPLES chapter at the end of this manual for a sample session and the \srdsp\examples subdirectory for practice programs.

DSPMON DOS COMMAND LINE OPTIONS

The following options can be specified on the DOS command line to control DSPMON.

dspmon {/50|/tty} [/n] [file.dso [file.dss]] [file.out]

where the options are:

/50	run in VGA 50 line full screen mode
/tty	run in tty mode
/n	cpu n, where n = 0,1,2,3 (DSP_MUL boards only)
file.dso	SR .dso object file to download
file.dss	SR .dss symbol table file to use
file.out	WE .out fully located coff file to download

The video mode /50 or /tty must be specified. Starting DSPMON with no command line options will result in a help screen.

Note that DSPMON is symbolic, and you can specify a DSPASM .dss symbol table file to use on the DOS command line. Symbolic information is *not* included in the DSPASM .dso object files. You must specify the .dss symbol table file to DSPMON to see the symbol names.

When using DSPMON with the DSP_MUL multiple cpu board, you need to specify the cpu you want to work with by the /n option. DSPMON can work on only one cpu at a time. It

maintains only one copy of the symbol table and other global variables to keep in track of the DSP32C.

You should set up batch files to start DSPMON with the command line options appropriate for your particular work. For example, you may want to specify only a base filename without an extension, and have DSPMON automatically start up in 50 line mode with both the .dso object and .dss symbol table files. The file dspwin.bat in \srdsp\bin is an example of such a batch file.

Finally, observe that if you try to start DSPMON with no board installed in the machine, you will get a warning, and then be allowed to continue on. Executing the program with no board installed will of course give incorrect results.

DSPMON COMMANDS

The following headings cover the commands DSPMON recognizes. These commands can be typed in at the DSPMON prompt. Several of them also can be invoked with function keys, as listed on the DSPMON help screen.

- **DISPLAY MEMORY** **d[b,w,l,f,i] [addr]**

This command displays DSP32C memory. The optional b, w, l, f, or i specifies whether the display is formatted as bytes, 16 bit words, 32 bit longs, DSP32C floats, or IEEE floats. Typical usage is:

dw	400	display 16 bit words at 0x400
df	2000	display DSP32C dspf floats at 0x2000
di	2000	display IEEE floats at 0x2000
dl		displays current memory as longs

where bytes, words, and longs are always displayed in hexadecimal. Note that displays of bytes are forced to be aligned on even boundaries because the DSP32C PC/PIO interface is run in its 16 bit mode.

- **ENTER VALUE** **e[w,l,f,i] [addr]**

Enter allows you to interactively set specific memory locations to specified values. The command will prompt you for the values to enter into successive memory locations, where typing an invalid entry will end interactive entry. With the f and i data type options, floating point numbers are expected as input and are formatted into DSP32C or IEEE floats respectively.

- **CLEAR MEMORY** **c [addr] [nbytes]**

This command clears memory to zero, starting at the specified address. If a starting address and/or number of bytes are not specified, then the values from the previous clear command are used. For the first clear command the starting address and number of bytes default to location 0 and 0x80 bytes respectively. The number of bytes to clear is always rounded up to an even number as with the display command, because the DSP32C PC/PIO interface is run in its 16 bit mode.

- **UNASSEMBLE** **u [addr]**

Unassemble displays memory as AT&T assembler mnemonics. It is your responsibility to know that you are disassembling program memory locations and not data in order to make sense out of what you are looking at. Typical usage is:

u 500 disassemble memory beginning at 0x500

Note that if you have also loaded a .dss symbol table you will be able to see and refer to your program symbols. For example, if you have a function named main, the command u main will disassemble your program at location main. Only program and goto labels are displayed in the disassembly, although you can use any program or data symbol where addresses are required in any DSPMON command.

- **DOWNLOAD FILE** **l filename**

Downloads object and symbol table files to the coprocessor board. The input file may be either be a DSPASM .dso or .dss or AT&T .out coff file. Each format has a leading identifier that the downloader examines to interpret the file correctly.

When downloading an AT&T assembler .out file, be sure that it has been completely located and does not have any relocatable sections left. The downloader must have all the information about where to download the program resolved.

Before downloading a file, the downloader halts the DSP32C to make sure its execution can't overwrite the downloaded program. After the file is downloaded, you can begin execution with the go command.

- **GO** **g [addr]**

The go command begins DSP32C execution at the current value of the program counter with an optional breakpoint set at the specified address. If a program has just been downloaded then execution will begin from location 0, otherwise it will continue on from the most recent breakpoint.

When a breakpoint occurs the registers and flags are displayed. *If the breakpoint is never reached* then you are caught in an infinite wait loop and any keyboard hit will halt and reset the DSP32C.

Together the download and go commands are used to control execution. Typical usage is:

l	test.dso	download a DSPASM object file
g	100	begin execution with a break at 100
g	108	continue, breaking again at 108

In order to breakpoint, the go command must download some code of its own to the DSP32C. The breakpoint code is downloaded to high memory at location 0xFFF800 in RAM1, and will overwrite any other program or data there. This address is in the on chip memory of the processor and is available for all DSP32C memory modes.

In addition, the DSPMON monitor places a {nop, goto breakpoint code, nop} sequence in your code at the breakpoint address. These instructions force a branch to the breakpoint code to dump the registers and flags when the breakpoint occurs. Because this branch must be in your code while it is executing, your code is modified during execution. After the breakpoint your original program instructions are restored, but the fact that the code is modified can lead to unexpected results in some situations. An example would be where a program is reading data immediately following the breakpoint address and you incorrectly read one of the {nop, goto breakpoint code, nop} words.

- **SINGLE STEP s**

Single stepping is the same as the go command with the breakpoint automatically set 4 bytes onward from the current program location. Since all DSP32C instructions are 4 bytes long, this will usually step onward from one instruction to the next. As with the go instruction, if the breakpoint is not reached, execution can be interrupted with a keyboard hit.

An important special case of single stepping to watch out for are the conditional if instructions. Single stepping *will not* necessarily move you on to the next logical instruction consistent with the processor flags after the if statement. DSPMON blindly places its breakpoint 4 bytes onward from the current location, and this may be an unreachable portion of code. *You* must examine the flags and set a breakpoint with the go command at a location you know will be reached for successful conditional execution.

- **HALT h**

Halt stops the DSP32C processor, asserting the PCR[0] bit on the PIO port, and resetting the program counter to 0. When the chip is halted you can still display and examine all memory.

- **HELP** **?**

This command displays help information about the DSPMON commands and function hot keys.

- **SYMBOLS** **symbols**

This command displays the current symbol table information. To load a symbol table you can either specify a .dss file on the DSPMON DOS command line, or explicitly use the l command.

- **ENABLE PIF INTERRUPT** **eni**

This command sets the value of the PCR[2] ENI bit to 1, enabling assertion or negation of the DSP32C PIF pin which is connected to the PC interrupt header on the board. Once enabled, the PIF pin is set and cleared by reads and writes to the PIR register. Also, note that the ENI bit is automatically cleared by a reset of the DSP32C, and must be reenabled after starting DSPMON or executing a go command.

- **DOS SHELL** **dos**

This command starts up a DOS shell, so you can issue DOS commands without quitting your current DSPMON session. To return to DSPMON from the DOS shell issue the exit command at the DOS prompt.

- **QUIT** **q**

Quits the monitor. This command does not change any DSP32C memory, so that if DSPMON is restarted after quitting, the same memory values will be there.

DSPTOOLS SYSTEM UTILITIES

INTRODUCTION TO DSPTOOLS

The DSPTOOLS system utilities library is a collection of functions for controlling the execution of the DSP32C coprocessor boards from C programs running on the PC. With it, you can upload and download the DSP32C boards from programs you write. In fact, the DSPMON monitor is an example of a program that makes use of these functions, where each operation allowed in the monitor corresponds to a function call in the DSPTOOLS library.

The library has been written to be called from C and to be compatible with either the Microsoft QuickC v2.5 or Borland C++ v3 compilers. The compiler used to build the library is indicated on the disk label. Batch files have been included to do a complete rebuild with either compiler if needed. Note that usually you will not be successful in linking Microsoft and Borland object and library files together and must be sure to use DSPTOOLS built with the correct compiler.

To begin using the DSPTOOLS library in your programs, you should do the following two things:

- *Always* include dsptools.h in your source code files using the functions
- Link with the library in the appropriate memory model like:

```
> link yourprogram,,,cs_tools.lib
```

It is important to *always* include dsptools.h in your source code files using DSPTOOLS. This file has C prototypes for all the functions and guarantees you are calling them correctly. Several of the functions use far and huge pointers, and the protos will automatically force any coercions of near pointers to far or huge that are required. Calling a function with the incorrect pointer type will often crash the PC, and including dsptools.h avoids this.

To make the DSPTOOLS library easy to use, you should also include the directories \srdsp\include and \srdsp\lib on the paths your compiler automatically searches for include and library files. This makes compiling and linking easy to do, and as explained in the INSTALLATION chapter, *avoids* creating multiple copies of the system files on your hard disk. Having only a single copy of the system software on your disk makes maintaining your software and installing future upgrades much easier.

The DSPTOOLS library is supplied in four memory models for the PC. They are small, medium, compact, and large. Each file is called cx_tools.lib, where x indicates the model:

cs_tools.lib	small model
cm_tools.lib	medium model
cc_tools.lib	compact model
cl_tools.lib	large model

Programmers using huge model should link to the large model library, cl_tools.lib.

It is important to link to the correct library for the memory model you are using. Linking to the incorrect model, or to a library built with the wrong compiler, will usually fail with either a compiler, linker, or runtime error. Note that the same include file, dsptools.h is used regardless of the memory model you are using.

For users interested in seeing or customizing how the library works, the source code for DSPTOOLS can be found in the \srdsp\dsptools directory. Also included in that directory is a make file with the commands that were used to build it. Note that the DSPTOOLS source code is split between two files, a higher level dsptools.c and a lower level dspio.asm. This has been done to give the best performance, and to provide mixed language programmers a way to avoid the C runtime library if necessary. Because it does not reference functions like printf, the lower level dspio.asm does not pull in the C runtime when linking. However, it still has enough functionality to accomplish basic interfacing tasks with the DSP32C boards. Users wishing to run stand alone may link to dspio.obj directly. This technique has been used with the DSPMATH library which can be called from either C or FORTRAN, and has no references to the C runtime. Usually however, we recommend linking with one of the cx_tools.lib files and making use of the full functionality of DSPTOOLS.

Finally, note that the DSPTOOLS library supports all three SR DSP32C boards, DSP_400, DSP_MOD, and DSP_MUL. The hardware interface to each of these boards is basically the same, and the same low level functions can be used with any of them. To fully utilize the multiple cpu board, DSP_MUL, there is an additional function, select_cpu, which can be called to control which cpu has its PIO interface connected to the PC bus. This function can be called at any time with little overhead. See the select_cpu function header below for more details.

The following pages describe each DSPTOOLS function, and are organized by functional area. Refer to the index for an alphabetical listing. Several example programs using the DSPTOOLS library are also included with the supplied software in the \srdsp\examples directory.

INITIALIZE THE DSP32C

C Usage:

```
#include "dsptools.h"

int init_32C( void);
```

Summary:

This function initializes the DSP32C, halting it, and programming its PIO port for the 16 bit mode the PC uses for communicating with the coprocessor boards. It *must* be called to initialize the DSP32C *before* calling other DSPTOOLS library functions. If everything is successfully initialized a 1 is returned, otherwise 0. If no coprocessor board is present in your PC, the function will return a 0.

If you are using the DSP_MUL multiple cpu board, you *must also* call the select_cpu function *before* calling init_32C to specify which cpu you want to initialize. Select_cpu and init_32C pairs should be called for each cpu installed on the board. See the select_cpu function for more details.

Init_32C normally assumes you are working with the board at its usual default PC/IO base address. However, as mentioned in the INSTALLATION chapter, there are two ways to override this default. One way is to explicitly set the value of the global baseio variable in your program *before* calling init_32C. If baseio is nonzero, init_32C will assume an override is in effect and use the current value. Second, if init_32C doesn't find that baseio has been explicitly overridden, i.e. baseio is zero, it will check DOS environment space to see if an override has been specified there. If you have specified a DSP32C_BASEIO DOS environment variable as described in the INSTALLATION chapter then that value will be used for baseio. If neither an explicit override or DOS environment variable is found, init_32C will assume the normal default base address of 0x280.

Finally, for installations where multiple boards will be run in one machine, note that the value of baseio can be changed at any time in your program and the DSPTOOLS library will communicate with the board at the current baseio address. By setting up the address of each board as a defined constant, you can switch from one to another. For installations with a single board that must run at a nonstandard address, setting the DOS environment variable DSP32C_BASEIO is probably easiest method of setting baseio.

CPU SELECTION (DSP_MUL boards only)

C Usage:

```
#include "dsptools.h"

void select_cpu( unsigned n);
```

Summary:

This function selects which cpu on a multiple cpu board, DSP_MUL, is currently connected to the PC bus. It *must* be called at least once to make a cpu selection. Be aware that after powering up, no default cpu selection is automatically made, and in fact it is likely no communication with the board will be possible at all unless select_cpu is called. The function works by poking the cpu selection register on the DSP_MUL board with the number of the selected cpu, where n = 0, 1, 2, 3.

The select_cpu function can be called at *any* time in your programs to change which cpu you are communicating with on the PC bus. The function is quite fast, and very little overhead is incurred in changing cpus. Furthermore, changing the cpu selection has no effect on any DSP's execution. Each processor will continue executing no matter which one is currently connected to the PC bus.

Note that when initializing the board, you *must* call select_cpu *before* calling init_32C to initialize a particular cpu. You should have select_cpu, init_32C pairs for each cpu that must be initialized. Failing to preceded init_32 with a call to select_cpu will result in improper initialization. Once each cpu is initialized, you can intersperse calls to select_cpu with calls to any other DSPTOOLS functions to carry out your operations.

The cpu selection register has its own base address dip switch on the DSP_MUL board and corresponding base address global variable basesel. The default value for basesel is 0x2A0, where, as with the baseio, this can be overridden by setting the DOS DSP32C_BASESEL environment variable or by explicitly setting the global variable basesel in your programs. The order in which select_cpu checks for overrides is the same as init_32C described previously. For details on setting the 8 position basesel DIP switch on the DSP_MUL board, refer to the INSTALLATION chapter.

Finally, note that calling select_cpu on the DSP_400 and DSP_MOD single cpu boards will probably have no effect on your system. The single cpu on those boards is permanently selected at the baseio address, and unless some other device is at the basesel address, calling select_cpu will execute with no result.

BASEIO AND BASESEL GLOBAL VARIABLES

C Usage:

```
#include "dsptools.h"

extern unsigned baseio;
extern unsigned basesel;
```

Summary:

These variables are defined in the DSPTOOLS library and specify the PC/IO space base addresses for the DSP32C PIO registers and DSP_MUL cpu selection register the library will use.

For the most part, single board systems running at the default PC/IO base addresses will never have to worry about these variables. The library automatically takes care of setting up their values when init_32C or select_cpu are called, and using them in communicating with the boards.

Users with applications running multiple boards can use baseio and basesel to control which board they are communicating with. Simply set their values *before* calling init_32C or select_cpu to control which board is addressed. Note that *any* nonzero value of the baseio and basesel global variables will override the usual system defaults and environment variables. See init_32C and select_cpu for more details.

START AND STOP DSP32C EXECUTION

C Usage:

```
#include "dsptools.h"

void go( void);
void halt( void);
```

Summary:

These functions start and stop DSP32C execution. They work by setting the PCR[0] bit on the PIO port of the processor.

The go function initiates a processor reset, starting execution from location 0. Before calling go, executable code that has either a program or branch instruction at location 0 should be downloaded to the board. In addition to resetting the processor, the go function also makes sure the PIO port is programmed for the 16 bit DMA mode the PC uses to communicate with the DSP32C.

The halt function stops execution. Once your program is finished, you may wish to halt. While the processor is halted, the PC can still communicate with the chip and the on board memory through the PIO.

Most users will not have to worry about setting various bits on the PCR control register to control execution. They can simply call go and halt. See the EXAMPLES chapter for typical programs.

Note also that the PC can access all of the on board memory *while* the DSP32C is executing. The PIO interface automatically takes care of transparently making any memory accesses for the PC *without* degrading the performance of the DSP32C.

READ AND WRITE PIO REGISTERS

C Usage:

```
#include "dsptools.h"

int pcr( void);
int pir( void);
int esr( void);
int emr( void);

void eni( unsigned newstate);
void set_pir( unsigned value);
```

Summary:

The functions pcr, pir, esr, and emr, access their respective PIO registers, returning their values. These registers have status information about the DSP32C, and are described in the AT&T DSP32C Information manual. The functions eni and set_pir respectively set the eni bit and the pir register to the specified value.

For exchanging small 16 bit messages with the PC, the PIR register is very useful. It can be read and written without disturbing the other aspects of the PIO. This often makes it ideal for passing status messages about the progress of a DSP32C calculation to the PC.

In addition, the PIR register can also be used for interrupting the PC. If interrupts have been enabled with the eni function, then writing or reading the PIR register from the DSP32C side sets or clears the PIF pin on the DSP32C chip. This pin is connected to the interrupt header on the board which can then be jumpered to one of the interrupt lines on the PC bus.

Note that as with most interrupting schemes, the PIF pin can be independently enabled or disabled by the eni bit on the PIO's PCR register. Calling the function eni with newstate = 1 or newstate = 0 will enable or disable the PIF pin respectively. After a DSP32C reset, eni is cleared.

Finally, if the DSP32C has set the PIF pin by writing to the PIR register, then reading it from the PC side will clear it. In effect, the PIF signal is like a FIFO status flag for the PIR register.

READ AND WRITE DSP32C MEMORY

C Usage:

```
#include "dsptools.h"

void get_32C( unsigned long src, void huge *dest,
                                    unsigned long nbytes);

void put_32C( void huge *src, unsigned long dest,
                                    unsigned long nbytes);
```

Summary:

These functions are for reading and writing data from and to the DSP32C coprocessor board. They use the 80286 insw and outsw instructions for maximum performance in moving arrays.

For get_32C, the source is a 24 bit address on the DSP32C, passed as a 32 bit unsigned long, and the destination is a huge pointer to the PC's memory.

For put_32C, the source is a huge pointer to the PC's memory, and the destination a 24 bit DSP32C address, again passed as an unsigned long.

In both functions, nbytes is the number of bytes to transfer. Any number of bytes up to 16 Mbytes can be specified, and the two functions will automatically take care of making sure any DOS segment boundary crossings are handled correctly. Be aware that when the transfer actually happens, the number of bytes is always rounded up to an even number. The PIO port is driven in its 16 bit mode by these functions, so only an even number of bytes can be transferred. The number of bytes to transfer can usually be computed in C programs with the sizeof operator. Note that it is easy to overwrite the PC's memory with these functions and it is up to the programmer to make sure this doesn't happen.

DOWNLOAD BINARY OBJECT FILES

C Usage:

```
#include "dsptools.h"

int sr_load( char *filename, int report);
int we_load( char *filename, int report);
```

Summary:

These functions download Symmetric Research DSPASM .dso and Western Electric AT&T coff .out object files to the DSP32C coprocessor board.

When either function completes its download successfully, a 1 is returned, otherwise 0 is returned for failure. User programs should always check the return value to ensure that a successful download occurred.

The filename should point to a string with the pathname of the file to be downloaded. Any legal DOS pathname to the file is allowed.

The report flag controls whether or not messages about the status of the download are sent to stdout. If it is zero, no messages are sent. However, if it is nonzero, then messages about the sections being downloaded and their sizes are printed out. The defined constants REPORT and NO_REPORT in dsptools.h are for use with this flag.

Finally, before beginning the download, both functions halt the DSP32C. This prevents a currently executing program from overwriting a file while it is being downloaded, and provides a known state for starting execution.

DOWNLOAD A HEX OBJECT ARRAY

C Usage:

```
#include "dsptools.h"

int sr_hexload( unsigned long far *array);
```

Summary:

This function is like the sr_load function, except that it takes its input from an array in memory, rather than from a disk file. For some applications it is preferable to download from an array included in a DOS .exe program rather than from a separate file. This eliminates the need to divide an application program between a DOS .exe and a .dso file with the DSP32C code.

To create an array of hex instructions, the DSPASM assembler can be run with the /h option to create a .dsh ASCII file that can be included in C programs. The function sr_hexload can then be used to download the included object file.

Finally, like the other downloaders, this function halts the DSP32C before downloading, and returns a 1 or 0 indicating the success or failure of the download.

FLOATING POINT FORMAT CONVERSION

C Usage:

```
#include "dsptools.h"

void ieeetodspf( float *x);
void dspftoieee( float *x);

char *ieeetoa( float *x);
char *dspftoa( float *x);

int atodspf( char *line, float *x);
int atoieee( char *line, float *x);
```

Summary:

These functions convert between IEEE, DSP32C, and ASCII floating point formats.

The functions ieeetodspf and dspftoieee, convert to and from binary IEEE single precision format and the internal binary floating point format used on the DSP32C. x must point to the number to be converted, and the converted result is returned in the same location.

For converting binary floating point numbers to ASCII strings, the functions ieeetoa and dspftoa take pointers to their respective binary input values, and return a pointer to an ASCII string with the converted number. The output string is kept as static data in the functions, and repeated calls to them will overwrite previous conversions.

Finally, to convert in the other direction, from ASCII to a binary float, the functions atodspf and atoieee take a pointer to a string with the ASCII floating point number, and return the converted binary version at the location pointed to by x. If they are successful, they return the number of bytes scanned off the ASCII string, otherwise, they return a zero.

See the AT&T DSP32C Information manual for a discussion of the DSP32C and IEEE floating point formats.

DISASSEMBLE A DSP32C INSTRUCTION

C Usage:

```
#include "dsptools.h"

void decode( unsigned long value, char *line);
```

Summary:

This function disassembles a single binary DSP32C instruction into its AT&T ASCII mnemonics. The binary instruction should be in the 32 bit unsigned long value, and the line pointer should point to a string buffer at least 64 bytes long.

If the instruction can be successfully decoded then the ASCII mnemonics will be left in the string buffer pointed to by line. However, if the value is unrecognizable as a machine instruction, then nothing will be left in the NULL terminated string buffer.

All DSP32C binary instructions are 32 bits long. The binary encodings can be found in the AT&T DSP32C Information manual.

DSPASM DSP32C ASSEMBLER

INTRODUCTION TO DSPASM

The DSPASM assembler is designed to process source files with AT&T mnemonics and high level control constructs to produce located object files that can be directly downloaded and executed on the coprocessor board. It is ideal for code development requiring fast edit and execute cycles. Some of the features of DSPASM are:

The source code is free format, along the lines of C and Pascal. Input is comprised of data, functions, and statements terminated with semicolons.

Symbolic register names and structures are supported for program readability.

All CAU arithmetic instructions are assembled to a 24 bit default. r1 = r2 + r3 automatically assembles to what would be r1e = r2 + r3 in the AT&T assembler, preventing incorrect truncation or sign extension of 24 and 16 bit quantities.

By default, nops are emitted where needed for all CAU instructions, relieving the programmer of having to worry about CAU latency. Nops are *not* emitted for DAU instructions.

C like conditional syntax is used, with high level if, while, and for loops supported.

The .dso object files produced by DSPASM are located and ready to be downloaded. There is no linking.

Forward references are allowed, with the assembler running optionally in one or two pass mode.

This chapter covers various aspects of using DSPASM with example source code fragments listed throughout, and a complete program given at the end. Several working programs are also listed at the end of this manual in the EXAMPLES chapter, and in the examples subdirectory on the supplied disk.

INVOKING DSPASM AT THE DOS COMMAND LINE

First, make sure the dspasm.exe program from the supplied disk is on your current DOS execution path. Then at the command line execute:

dspasm [/{1,2}] [/l] [/s] [/sa] [/sc] [/h] [/n{+,-}] file

where the options are:

/{1,2}	one or two pass assembly	(default = /1)
/n{+,-}	automatic CAU nops on or off	(default = /n+)
/l	emit a listing file in AT&T mnemonics	(output suffix = .dsl)
/s	emit the symbols in ASCII format	(output suffix = .dss)
/sa	emit the symbols in DSPASM format	(output suffix = .dsa)
/sc	emit the symbols in C format	(output suffix = .dsc)
/h	emit the output as a hexified C file	(output suffix = .dsh)

If the input file assembles successfully then DSPASM will return a DOS error code of 0. Otherwise, an error code of 1 will be returned, notifying the invoking make or batch file of the error.

FILENAME CONVENTIONS

The following filename extensions are used by DSPASM:

.dsp	input source code file
.dso	binary output object file
.dsl	listing in AT&T mnemonics
.dss	symbol table dumped in simple ASCII format
.dsa	symbol table dumped in DSPASM format
.dsc	symbol table dumped in C include format
.dsh	hexified C format object file

While the .dsp extension on the input source code file is not absolutely required, this convention is followed throughout this manual and the supplied software. The advantage of using the above extensions is that files generated with DSPASM are easily distinguished from those generated with the AT&T assembler or other language tools on the PC.

ERROR MESSAGES

When DSPASM encounters an error, assembly stops and an error message is generated. The error message is sent to stdout, and is of the form:

```
filename(linenumber) : Error : message text :
        line of code with error
```

In general, the line of code with an error will have an arrow pointing to the input DSPASM was unable to recognize. Usually, the arrow will point to exactly where the error occurred. Occasionally however, you will have to look further back up in the source to find the error. This can be particularly true with mismatched curly braces {}.

This error message format is compatible with editors like Brief and Epsilon for automatic error message location while editing in one window and assembling in another.

USE OF C PREPROCESSORS FOR MACROS AND DEFINES

DSPASM does not support macros and #defines. Like the AT&T assembler, you should use a preprocessor for these features. The source code format is compatible with most C preprocessors. To use the Microsoft QuickC preprocessor for example, use the /P option to create a .i file that can then be fed to DSPASM. Note that many C preprocessors will accept only filenames with a .c extension. You may have to rename your .dsp file to a .c extension temporarily when using a C preprocessor. Finally, when debugging a DSPASM error, always be sure to look at the input file being given to DSPASM to screen out any subtle errors introduced by the preprocessing.

SOURCE CODE FORMAT

Legal DSPASM source code is comprised of comments, data, and functions, interspersed with location directives to control where data and code will be located. As in C, the input is free format, with spaces, tabs, and newlines being treated as white space that is ignored. Although the input is free format, you should follow an indentation convention as in Kernighan and Ritchie or as in this manual for program readability.

Comments are delimited as in C by matching pairs of /* and */, and can span newlines as necessary. Nested comments are not allowed and will result in an error message.

Data declarations are denoted by one of the data type keywords described below, followed by the variable names. Variables can be dimensioned and also initialized. See the DATA subtitle below for details.

Functions are simply introduced by their name, with the body of the function being enclosed in curly braces. Between the curly braces are DSP32C instructions. No subroutine window arguments are allowed in DSPASM, and consequently no () follow function names. Function arguments should either be passed by global variables, or on a stack maintained by the DSPASM programmer. See the FUNCTION subtitle below for more details.

Putting this together, a source code fragment might look like:

```
constant N = 5;         /* declare a constant */
long x, y;              /* declare some global data */

add {                   /* declare a function */

   r1 = *x; r2 = *y; r3 = r1 + r2;
   return;
   }
```

and so on with more data and functions.

All constant names, variable names, function names, and goto labels are maintained in a single global symbol table in DSPASM. Named registers and structures are maintained separately, and are scoped only to the function in which they are declared. As in most languages, it is not a good idea to use keywords for symbol names. The results will be unpredictable. See the KEYWORDS subtitle below for a list of reserved keywords.

Input source code files may be of any size. DSPASM processes them in 64K chunks, keeping as much in memory at one time as possible.

Finally, DSPASM may be run either as a one or two pass assembler. When run as a single pass assembler, *all* symbols must be declared *before* they are used. While most programs can be organized for single pass assembly, many times it is more convenient or logical to have forward references to symbols not yet declared. To assemble with two passes, use the /2 option on the DSPASM command line. In two pass mode, DSPASM collects the values of all symbols on the first pass, and then completes the assembly with the resolved values on the second pass. See the EXAMPLES chapter of this manual for an example of the same program written to assemble in one or two passes.

CONSTANTS

There are only two kinds of constants in DSPASM, hex and floating point. Hex numbers are comprised of the digits 0-9, and the letters a-f or A-F. Optionally, hex constants may have a leading 0x or a trailing L for compatibility with C syntax conventions, but these are not required. Decimal constants are not supported in DSPASM. All floating point numbers are assumed to be in base 10, with leading digits, a required decimal point, and an optional fractional part with a base 10 exponent. Examples of each kind of number are:

1234	a hex integer constant
0FEEC	another hex integer constant
-1	two's complement of 1 as a hex integer
80	the hex number 80 (decimal 1240)
1.	a floating point 1
1.5E10	another floating point number
-1.E10	a negative floating point number
80.	the floating point number 80

The binary encoding of floating point numbers depends on the context in which they are used. If they appear in a statement referring to DSP32C floats, then they will be encoded in the DSP32C floating point format, otherwise, if they appear in a context referring to IEEE floats, they will be encoded as IEEE floats.

If desired, hex constants can be given symbolic names with the constant statement. Typically this would be used for an array dimension or an important address requiring a mnemonic name. Some examples would be:

```
constant N      = 5;
constant NDIM   = N*4;
constant DATA   = 100;
```

The symbols `N`, `NDIM`, and `DATA` would be listed in the DSPASM symbol table, and can be referenced later on in the program. Note that hex expressions can be used almost anywhere a hex constant is taken by the assembler. All the arithmetic and bitwise operations of C can be used, where the usual precedence rules are observed.

Finally, the special symbol . is reserved to stand for the current location in the assembly. It can be used as a hex constant anywhere a regular hex constant could appear.

LEGAL SYMBOL NAMES

Constant, variable, function, and goto symbol names may be of any length, and are unique in all their characters. All symbols must lead off with an alphabetical character or underscore, and after the first character, digits are also allowed. Symbol names are case sensitive, so that two symbols differing only by case are different. A few examples of legal symbol names are:

```
func        _func        _123_func
```

and so on. As in most languages, keywords should not be used as function names, variables, or goto labels.

KEYWORDS

The following keywords are recognized by DSPASM:

a0-a3	**aeq**	**age**	**agt**
ale	**alt**	**ane**	**auc**
aus	**avc**	**avs**	**byte**
call	**cc**	**cs**	**dauc**
do	**dsp**	**dspf**	**emit**
eq	**fbc**	**fbs**	**float**
float24	**for**	**ge**	**goto**
gt	**hi**	**ibe**	**ibf**
ibuf	**ic**	**ieee**	**if**
ifaeq	**ifagt**	**ifalt**	**iff**
int	**int24**	**ioc**	**ireq1_hi**
ireq1_lo	**ireq2_hi**	**ireq2_lo**	**ireturn**
le	**long**	**loop**	**ls**
lt	**mi**	**ne**	**nop**
nops_off	**nops_on**	**obe**	**obf**
obuf	**oc**	**pcw**	**pde**
pdf	**pdr**	**pdr2**	**pie**
pif	**piop**	**pir**	**pl**
r1-r22	**return**	**round**	**seed**
syc	**sys**	**vc**	**vs**
while	**word**		

These keywords should not be used as function names, variables, or goto labels.

THE LOCATION DIRECTIVE

You must designate where in the DSP32C address space your data and functions will be placed. After a reset for example, the DSP32C begins execution at location 0, and you must have code there for the processor to begin executing.

The current assembly location is set by a hex constant between angle brackets <>. Location directives may be freely interspersed between data and functions as needed, but *cannot* be used within a function. Each time the location directive is used, it causes a new output record to be emitted to the .dso object file. See the .DSO OBJECT FILE FORMAT subtitle below for more details on the object file format.

An example of the location directive would be:

```
constant RESET = 0;           /* memory addresses */
constant DATA = 100;

<DATA> long x;                /* some data */

<RESET> myfunc {              /* reset entry point */

   r1 = 5; *x = r1;
}
```

By default, assembly begins at location 0, so that if forward referencing is used and the reset entry point is listed first in the source file, then often no location directives at all are needed.

DATA

All data is global in DSPASM, and must be declared outside of functions. Furthermore, data objects are *always aligned to 4 byte boundaries*. The automatic alignment of data reduces many easily made programming mistakes. There are five recognized data types:

byte	8 bit integer
word	16 bit integer
long	32 bit integer
dspf	DSP32C format floating point number
ieee	IEEE format floating point number

Typical data declarations are much as they are in C. Some examples are:

```
/* a single byte */                   byte c;
/* an array of 5 words */             word[5];
/* declare and initialize x */        long x = 123;
/* a floating point array */          dspf y[2] = { 1.2, 3.5 };
/* array of unknown dimension */      ieee pixels[];
```

Uninitialized variables and arrays are *not* emitted to the .dso output file. Initialized arrays *are always* emitted in their entirety to the .dso output file. Declaring large initialized arrays will significantly increase the size of the output object file. It is usually better practice to leave them uninitialized if possible.

In addition, when assembling with two passes, it is important that all array dimensions be resolved on the first pass. This is required so the assembler can compute the locations of any following data or functions on pass one for use on pass two. Arrays that fail to have their dimensions resolved on pass one will result in an error message.

Finally, DSPASM maintains little type checking information. Any symbol can be used in nearly any context with no error messages. You are responsible for making sure of the proper usage of all symbols.

FUNCTIONS

Functions are declared by giving the function name, followed by the body of the function enclosed in curly braces. DSPASM does not maintain an argument stack for functions, so no parentheses () are allowed after the function name. An example of a function to count down and save values in an array might be:

```
long array[20];

count {

   r1 = 20; r2 = array;

   while ( r1-- > 0  )
          *r2++ = r1;

   return;
}
```

Note that unlike C, DSPASM does not automatically emit a return at the end of a function. If there is no return, execution will simply continue down into whatever follows the function declaration.

For nested function calls, you must maintain your own stack for return addresses. The DSP32C does not maintain a hardware stack, it simply saves the return address in a register when a call is made. See the EXAMPLES chapter of this manual for how to set up and use a stack.

NAMED REGISTERS

Within a function, local variables are usually kept in registers. With its large register set, the DSP32C supports this programming style fairly well. While registers can be referenced by their numbers, such as r1 r2 r3, it is often better to give them meaningful mnemonic names. Naming registers is done with the register keyword in DSPASM. An example is:

```
myfunc {

   register i = r1, j = r2, sum = r3;

   i = 5;
   j = 3;
   sum = i + j;
}
```

Any register declarations must appear at the beginning of a function before any instruction statements. Their names are scoped only to the function in which they are declared, and may be reused again in other functions.

STRUCTURES

In addition to named registers, structures may also be used within functions. By declaring several structures with the same base address, the same area of memory can be referred to in a variety of ways. An example of the syntax for declaring a structure is:

```
long input_array[10];

myfunc {

   /* Declare a struct at input_array */
   struct input_array { long x, y; word z; } input;

   /* Do some calculations with it. */
   r1 = input.x;       /* r1 = address of input.x */
   r2 = *input.y;             /* r2 = contents of input.y */
}
```

The fields in a structure can be any of the primitive data types, long, word, byte, dspf, or ieee, but arrays are not allowed. The exact syntax is:

```
struct N { type1 x, y, .. ; type2 z ..; } name;
```

where N is the base address of the structure, type is one of the primitive data types, and name is the symbolic name for referring to the structure with. Structures may only be dereferenced with the . operator, where name.field is treated by the assembler as a constant with the address of the field referred to.

Like named registers, structures are scoped only to the function in which they are declared and may be reused again in other functions. This allows a later function to refer to same area of global memory with a different layout and naming conventions.

INSTRUCTION STATEMENTS

Following the named registers and structures in a function are any instruction statements comprising the body of the function. For the most part, instruction mnemonics in DSPASM are exactly the same as those listed in the AT&T DSP32C Information manual and used by the AT&T assembler. However, there are some important differences. These changes have been made in DSPASM to help reduce programming errors and improve readability. Some of the most important changes are in the conditional and looping constructs, which have been given C like syntaxes and are discussed in detail in the next section. Changes to the basic AT&T instruction syntax are covered in the following paragraphs.

The first difference is that all CAU arithmetic instructions are assembled to a 24 bit default. The DSPASM instruction r1 = r2+r3, assembles to what would be written as r1e = r2+r3 in the AT&T mnemonics. The old short format 16 bit DSP32 compatible instructions are not supported. Assembling to a 24 bit default is helpful in preventing truncation of 24 bit addresses to 16 bits, and in preventing the sign extension of 16 bit addresses to incorrect 24 bit addresses.

The second difference is for specifying the size of CAU register loads and stores. Rather than using optional e, h, and l specifiers as in the AT&T assembler, DSPASM uses .b .w and .l to specify byte, word, and long transfers. These specifiers must always appear on the right hand side of an equation, and in addition to specifying the size of a transfer, they will also control the size of any post increments or decrements. As an example, the program fragment given above for counting down can be changed to save words rather than longs as follows:

```
word array[20];

count {
   r1 = 20; r2 = array;

   while ( r1-- > 0  )
```

```
            *r2++ = r1.w;

      return;
   }
```

where the increment now will add 2 to register r2 on each pass through the loop because a word save has been specified.

The third difference is that the call and return instructions automatically use register r14 for saving the return address. They assemble to what would be written in AT&T mnemonics as call function (r14) and return (r14). An example would be:

```
add { r3 = r1 + r2; return; }

main {
   r1 = 2; r2 = 3;
   call add;
}
```

Register indirect computed subroutine calls are also allowed in DSPASM. Written to use a computed function call, the above code fragment would read as:

```
add { r3 = r1 + r2; return; }

main {
   r1 = 2; r2 = 3; r3 = add;
   call r3;
}
```

Finally, for any of the dyadic CAU instructions the C like op = syntax may be used. The following are some examples:

```
   r1 += r2;
   r1 #= r2;
   r1 *= 2;
   r1 *= 4;
```

where the *= 4 instruction will emit *= 2 twice.

IF, WHILE, FOR, DO ... CONTROL SYNTAX

Like the syntax differences with the AT&T assembler noted above, the if, while, and do statements have been changed to make programming the DSP32C easier. Generally, these statements are more like they would be for C.

The if statement has been changed to read as follows:

```
if ( cond ) { statements }
```

where cond can be any of the ca, da, or io conditions, and the statements between the curly braces *are executed* if the condition is *true*. The assembler automatically emits branches and nops associated with the if statement for proper execution. An example would be:

```
if ( gt ) { r1 = -r1; r2 = r3; }
```

To see the additional statements emitted to the object file in AT&T mnemonics, assemble this code fragment with the /l listing option.

In addition to using the AT&T mnemonics for the CAU flag conditions, relational operators can be used to specify some of the conditions. The alternative relational operator syntax is:

>	for	gt
<		lt
>=		ge
<=		le
!=		ne
==		eq

with this syntax the above if statement can be written as:

```
if ( > ) { r1 = -r1; r2 = r3; }
```

Not only can a relational operator be used to replace an AT&T CAU flag condition, a register expression can be used to implicitly emit the compare instruction normally preceding an if. The following is legal:

```
if ( r1 > r2 ) { statements; }
```

where any of the relational operators could be used. The full syntax for the if statement is:

```
if ( rN[++,--] relop {rN,N} [,caexp] ) { ... }
```

The optional ++, --, or ,caexp can be used to force an instruction on the nop normally following the generated if statement. An example of the full conditional if syntax is:

```
if ( r1 <= r2 , r1 += 4 ) { r5 = r6; }
```

which would result in the following AT&T mnemonics:

```
   r1-r2
   if ( gt ) goto skip
   r1 = r1 + 4
   r5 = r6

skip: ...
```

As in C, an initial if may be followed by else if and else blocks. The following code fragment is an example:

```
if ( r1 > N ) {

   if ( r1 == N+2 )
          { r4 *= 4; r4++; }
   else
          r4 = 1;
   }

else if ( r1 == N )
   r3 = r4;

else
   r3 = 7;
```

While and for use the same basic conditional syntax as the if statement. The following shows their use:

```
while ( i-- > 0 ) {
   for ( j = 0 , k = 4 ; j < N ; j++ )
          *A = a0 = *B - *C;
   }
```

where named registers have been used for program clarity. Any number of comma separated CAU expressions may appear in the first argument of the for. The middle conditional part is

the same as the if syntax, and third argument can be any single CAU expression. This trailing expression is emitted instead of a nop with the looping branch at the bottom of the for.

With the if, while, and for, along with named registers and structures, you can make your DSPASM assembler program look remarkably like C. These constructs make programs easier to write and maintain.

In addition to the if, while, and for C like statements, there are also three other conditional statements that are supported by the assembler and map into the instruction set of the DSP32C.

The first is the DSP32C do statement. Do can be used to repeat up to 32 instructions 2048 times. The advantage of using do instead of a for loop is that instruction fetching is turned off while executing. This results in an automatic performance gain by a factor of 2 in many cases! There are two possible forms of the do statement:

```
do rN+1 { statements; }     do N { statements; }
```

In the first form, the statements are done rN+1 times where rN is a CAU register. The value of rN may be used inside the do statement, however be aware that it retains its initial value and does not decrement as the loop is executed. The second form executes the loop N times, where N may be any hex constant expression. The DSPASM assembler automatically counts the number of instructions between the curly braces and sets up the actual machine instruction accordingly.

An important restriction on the do statement is that *no branching* may be done inside a do loop. This makes sense if you think about the pipeline of the processor. Any branching will break the flow of control in the cached do instructions. The assembler will automatically give an error message for any illegal instructions inside a do loop.

The next construct is the loop statement. Its form is limited to the following:

```
loop { .. instructions .. } while ( rN-- >= 0 )
```

This syntax maps directly into the AT&T if (rN-- >= 0) mnemonic.

Finally, the DSP32C also has a "single instruction if" statement. DSPASM uses the keyword iff to denote these statements, and they are all of the form:

```
iff ( cacond ) caexp;
```

where the ca expression may not involve an immediate constant. Refer to the AT&T Information manual for more information about this form of if.

GOTO's

You can use goto's in DSPASM programs. Goto labels are any legal symbol terminated with a colon. Examples of goto statements are:

```
wait: goto wait;       /* waste some time */

goto special_case;     /* forward reference */
  special_case:
```

If you are going to use forward referenced gotos, be sure to assemble with the DSPASM /2 option to resolve all references on the second pass.

Goto labels are kept in the global symbol table maintained by DSPASM, and are scoped to the *source file* in which they appear. They are *not* scoped to the function in which they are declared as in C.

CUSTOM EMITS

Occasionally there is a need for emitting a hex constant directly to the .dso file from inside a function. This can occur when you want to have a special opcode for example. The intrinsic emit() function takes a comma separated list of hex expressions and emits them directly to the .dso file. An example of its use is:

```
constant MYOPCODE = 0x76543210;

function {

   emit( /* emit some custom opcodes. */
          MYOPCODE |0,
          MYOPCODE |2,
          MYOPCODE |3,
          MYOPCODE |4,
          MYOPCODE |5,
          );
}
```

AUTOMATIC LATENT CAU NOPS

For CAU instructions, the DSPASM assembler can automatically emit nop's as needed for latent instructions. Generally, register loads and stores, and conditional instructions must be followed with nops. Refer to the AT&T DSP32C Information manual for full details on instruction latency.

Automatic emission of CAU nops can be controlled in two ways. The first is from the command line, with the /n+ and /n- options. The default is the /n+ option, turning automatic nops on. The second is with the nops_on; and nops_off; statements inside a source file. These two keywords can be used to selectively turn automatic nops on and off as desired within a function. Once nops are turned off they stay off until a nops_on; is encountered elsewhere in the file.

DSPASM does *not* automatically emit nops for DAU instructions. Usually, tight control over the execution and performance of the DAU multiply accumulate pipeline is desired, and the assembler cannot make decisions as to the optimal use of DAU instruction latency. Refer to the AT&T DSP32C Information manual for details on DAU latency.

OBJECT FILE FORMAT

The binary object files generated by DSPASM are comprised of DSP32C code and data, with additional information included for locating the code and data. All .dso files are completely located by DSPASM, and do not need to be passed through a linker to resolve relocatable references. In addition, they are comprised entirely of 4 byte long words, so they can be easily examined in hex dumps without losing alignment. The file layout is comprised of a leading identifier, then following records with code and data. The specifics of the format is covered in the following paragraphs.

The first long word in a .dso file is an identifier, indicating that the file has been generated by DSPASM. The value of the identifier is 0x52535F5FL, which in bytes is "__SR". Downloaders and other software can examine the identifier to verify whether the file was generated by DSPASM or not.

Following the identifier are the records comprising the file. Each record has a header consisting of two long words:

```
struct record_header {
   unsigned long start;
   unsigned long nbytes;
   }
```

These two long words give the starting address and number of bytes to be located at the specified address. The code or data to be downloaded immediately follows the record header, and after that there may be additional records each with their own individual headers.

Finally, the file is terminated with a special record header whose start address is FFFFFFFF, (8 F's). Since this is greater than any valid DSP32C address, it cannot be confused with legal record headers, and serves as an end of records marker.

For an example of a .dso downloader, look at the sr_load() function in DSPTOOLS.

SYMBOL TABLE FILES

In addition to outputting binary object files, the DSPASM assembler can output hexified object and symbol table files suitable for including in C and DSPASM programs.

To obtain hexified output from DSPASM, invoke it with the /h command line option to create a .dsh file. Hexified object arrays are useful when you are working on an application and would rather not have the DSP32C code in a separate .dso file. By including the DSP32C code in a .c source file, you can have your entire application reside in a single final .exe file. Hexified object arrays included in C programs can be downloaded with the sr_hexload() function in DSPTOOLS.

In addition to producing hexified output, DSPASM can also dump its symbol table in a variety of formats. These can be very useful for including in other files that need to know about DSP32C addresses. The available command line options and resulting formats are:

/s	.dss ASCII dump in a simple format
/sa	.dsa ASCII dump in DSPASM format
/sc	.dsc ASCII dump in C include file format

The .dsc format is especially useful, allowing C programs to know the values of the symbols in the DSPASM file. In the .dsc include file, each DSPASM symbol will have the prefix DSP_ appended to it to avoid collisions with other symbols in the C program.

For an example of using both a hexified .dsh file and a .dsc symbol table file, refer to the DSPMATH library in the supplied software.

EXAMPLE PROGRAM

The following is a complete program for the DSP32C written in DSPASM assembler to assemble in a single pass. The program averages the values of two floating point arrays, saving the results in a third array.

```
constant RESET = 0, DATA = 100;      /* memory locations */

<DATA>                               /* allocate some data */
constant N = 3;
dspf
   half   = 0.5,
   array1[N]     = { 1.0, 2.0, 3.0 },
   array2[N]     = { 4.0, 5.0, 6.0 },
   result[N];

<RESET> main {

   r1 = array1;                      /* set up pointers */
   r2 = array2;
   r3 = half;
   r4 = result;

   /* do {} N times */
   do N {
          a0 = *r1++ + *r2++;
          nop; nop;
          *r4++ = a0 = a0 * *r3;
          }

   nop; nop; nop;                    /* flush DAU pipeline */

wait:    goto wait;                  /* wait for the PC */
}
```

After assembling this program with DSPASM, the .dso object file should be downloaded and executed with the DSPMON monitor. When finished, the result array should be 2.5, 3.5, 4.5.

In addition to this program, you should also refer to the EXAMPLES chapter of this manual and to the examples subdirectory on the supplied disk for additional practice programs.

DSP32C MATH LIBRARY

INTRODUCTION TO DSPMATH

The DSPMATH library is a collection of math functions for the DSP32C coprocessor boards. With this library functions can be called from high level languages without having to program the board directly. It is one of the easiest ways to begin getting results from the DSP32C boards. This chapter lists the functions included in the library and their C and Fortran interfaces.

The library functions have been written in C and DSPASM assembler. To build the library, either the Microsoft QuickC v2.5 or Borland C++ v3 compilers have been used as indicated on the disk label. Batch files have been included to do a complete rebuild with either compiler if needed. Note that usually you will not be successful in linking Microsoft and Borland object and library files together and must be sure to use DSPMATH built with the correct compiler.

To begin using the DSPMATH library in your programs, you should do the following two things:

- *Always* include dspmath.h in your C source code files using the functions, or include dspmath.for if you are using Fortran
- Link with the library in the appropriate memory model like:

```
> link yourprogram,,,cs_math.lib
```

It is important to *always* include dspmath.h or dspmath.for in your C and Fortran source code files using DSPMATH. These files have the prototypes for all the functions and guarantee you are calling them correctly. Several of the functions use far and huge pointers, and the protos will automatically force any coercions of near pointers to far or huge that are required. Calling a function with the incorrect pointer type will often crash the PC, and including dspmath.h avoids this. Also, in the case of dspmath.for for Fortran, the include file forces the Fortran compiler to follow C calling conventions when using the DSPMATH functions.

To make the DSPMATH library easy to use, you should also include the directories \srdsp\include and \srdsp\lib on the paths your compiler automatically searches for include and library files. This makes compiling and linking easy to do, and as explained in the INSTALLATION chapter, *avoids* creating multiple copies of the system files on your hard disk. Having only a single copy of the system software on your disk makes maintaining your software and installing future upgrades much easier.

DSPMATH already includes a copy of the DSPTOOLS library. When using DSPMATH you do not need to explicitly link to DSPTOOLS. You will also not usually need to call any functions in the DSPTOOLS library. The DSPTOOLS functions are called as needed by DSPMATH.

The DSPMATH library is supplied in four memory models for the PC. They are small, medium, compact, and large. Each file is called cx_math.lib, where x indicates the model:

cs_math.lib	small model
cm_math.lib	medium model
cc_math.lib	compact model
cl_math.lib	large model

Programmers using huge model should link to the large model library, cl_math.lib.

It is important to link to the correct library for the memory model you are using. Linking to the incorrect model, or to a library built with the wrong compiler, will usually fail with either a compiler, linker, or runtime error. Note that the same include file, dspmath.h or dspmath.for, is used regardless of the memory model you are using.

No initialization functions need to be called when using the DSPMATH. A call to any DSPMATH function will automatically make sure the board has been properly initialized and downloaded. You only need to call a library function to use it.

A simple C program using the library function inner is:

```
#include "dspmath.h"

main() {

   float x[3], y[3], result; int i;

   /* Set some values into x and y. */
   for ( i = 0 ; i < 3 ; i++ )
          { x[i] = i; y[i] = i; }

   /* Compute the inner product. */
   inner( x, y, 3, &result);

   /* Print out the result. */
   printf( "result = %e", result);
   }
```

To link and run this program after it has been complied, execute link example,,,dspmath; and then run example.exe from the DOS command line. More complete test programs for calling the library from C and FORTRAN are also included in the dspmath and examples subdirectories on the supplied disks.

In general, the minimum dimensions of the functions are those that would make sense, and the maximum dimensions are limited by the amount of memory installed on the board. For the

inner product function for example, the minimum array dimension would be 1, and the maximum dimension would be limited by the fact that each array entry takes up 4 bytes of memory. If you have a board with .5Mb of memory, then x and y could each have a maximum dimension of 65536 real entries. Sometimes the functions also have their dimensions limited by the algorithms they use. The FFT functions are an example. If there are limitations they are noted in the function descriptions that follow.

For reporting error status, the library maintains a global variable DSPMATH_status. This 16 bit integer can be checked after a calculation to determine if any errors occurred during the function just completed. See the description of DSPMATH_status on the following pages for its possible values.

Finally, a number of data type defines are used in the C interfaces listed on the following pages to improve readability. They are:

```
struct complex_single { float r, i; };

#define WORD                unsigned
#define FLOAT               float
#define COMPLEX             struct complex_single

#define INT_ARRAY           int huge *
#define LONG_ARRAY          unsigned long huge *

#define DIM                 unsigned long
#define INPUT               float huge *
#define OUTPUT              float huge *

#define COMPLEX_INPUT       struct complex_single huge *
#define COMPLEX_OUTPUT      struct complex_single huge *
```

`INPUT` and `OUTPUT` are used to denote data passed to and returned by the function. These data types are defined in dspmath.h and are available when it is included in C sources.

The following pages describe each DSPMATH function, and are organized by functional area. Refer to the index for an alphabetical listing. Several example programs using the DSPMATH library are also included with the supplied software in the \srdsp\examples directory.

DSPMATH INITIALIZATION

C Usage:

```
#include "dspmath.h"

int init_dspmath( void);
```

Fortran usage:

```
$include: 'dspmath.for'

INTEGER*2 init_dspmath
external init_dspmath
```

Summary:

This function explicitly initializes the DSPMATH library. It can be called to do a presence test for the board and to make sure everything is correctly initialized. If the initialization is successful a 1 will be returned, otherwise a 0 for failure.

It is *not* necessary to call this function before using the library. DSPMATH automatically keeps track of whether the board has been initialized or not, and performs an initialization on the first call to a library function if it has not been done yet.

FORTRAN programmers should note that they *must* declare the return type of the function to be INTEGER*2 in their source for init_dspmath to work correctly.

DSPMATH_status

C Usage:

```
#include "dspmath.h"

extern unsigned DSPMATH_status;
```

Fortran usage:

```
$include: 'dspmath.for'

INTEGER*2 DSPMATH_status[C,EXTERN,ALIAS:'_DSPMATH_status']
```

Summary:

The value of this global variable reflects the status of a calculation just completed in the library. It can be checked to determine if any divide by zero or other errors have occurred. The possible *return* values are:

1	SUCCESS	successful calculation
2	ERROR_INV	divide by zero
3	ERROR_SQRT	square root argument negative
4	ERROR_LOG	log of 0 or negative number

When an error occurs during a calculation, it is *immediately* halted and control is returned to the PC.

FORTRAN programmers should note that this variable is not declared in the include file dspmath.for. They must explicitly give the declaration as above in their source code to have access to this variable.

SETTING WAIT STATES

C Usage:

```
#include "dspmath.h"

void set_waits( unsigned A, unsigned B);
```

Fortran usage:

```
$include: 'dspmath.for'

INTEGER*2 A, B
call set_waits( A, B)
```

Summary:

This function programs the number of wait states the board will use for its memory partitions A and B. By default, the system will run at 2 wait states after power up. If you are running 35ns or faster memories, you can probably run at 1 or 0 wait states. A and B should have the values 0, 1, 2, or 3 for the corresponding number of wait states.

To experiment with the number of wait states your board can run at, you can execute the diagnostic program diag.exe supplied in the examples directory. This program can take command line arguments for the number of wait states used in the diagnostics. When the speed is set too fast for the installed memories, errors will begin to occur. For a more complete description of setting the wait states, refer to the setting wait states topic in the EXAMPLES chapter of this manual.

VECTOR ARITHMETIC

C Usage:

```
#include "dspmath.h"

void add( INPUT x, INPUT y, DIM n, OUTPUT z);
void sub( INPUT x, INPUT y, DIM n, OUTPUT z);
void xpcy( INPUT x, FLOAT c, INPUT y, DIM n, OUTPUT z);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 x(n), y(n), z(n)
INTEGER*4 n
call add( x, y, n, z)
call sub( x, y, n, z)
call xpcy( x, c, y, n, z)
```

Summary:

These functions compute simple vector arithmetic. The C pseudo code for each is as follows:

```
add:            for ( i = 0 ; i < n ; i++ )
                       z[i] = x[i] + y[i];

sub:            for ( i = 0 ; i < n ; i++ )
                       z[i] = x[i] - y[i];

xcpy:           for ( i = 0 ; i < n ; i++ )
                       z[i] = x[i] + c * y[i];
```

In these functions and elsewhere in the library, x, y, and z need not be distinct vectors. The inputs and outputs may point to any sources and destinations desired. For example, the call add(x, x, n, x) would add x to itself and save the result back in x.

VECTOR SCALING AND CONVERSION TO INTS

C Usage:

```
#include "dspmath.h"

void scale( INPUT x, DIM n, FLOAT c, OUTPUT y);
void iscale( INPUT x, DIM n, FLOAT c, INT_ARRAY y, WORD mode);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 x(n), c, y(n)
INTEGER*4 n
INTEGER*2 mode
call scale( x, n, c, y)

REAL*4 x(n), c
INTEGER*4 n
INTEGER*2 y(n), mode
call iscale( x, n, c, y, mode)
```

Summary:

The function scale multiplies each entry of x by c. The function iscale performs the same operation, but the results are returned as 16 bit integers. This can be useful for generating results that are to be plotted. The C pseudo code for each is as follows:

```
scale:          for ( i = 0 ; i < n ; i++ )
                      y[i] = c * x[i];

iscale:         for ( i = 0 ; i < n ; i++ )
                      y[i] = (int)(c * x[i]);
```

The mode variable for iscale selects truncating or rounding. Mode = 0 truncates the result, mode = 1 rounds toward 0.

INNER PRODUCT AND VECTOR MULTIPLY

C Usage:

```
#include "dspmath.h"

void inner( INPUT x, INPUT y, DIM n, OUTPUT result);
void vmult( INPUT x, INPUT y, DIM n, OUTPUT z);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 x(n), y(n), result, z(n)
INTEGER*4 n
call inner( x, y, n, result)
call vmult( x, y, n, z)
```

Summary:

These functions compute the inner product and multiply two vectors entry by entry respectively. The C pseudo code for each is as follows:

```
inner:      for ( i = 0 , result = 0 ; i < n ; i++ )
                  result +=  x[i]*y[i];

vmult:      for ( i = 0 ; i < n ; i++ )
                  z[i] = x[i] * y[i];
```

As with the other library functions, the sources and destinations may point to any inputs and outputs. For example, to compute the inner product of x with itself you could call inner(x, x, n, result);

MATRIX MULTIPLY

C Usage:

```
#include "dspmath.h"

void mmult( INPUT A, INPUT x, DIM n, DIM m, OUTPUT y);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 A(n*n), x(m*n), y(m*n)
INTEGER*4 n, m
call mmult( A, x, n, m, y)
```

Summary:

This function multiplies m vectors into an nxn matrix A. x is the array of vectors to multiply into A, where each vector is of length n and there are m of them. This function is designed to be used for operations like rotations, where many vectors are to be multiplied into the same matrix. A typical call from C would be:

```
#define N       3
#define M       2

float A[N*N] = {
   1.0, 0.0, 0.0,
   0.0, 1.0, 0.0,
   0.0, 0.0, 1.0,
   };

float x[M*N] = {
   1.0, 1.0, 1.0,      /* vector #1 */
   2.0, 2.0, 2.0,      /* vector #2 */
   };

main() { mmult( A, x, N, M, x); }
```

Note that A and x are declared as singly dimensioned arrays and their starting addresses are passed to mmult.

Ax = b SOLVER AND L2

C Usage:

```
#include "dspmath.h"

void gauss( INPUT A, INPUT b, DIM n, OUTPUT LU, OUTPUT x,
                                                    LONG_ARRAY p);
void AtranA( INPUT A, INPUT b, DIM n, DIM m,
                                OUTPUT ATA, OUTPUT ATb);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 A(n*n), b(n), LU(n*n), x(n)
INTEGER*4 n
INTEGER*4 p(n)
call gauss( A, b, n, LU, x, p)

INTEGER*4 NO_PIVOT[C,EXTERN,ALIAS:'_NO_PIVOT']

REAL*4 A(m*n), b(m), ATA(n*n), ATb(n)
INTEGER*4 n, m
call AtranA( A, b, n, m, ATA, ATb)
```

Summary:

These two functions have been designed for solving the linear system Ax = b, and setting up the normal equations for L2 approximation problems.

The function gauss computes the solution x to the problem Ax=b, returning the LU decomposition computed during the gaussian elimination in the matrix pointed to by LU. Row pivoting is optional, and can be turned on and off by passing special values for the pivot array p. To compute the solution with no pivoting, C programmers should pass a NULL pointer for the pivot array. FORTRAN programmers should declare the symbol NO_PIVOT as external in their programs as indicated and pass it for no pivoting. To compute with pivoting, pass an array of n long integers for p.

The function AtranA sets up the normal equations for an L2 approximation. On entry, A and b would be the matrix on the left and data vector on the right. Both are multiplied by A transpose and returned in the arrays pointed to by ATA and ATb on exit.

TRIDIAGONAL MATRIX SOLVER

C Usage:

```
#include "dspmath.h"

void tridiag( INPUT u, INPUT d, INPUT l, INPUT b, DIM n,
              OUTPUT LU_u, OUTPUT LU_d, OUTPUT LU_l, OUTPUT x);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 u(n), d(n), l(n), b(n)
REAL*4 LU_u(n), LU_d(n), LU_l(n), x(n)
INTEGER*4 n
tridiag( u, d, l, b, n, LU_u, LU_d, LU_l, x)
```

Summary:

This function solves the linear system Ax = b where the matrix A is tridiagonal. It is faster and uses less storage than the previous function gauss. The vectors u, d, and l are the upper, diagonal and lower entries of A, while b and x are the input and solution vectors to the system. The u, d, and l vectors each have dimension n, where the last entry of u and first entry of l are not significant for the calculation.

Like the gaussian matrix function, tridiag works by LU decomposition. However unlike gauss, pivoting is not an option.

FOURIER TRANSFORMS

C Usage:

```
#include "dspmath.h"

void fourier( COMPLEX_INPUT A, DIM n, COMPLEX_OUTPUT FFTA,
                                                    int isign);

void rfourier( INPUT x, DIM n, OUTPUT FFTx, int isign);
```

Fortran usage:

```
$include: 'dspmath.for'

COMPLEX*8 A(n), FFTA(n)
INTEGER*4 n
INTEGER*2 isign
call fourier( A, n, FFTA, isign)

REAL*4 x(n), FFTx(n)
INTEGER*4 n
INTEGER*2 isign
call rfourier( x, n, FFTx, isign)
```

Summary:

These functions compute the complex and real fourier transforms of their input using the FFT algorithm. For both functions, *n must be a power of 2*, where *no checking* of the dimension is done on entry. Calling either function with incorrect dimensions will result in erroneous output. The direction of the transform is determined by isign, where it should be +1 for the forward transform and -1 for the inverse. No scaling by n is done on the inverse transforms.

The function fourier computes a full complex transform. Complex input is expected and complex output is returned in the vector pointed to by FFTA. For the function fourier, the minimum dimension is n = 16, while the maximum is n = 32768 or less depending on the amount of memory on the board.

The function rfourier has been optimized to compute real transforms. When isign is +1, it expects real input and returns complex output. Because frequencies above the Nyquist are the complex conjugate of those below for real transforms, only the first half of the complex entries up to the Nyquist are returned. The Nyquist point is packed into the imaginary part of the first entry in the usual way. When isign is -1, the function expects complex input in the same

format as the output for isign +1. Because the input and output may be real or complex depending on isign, they have both been declared as real, and you should typecast them if necessary in your program. Both arrays should be dimensioned to be the same size. For rfourier, the minimum dimension is n = 32, while the maximum dimension is n = 65536 again depending on the amount of installed memory.

For real input the function rfourier is considerably faster than setting up a complex array whose imaginary part is zero and calling fourier. About half of the redundant calculations in the butterfly diagrams have been eliminated, and also only half as much data has to be passed to the board.

As with the other functions in the math library, the source and destination pointers in these functions may point to any inputs and outputs as desired. In the function fourier for example, if you wish to store the output back in the input, call fourier(A, n, A, isign);

For computing spectra, see the next function.

SPECTRUM

C Usage:

```
#include "dspmath.h"

void spectrum( INPUT x, INPUT taper, DIM n, OUTPUT xspectrum);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 x(n), taper(n), xspec(n)
INTEGER*4 n
call spectrum( x, taper, n, xspec)

real*4 NO_TAPER[C,EXTERN,ALIAS:'_NO_TAPER']
```

Summary:

This function computes the spectrum of a real input array. It combines several of the operations usually involved in computing a spectrum into one function.

To prepare the real input for transforming, the function first multiplies each entry of the input series x by the specified real taper. Tapering the input time series is a good idea to avoid ringing at the Nyquist, and also to minimize spectral leakage. From C, a NULL pointer for the taper may be optionally passed to spectrum, where a linear taper will then be applied to the first and last n/4 points of the input series. FORTRAN programmers should declare the symbol NO_TAPER as external in their program and pass it to use the default linear taper. To apply no taper to the input array, pass a taper array with all entries set to 1.

After tapering, the series is then transformed with the rfourier function listed earlier in this chapter. The minimum and maximum dimensions allowed are the same as for the rfourier, namely n = 16 and n = 65536.

Once the series is transformed, the square root of the mod square of each complex entry is computed and the final real result is returned in the vector pointed to by xspectrum.

FFT CONVOLUTION

C Usage:

```
#include "dspmath.h"

convolve( INPUT x1, INPUT x2, INPUT filter, DIM n, OUTPUT y1,
                                                    OUTPUT y2);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 x1(n), x2(n), filter(n), y1(n), y2(n)
call convolve( x1, x2, filter, n, y1, y2)
```

Summary:

This function convolves two real times series with a filter function at once. Each input series is fourier transformed into the frequency domain, and then is multiplied term by term with the transform of the filter function. The filtered data is then transformed back to the time domain and returned in the arrays y1, y2.

To compute the convolution, the two real input series are packed into one complex series internal to the function, and convolved with the filter at the same time. This effectively doubles the performance of the function on real data.

Convolutions can also be computed in the time domain with the following FIR function. It is generally faster to use the FIR function when the length of the filter function in the time domain is short.

FIR FILTER

C Usage:

```
#include "dspmath.h"

void fir( INPUT leader, INPUT x, DIM n, INPUT h, DIM nfilter,
                                                  OUTPUT y);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 leader(nfilter), x(n), h(nfilter), y(n)
INTEGER*4 n, nfilter
call fir( leader, x, n, h, nfilter, y)
```

Summary:

This function computes a convolution in the time domain by applying a FIR filter kernel h to the input vector x. The leader array is used as the leading part of the series x for getting the computation started. The C pseudo code is as follows:

```
fir:     concatenate leader with x

         for ( i = 0 ; i < n ; i++ ) {

                for ( j = 0 , sum = 0 ; j < nfilter ; j++ )
                       sum = sum + h[j] * x[i-j];

                y[i] = sum;
                }
```

where the leader array is used as the leading part of the series x, and should have dimension nfilter. The leader array should typically be filled with zeros on entry unless a long series is being filtered in segments. In that case the leader for each segment is taken from the end of the previous segment.

POLYNOMIAL EVALUATION

C Usage:

```
#include "dspmath.h"

void poly_array( INPUT c, DIM nc, FLOAT xstart, FLOAT delta,
                                                OUTPUT y, DIM n);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 c(nc), xstart, delta, y(n)
INTEGER*4 nc, n
call poly_array( c, nc, xstart, delta, y, n)
```

Summary:

This function evaluates a polynomial with the coefficients c[] at n successive points starting at xstart and spaced delta units apart. The polynomial evaluated is:

```
y = c[0] + c[1]*x + c[2]*x^2 + ... + c[n]*x^n
```

where C array indexing has been used in denoting the entries of c[]. In FORTRAN the equation would read as:

```
y = c(1) + c(2)*x + c(3)*x**2 + ... + c(n+1)*x**n
```

This function is particularly useful for evaluating polynomials resulting from L2 approximations with the AtranA and gauss functions.

VECTOR ARRAY FUNCTIONS

C Usage:

```
#include "dspmath.h"

void inv_array( FLOAT xstart, FLOAT delta, OUTPUT y, DIM n);
void sin_array( FLOAT xstart, FLOAT delta, OUTPUT y, DIM n);
void cos_array( FLOAT xstart, FLOAT delta, OUTPUT y, DIM n);
void sqrt_array( FLOAT xstart, FLOAT delta, OUTPUT y, DIM n);
void log10_array( FLOAT xstart, FLOAT delta, OUTPUT y, DIM n);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 xstart, delta, y(n)
INTEGER*4 n

call inv_array( xstart, delta, y, n)
call sin_array( xstart, delta, y, n)
call cos_array( xstart, delta, y, n)
call sqrt_array( xstart, delta, y, n)
call log10_array( xstart, delta, y, n)
```

Summary:

These functions each evaluate their respective function at n successive points starting at xstart and separated by delta. The C pseudo code is as follows:

```
func_array:     for ( i = 0 , x = start ; i < n ; i++ )
                       { y[i] = func( x); x += delta; }
```

VECTOR FUNCTIONS

C Usage:

```
#include "dspmath.h"

void vinv( INPUT x, DIM n, OUTPUT y);
void vsin( INPUT x, DIM n, OUTPUT y);
void vcos( INPUT x, DIM n, OUTPUT y);
void vsqrt( INPUT x, DIM n, OUTPUT y);
void vlog10( INPUT x, DIM n, OUTPUT y);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 x(n), y(n)
INTEGER*4 n

call vinv( x, n, y)
call vsin( x, n, y)
call vcos( x, n, y)
call vsqrt( x, n, y)
call vlog10( x, n, y)
```

Summary:

These functions evaluate their respective function on *each* entry of their input array. The C pseudo code is as follows:

```
vfunc:      for ( i = 0 ; i < n ; i++ )
                  y[i] = func( x[i]);
```

POLAR TO CARTESIAN CONVERSION

C Usage:

```
#include "dspmath.h"

void polar_to_xy( INPUT rtheta, DIM n, OUTPUT xy);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 rtheta(2*n), xy(2*n)
INTEGER*4 n
call polar_to_xy( rtheta, n, xy)
```

Summary:

This function converts the polar coordinates stored in the array rtheta into cartesian coordinates. The coordinates should be stored as radius angle pairs in the input array and will be returned as xy pairs in the output array. The C pseudo code is as follows:

```
polar_to_xy:    for ( i = 0 ; i < n ; i++ ) {

                        *xy++ = rtheata[0] * cos( rtheta[1]);
                        *xy++ = rtheata[0] * sin( rtheta[1]);

                        rtheata += 2;
                        }
```

LINEAR INTERPOLATION

C Usage:

```
#include "dspmath.h"

void interpolate( INPUT array1, DIM n1, OUTPUT array2, DIM n2);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 array1(n1), array2(n2)
INTEGER*4 n1, n2
call interpolate( array1, n1, array2, n2)
```

Summary:

This function linearly interpolates array1 into array2. The dimension n2 may be either greater than or less than n1 for interpolation or decimation respectively.

SPLINE INTERPOLATION

C Usage:

```
#include "dspmath.h"

void spline( INPUT x, INPUT y,
          OUTPUT h, OUTPUT a, OUTPUT b, OUTPUT c, OUTPUT d,
              OUTPUT xx, OUTPUT yy, DIM n, DIM nn, FLOAT xdiv);
```

Fortran usage:

```
$include: 'dspmath.for'

REAL*4 x(n), y(n), h(n), a(n), b(n), c(n), d(n)
REAL*4 xx(nn), yy(nn), xdiv
INTEGER*4 n, nn
call spline( x, y, h, a, b, c, d, xx, yy, n, nn, xdiv)
```

Summary:

This function computes both the spline coefficients and the spline interpolate. The input arrays x and y have the function y(x) at n points for which the spline approximation is to be computed. On output, the arrays a, b, c, and d have the coefficients of the spline approximation on each interval, where the approximating polynomial is:

$$y(x) = a*x^3 + b*x^2 + c*x + d$$

The array h has the distance between successive x points. xx and yy are the interpolated values which are interpolated at nn points separated a distance xdiv apart. The points on input need not be equally spaced, however the interpolated points are always equally spaced xdiv units apart.

GRAPHICS LIBRARY

INTRODUCTION TO THE GRAPHICS LIBRARY

The GRAPHICS library is a collection of functions for displaying the results of numerical calculations on the Hercules and EGA/VGA/SuperVGA graphics adapters. The library runs stand alone on the PC, and does not require a DSP32C board to be installed for its operation. A number of video modes for each adapter are supported and, in addition to graphics, keyboard and mouse support for menu selections is also provided.

As with the other libraries included with the system, the compiler used to build the library is indicated on the disk label, and batch files have been included to do a complete rebuild if needed. Note that usually you will not be successful in linking Microsoft and Borland object and library files together, and you must be sure to consistently use the same compiler tools.

To begin using the GRAPHICS library in your programs, you should do the following two things:

- *Always* include graphics.h in your source code files using the functions
- Link with the library in the appropriate memory model like:

  > link yourprogram,,,cs_graph.lib

It is important to *always* include graphics.h in your source code files using the GRAPHICS library. This file has C prototypes for all the functions and guarantees you are calling them correctly. Several of the functions use far and huge pointers, and the protos will automatically force any coercions of near pointers to far or huge that are required. Calling a function with the incorrect pointer type will often crash the PC, and including graphics.h avoids this.

To make the GRAPHICS library easy to use, you should also include the directories \srdsp\include and \srdsp\lib on the paths your compiler automatically searches for include and library files. This makes compiling and linking easy to do and, as explained in the INSTALLATION chapter, *avoids* creating multiple copies of the system files on your hard disk. Having only a single copy of the system software on your disk makes maintaining your software and installing future upgrades much easier.

The GRAPHICS library is supplied in four memory models for the PC. They are small, medium, compact, and large. Each file is called cx_graph.lib, where x indicates the model:

cs_graph.lib	small model
cm_graph.lib	medium model
cc_graph.lib	compact model
cl_graph.lib	large model

Programmers using huge model should link to the large model library, cl_graph.lib.

It is important to link to the correct library for the memory model you are using. Linking to the incorrect model, or to a library built with the wrong compiler, will usually fail with either a compiler, linker, or runtime error. Note that the same include file, graphics.h is used regardless of the memory model you are using.

Within your code, you must *always* call the function set_graphics to initialize the library and graphics adapter. Calling set_graphics selects the device drivers and sets the global video parameters for the specified video mode. Set_graphics *must* be called before any other functions in the library are used.

A variety of video modes are supported, from the 720x348 Hercules monochrome to the Super VGA 1024x768 256 color. However, note that the Super VGA modes are limited to the Tseng ET-3000 and ET-4000 graphics chip sets. The Super VGA is not yet standardized, and a variety of paging modes are being used by various manufacturers for frame buffers larger than 64K bytes. The Tseng Labs chips have fairly typical of paging schemes, and are used on boards by Orchid, STB, and Tseng Labs. It should be possible to make modifications to the graphics library source code to support the paging modes of other Super VGA chip sets. See the source code in the graphics subdirectory for more details.

With only a few exceptions, all the functions in the library will work in any graphics mode. If a function specific to a particular mode is called in a different mode then nothing will happen. An example would be calling twiddle_palette to set the colors when Hercules monochrome video has been selected. This makes it easy to port an application across video modes, where inappropriate functionality will drop out.

Finally, pixel graphics coordinates are used by all the functions to locate objects. The origin is always in the *upper left hand corner* of the screen, where the i coordinate runs *down* the screen, and the j coordinate runs *across* the screen. This coordinate system is used consistently throughout the library. The screen dimensions for any selected video mode are available in the global video_xxx parameters after a call to set_graphics. To write applications that are portable across a variety of video modes, you must use these parameters to scale your screen coordinates appropriately.

The following pages describe each GRAPHICS function, and are organized by functional area. Refer to the index for an alphabetical listing. Several example programs using the GRAPHICS library are also included with the supplied software in the \srdsp\graphics subdirectory.

INITIALIZATION AND VIDEO MODE

C Usage:

```
#include "graphics.h"

int set_graphics( int library_mode);
void set_text( void)
```

Summary:

These functions initialize the GRAPHICS library for a particular video mode, and return the system to normal DOS text mode respectively. They should be called at the beginning and end of your programs to properly use the graphics library.

Calling set_graphics at the beginning of your programs selects the drivers the graphics library will use and calls BIOS to initialize the graphics adapter board and mouse. Set_graphics *must* be called to initialize the GRAPHICS library.

The following modes are supported by the library:

LIBRARY MODE	BIOS MODE	DESCRIPTION	RES	NOTES
1	NA	HERCULES	720x348	
2	0x10	VGA 16 color	640x350	
3	0x12	VGA 16 color	640x480	
4	0x29	VGA 16 color	800x600	
5	0x37	VGA 16 color	1024x768	Tseng only
6	0x13	VGA 256 color	320x200	
7	0x2D	VGA 256 color	640x350	Tseng only
8	0x2E	VGA 256 color	640x480	Tseng only
9	0x30	VGA 256 color	800x600	Tseng only
10	0x38	VGA 256 color	1024x768	Tseng only

You should use one of the *library mode* numbers with set_graphics to specify the desired video mode. Set_graphics will automatically map the library mode into the listed default BIOS mode when initializing the graphics board. To override the BIOS default value used, you can set the value of the global video parameter video_bios *before* calling set_graphics. Any nonzero value of video_bios will override the default. See the description of the global

video parameters for an example of doing this. To refer mnemonically to the library modes, use the defined constants in graphics.h.

For the Super VGA modes, the library assumes a Tseng Labs ET-4000 chip set is being used by default. Like the video_bios value, this can be overridden by setting the global parameter video_chip *before* calling set_graphics. Use the defined constants in graphics.h for setting video_chip to an ET-3000.

If the requested video mode is supported by the library, a 1 will be returned, otherwise a 0. Set_graphics does not do a presence test for any particular graphics adapter. It is assumed that an appropriate board is installed in the system for the requested video mode.

GLOBAL VIDEO PARAMETERS

C Usage:

```
#include "graphics.h"

extern unsigned
   video_mode,
   video_bios,
   video_chip,
   video_colors,
   video_hpixels,
   video_vpixels,
   video_segment,
   video_hbytes;

extern unsigned long
   video_total;

extern int mouse_installed;

extern char far *char_generator;
```

Summary:

These parameters describe the current video state. Their values are set when set_graphics is called to initialize the graphics library. Most should only be read by your applications to get information about screen size and number of colors etc. Changing their values other than by calling set_graphics will cause some of the library functions to fail. They are declared extern in the header file graphics.h so you won't have to redeclare them in your source code.

Two of the parameters, video_bios and video_chip can be set to nonstandard values *before* calling set_graphics to override the normal default values used by set_graphics. A typical program fragment might be:

```
video_bios = 0x6A; set_graphics( 4);
```

where a BIOS mode value of 0x6A is used by many graphics boards to specify Super VGA 16 color 800x600 graphics.

In addition to setting up the graphics board, set_graphics also tests for the presence of a mouse. If the mouse can be successfully initialized, the mouse_installed parameter is set to 1, otherwise a 0.

Finally, the far pointer char_generator is also initialized by set_graphics to point to the character bitmap source used by the library. The set_graphics default points to the BIOS ROM on the PC's system board. This is one of the most reliable places. However, you can change the pointer *after* calling set_graphics to point to any other source you like. See the characters and strings functions for more details.

PALETTE SETTING FOR 16 and 256 COLOR MODES

C Usage:

```
#include "graphics.h"

void set_palette( char far *palette);
```

Availability:

This function is applicable only to the EGA/VGA/SuperVGA 16 and 256 color modes. Calling it in the Hercules monochrome mode will simply return with no action taken.

Summary:

This function programs the color registers on the graphics adapter. These registers map the value of a particular pixel into the final color that is sent to the video monitor. You *do not* have to call this function as part of your initialization code. After calling set_graphics, BIOS will have automatically programmed the graphics adapter with a default palette. However, if you wish to use a non standard palette then you'll need this function.

When calling the function in a 16 color mode, the palette should be declared as an array of 16 bytes.

```
char palette[16];
```

These bytes have the RGB bits for each of the 16 possible values a pixel could have. The bits for each palette byte are laid out as follows:

bit 0	B0 blue
bit 1	G0 green
bit 2	R0 red
bit 3	B1 blue
bit 4	G1 green
bit 5	R1 red
bit 6	not used
bit 7	not used

where two bits are assigned to each color, allowing for one of four intensity levels on each of R, G, and B.

For interactively setting the palette in a 16 color mode see the next function, twiddle_palette, which provides a menu interface for selecting the colors.

When calling the function in a 256 color mode, there are 256 DAC palette registers. Each register is comprised of 3 bytes with values for the red, green, and blue colors. On most boards, only the low 6 bits of each color byte are used, with 0 as the lowest intensity and all bits set as the highest. The palette should be an array declared as:

```
char palette[3*256];
```

or as:

```
struct DAC_register { char red, green, blue; } palette[256];
```

Where with the second form of declaration, you'll have to explicitly coerce the palette array to char far * in your code when passing its address to set_palette.

INTERACTIVE PALETTE SELECTION FOR 16 COLOR MODES

C Usage:

```
#include "graphics.h"

void twiddle_palette( int iorg, int jorg, char far *palette);
```

Availability:

This function is applicable only to the EGA/VGA/SuperVGA 16 color modes. Calling it in the Hercules or 256 color modes will simply return with no action taken.

Summary:

This function allows you to interactively select a 16 color palette. The function will display a color bar containing 16 entries on the screen with its upper left hand corner located at (iorg, jorg).

The palette entries are displayed as small color squares with the number of the palette entry selected for modification displayed directly above the respective color square. The right and left arrow keys move to the neighboring entry, while the amount of color in an entry is selected by the up and down arrow keys. Whether red, green, or blue is being modified is indicated by the RGB letter on the right, and can be selected by the keys r, g, or b on the keyboard.

To exit the interactive selection of the palette, press the enter key on the keyboard.

SCREEN CLEARING AND TITLES

C Usage:

```
#include "graphics.h"

void turnon( int color);
void border( int color);

void header( char *left, char *right,
                              int fore, int back, int bord);
```

Summary:

These simple functions are usually called immediately after set_graphics to clear the screen and put some titles on it.

Turnon clears the entire screen to the specified color. Clearing to a non zero value is useful when you want to have a colored background.

Border draws a rectangle around the very edge of the screen, giving the user a feel for the size of the frame buffer. The function automatically sizes itself for the current video mode.

Header puts some titles at the top of the screen, automatically positioning one at the left and one at the right. Fore, back, and bord specify the colors for drawing the text and border around the titles.

DOTS AND CURSORS

C Usage:

```
#include "graphics.h"

void dot( int i, int j, int color);
void cursor( int i, int j, int color);
void crosshair( int i, int j, int color);
```

Summary:

The function dot draws a single pixel on the screen in the specified color. The coordinates are absolute pixel coordinates with (0, 0) in the upper left hand corner of the screen.

Cursor and crosshair draw their respective objects at the pixel location (i,j). The tip of the cursor arrowhead and the center of the crosshairs are located exactly on the (i,j)th pixel. The cursor is always drawn in XOR mode so it can be easily moved and erased on the screen. See the function move_mouse for a routine that will automatically move the cursor around. The crosshair is simply drawn to the screen and not XORed.

Generally drawing single dots isn't too useful, and is very slow when done repeatedly to draw more complicated objects. If you can, use one of the higher level functions to draw more complicated objects like lines and rectanges. They are optimized for their respective operations.

LINE DRAWING

C Usage:

```
#include "graphics.h"

void hline( int i1, int j1, int j2, int color);
void vline( int i1, int j1, int i2, int color);

void line( int i1, int j1, int i2, int j2, int color);
```

Summary:

These functions draw lines starting at the pixel location (i1, j1) with the specified color.

Hline and vline have been optimized to draw horizontal and vertical lines as fast as possible, and are considerably faster than line.

Line draws a Bresenham line connecting any two points. For plotting time series, don't make repeated calls to the line function. Instead, use the plot function to draw y(x) plots fast on the screen.

RECTANGLES

C Usage:

```
#include "graphics.h"

void rectangle( int i1, int j1, int i2, int j2, int color);
void fill( int i1, int j1, int i2, int j2, int color);

void rect8x8( int i, int j8x8, int color);
```

Summary:

These functions draw rectangles where (i1, j1) and (i2, j2) are any two opposing corners. Rectangle draws only the boundary lines around the edge of the area, while fill draws a filled rectangle.

Rect8x8 is optimized specifically for drawing filled 8x8 pixel rectangles in the Hercules and 16 color modes. The coordinates specify the upper left hand corner of the rectangle, and the j coordinate has the 8x8 suffix to denote that it will be byte aligned in the horizontal direction. This function is not available in the 256 color modes because 8x8 rectangle drawing can't be optimized for their frame buffer layout. To draw small rectangles in the 256 color modes, call rectangle and fill.

AXES AND GRIDS

C Usage:

```
#include "graphics.h"

void haxis( int i1, int j1, int j2, int dj, int color);
void vaxis( int i1, int j1, int i2, int di, int color);

void grid( int i1, int j1, int i2, int j2, int di, int dj,
                                                    int color);
```

Summary:

These functions draw horizontal and vertical axes with regularly spaced tick marks and a complete grid.

Haxis and vaxis draw horizontal and vertical axes starting at (i1, j1) with tick marks every dj or di pixels.

Grid covers the rectangle defined by (i1, j1) and (i2, j2) as its opposing corners with grid lines every dj and di pixels.

CHARACTERS AND STRINGS

C Usage:

```
#include "graphics.h"

extern char far *char_generator;

void bltchar( int i, int j, int c, int fore, int back);
void gputs( int i, int j, char *s, int max, int fore, int back);
void ggets( int i, int j, char *s, int max, int fore, int back);
```

Summary:

These functions draw text characters on the graphics screen. Bltchar draws the single character c where (i,j) is the pixel coordinate of the upper left hand corner of the character cell.

Gputs and ggets move entire NULL terminated strings to and from the graphics screen. At most max number of characters are moved, and tabs are correctly expanded by ggets. Also, ggets will echo characters from the keyboard to the screen until enter is hit. Fore and back specify the colors for drawing the letters.

The source for the characters used by these routines is taken from the bitmap pointed to by char_generator. The character generator should be a bitmap of 8x8 pixel characters, where each character occupies 8 bytes. Set_graphics initializes the char_generator pointer to point to the system ROM. You can change it on the fly to point to custom character bitmaps. This is a good way to display specialized symbols on maps and other types of plot.

1D FUNCTION AND TIME SERIES PLOTTING

C Usage:

```
#include "graphics.h"

void plot( int i, int j, int far *array, int n, int color);
```

Summary:

This function is for plotting one dimensional arrays as y(x) plots horizontally on the screen. The function starts at pixel location (i,j) and moves one pixel to the right for each value in array as it plots. The array should be declared as int array[n] in the user's program. No checking is done for array values that are off the screen, and they will be incorrectly plotted.

This function has been optimized to plot y(x) functions as fast as possible. No checking is done on the number or size of the array entries.

2D ARRAY PLOTTING AS COLOR SQUARES

C Usage:

```
#include "graphics.h"

void blt8x8( char far *array, int ni, int nj, int i, int j8x8);
void bltdot( char huge *array, int ni, int nj, int i, int j);
```

Availability:

The function blt8x8 is applicable to the EGA/VGA/SuperVGA 16 color modes, while bltdot is available in both 16 and 256 color modes. Calling either in the Hercules monochrome mode will simply return with no action taken.

Summary:

These functions are for drawing 2d arrays as color squares on the screen. They are useful for displaying images.

Blt8x8 draws a 2d array as an image of 8x8 pixel rectangles in the 16 color modes. It is appropriate when there is a small amount of data with limited dynamic range to present. The array should be declared in your program as char array[ni*nj]. (i,j8x8) are the pixel coordinates of the upper left hand corner of the displayed area, where the 8x8 suffix on the j coordinate denotes bytes alignment in the horizontal direction. Blt8x8 is *not* available in the 256 color modes.

Bltdot draws a 2d array as an image of single pixels in both 16 and 256 color modes. The array values should be declared as char array[ni*nj], and is passed as a huge pointer to denote that the function can take input arrays larger than 64K. Huge arrays are required when covering most of the screen in the high res 256 color modes.

KEYBOARD INPUT

C Usage:

```
#include "graphics.h"

int keyin( void);
int keystat( void);
```

Summary:

These functions return keyboard input by making calls to BIOS interrupt 16H. They fetch single keystrokes. For inputting an entire string, you should use the function ggets.

Keystat returns the status of the keyboard. If no key has been pressed it returns 0, otherwise it returns the extended ASCII code of the pressed key.

Keyin waits until a keyboard key has been pressed and returns the extended ASCII code of that key.

The extended key code contains the ASCII character in the low byte and the scan code for the key in the high byte. Keys without ASCII codes, such as the function keys, return zero in the low byte and their scan code in the high byte. The scan code encodes the position of the key on the keyboard so that keys without ASCII codes can be detected. Extended keycode constants are defined in graphics.h for several keys like up and down arrow.

MOUSE

C Usage:

```
#include "graphics.h"

extern int mouse_installed;

typedef struct {
   int i, j;                  /* mouse position */

   int color;                 /* mouse parameters */
   int iinc, jinc,
   int shift_iinc, shift_jinc,
   int mini, minj, maxi, maxj;

   int left, right;           /* additional returned data */
   int key;

   }; MOUSE

int mouse_init( void);

void mouse( int far *i, int far *j);
int mouse_button( void);

void move_mouse( MOUSE_DATA *m);
void mouse_wait( void);
```

Summary:

These functions provide support for a Microsoft or compatible mouse. When the graphics library is initialized with a call to set_graphics, a presence test and initialization of the mouse is automatically done, and the global mouse_installed variable is set to reflect its availability. The mouse_installed variable is declared external in graphics.h, and you don't need to redeclare it in your program. Also, while the mouse_init function can be called to explicitly initialize the mouse, most mouse BIOS's also change the video graphics mode, and it is generally better to let set_graphics do the initialization.

After the mouse is installed, you can get its current coordinates by calling the function mouse. The current mouse coordinates will be returned in the integers pointed by i and j. The mouse button status can be gotten by calling mouse_button. This function returns the current status

of the left and right buttons in the two low bits of its return value. Bits 0 and 1 will be set if the left or right mouse buttons respectively are pressed.

Rather than calling the mouse and mouse_button functions to maintain the mouse, the function move_mouse can usually be called to take care of most of the mouse requirements. To use it, first declare a mouse data structure in your program. A typedef for this structure is in graphics.h and is listed above. Before calling move_mouse, fill in this structure with the initial values your wish to use for the mouse, such as the initial position and color. A cursor will then be moved around on the screen in response to mouse movements until a keyboard key or mouse button is pressed. After returning, the mouse data structure will be filled in with the current mouse data. The variables left and right reflect the button status, while the key variable will have the extended ASCII scan code if a keypress occurred.

The function mouse_wait waits until the user releases the mouse buttons. This can be useful when move mouse is buried inside a control loop, and you don't want to continue on until a button has been released.

MENUS

C Usage:

```
#include "graphics.h"

typedef struct
   int tag;            /* menu item identifier tag */
   int key;            /* menu item keyboard key */
   int i, j;           /* button coordinates */
   int frame;          /* button frame color */
   int on, off;        /* button on off colors */
   int char *s;        /* menu string */

   } MENU_DATA;

void draw_button( MENU_DATA *item, int state);

MENU_DATA *selection( MOUSE_DATA *m, MENU_DATA *menu);
```

Summary:

These functions provide support for graphical menus.

The function draw_button draws a menu item selection button on the screen in either an on or off state.

The function selection determines if the mouse coordinates i and j are in any of the menu areas on the current menu list. If a menu item is selected, a pointer to it is returned, otherwise NULL is returned.

To use these functions, you should set up an array of menu data items in your program with the data for each menu item. The array should be terminated with an entry whose values are all NULL in order for the selection function to work correctly. A typedef for the menu item data structure is in graphics.h and listed above. For an example of using the menu functions refer to the example program life.c in the graphics subdirectory.

EXAMPLES

This chapter covers several example programs for the DSP32C boards. The examples are designed to get you started, and progress through some of the DSP32C features. In addition to the material covered here, there are also programs in the examples subdirectory on the supplied disk that should be referred to.

ADDING TWO NUMBERS IN DSPMON

This example shows how to write and run a program to add two integers in the DSP32C's CAU. In addition to the listing here, the program can also be found in the file add.dsp in the \srdsp\examples subdirectory in the supplied software. The DSPASM assembler code for the program is:

```
constant PROGRAM = 0, DATA = 100;

<DATA> long sum, x = 2, y = 3;

<PROGRAM> main {

   /* Add x and y, saving the result in sum. */
   r1 = *x; r2 = *y; r1 = r1 + r2; *sum = r1;

   /* Wait for the PC to get the result. */
   wait: goto wait;
}
```

The constants PROGRAM and DATA define the DSP32C memory locations to be used with the location directive <>. Note that the program is located at 0 so that the DSP32C will have something to execute after the reset that occurs with the DSPMON go command.

To assemble the source file, execute dspasm add.dsp from the DOS command line. This will create the output file add.dso, where if any errors occur assembly will stop and an error message will be printed to the screen. After that, the program add.dso should be downloaded and run in the DSPMON monitor. From the DOS command line, execute dspmon /50 add.dso or dspwin add. The monitor will be started, and add.dso automatically downloaded. Once started, the monitor will display its prompt and be ready for commands. If you like, you can also have the DSPASM assembler emit a symbol table add.dss, and specify it on the command line for symbolic debugging. The following is a typical session:

```
C:\DSP\EXAMPLES>dspmon add.dso

|u 0
      000000: 1CC10104 r1e = *0x0104
      000004: 00000000 nop
      000008: 1CC20108 r2e = *0x0108
      00000C: 00000000 nop
      000010: 98010040 r1e = r1 + r2
      000014: 1DC10100 *0x0100 = r1e
      000018: 00000000 nop
```

```
      00001C: A000001C goto 0x00001C
      000020: 00000000 nop

|dl 100

000100 LONG  00000000 00000002 00000003 FFEFFFFF

|g
|dl 100

000100 LONG  00000005 00000002 00000003 FFEFFFFF

|q
```

Reviewing the session, dspmon was started with add.dso specified as a file to download. The first command was to disassemble the program. The disassembly listing shows the current DSP32C memory location, the hex value at that address, and the AT&T mnemonic for the instruction. You can see that the source program add.dsp has been downloaded, and that nops have been added by DSPASM to avoid any CAU latency problems.

As programmed, the sum of x and y will be saved in the memory location 100, so the display command is used to look at that location before and after running the program. If you are using the .dss symbol table, the command could also have been issued as dl sum. After the program has run the value is a 5 as expected. Note that it was not necessary to halt the processor with the h command in order to look at the memory location. The PC can look at the DSP32C memory while it is executing.

Finally, the program was terminated with an infinite loop. This gives the DSP32C something to do while it waits for the PC to come and get the result. If you fail to end the program in some predictable way, the DSP32C will continue executing garbage instructions off the end of the program and may start overwriting memory locations containing your results.

MATRIX MULTIPLICATION UNDER PC CONTROL

The previous program was run by hand in the DSPMON monitor. While useful for stand alone applications and testing, programs often need to run under the control of the PC. This example shows how to do that for a simple matrix multiplication program. The scheme is basically the same as in the previous example, except that functions from the DSPTOOLS library rather than the monitor are used to control the DSP32C.

The goal of the program is to multiply a 3x1 vector into a 3x3 matrix, where the vector and matrix are specified in the C portion of a program running on the PC. The C and DSPASM portions of the code can be found in the files mmult.c and mmult.dsp in the examples subdirectory on the supplied disk. The C portion of the code is as follows:

```
#include "dsptools.h"

#define INPUT          0x000100L
struct input { float A[3][3], x[3]; } in;

#define OUTPUT         0x000200L
float y[3];

main() {

   /* Initialize and download the DSP32C. */
   if ( !init_32C() || !sr_load( "mmult.dso", NO_REPORT) ) {
         printf( "ERROR: unable to init DSP32C");
         exit( 1);
         }

   /* Initialize the vector x and matix A. */
   in.A[0][0] = 1.0; in.A[0][1] = 1.0; in.A[0][2] = 1.0;
   in.A[1][0] = 2.0; in.A[1][1] = 2.0; in.A[1][2] = 1.0;
   in.A[2][0] = 1.0; in.A[2][1] = 1.2; in.A[2][2] = 1.4;

   in.x[0] = 1.0; in.x[1] = 2.0; in.x[2] = 3.0;

   /* Download the input data. */
   put_32C( &in, INPUT, sizeof( in));

   /* Execute the DSP32C, computing y = A*x. */
   go(); while( !pir() ); halt();

   /* Read the result back. */
```

```
   get_32C( OUTPUT, y, sizeof( y));

   /* Print out the input and output. */
   ...
}
```

There are several important things to note about this part of the program. First, the constants `INPUT` and `OUTPUT` define the DSP32C memory space addresses where the input and output data can be found. These constants will also be defined in the DSPASM portion of the program, and *must* agree between the two files. Secondly, the layout of the input data is defined in a structure. Declaring the layout in a structure forces the C compiler to follow the specified alignment of data, again maintaining agreement with the DSPASM portion of the program. Arrays declared sequentially as global data in a C program do not necessarily follow any specified alignment.

In addition to data locations, the communication protocol used with the DSP32C board is important. The functions put_32C and get_32C are used to put the input data on the board and to get the output, and the function go is used to start execution. The first thing the DSP32C will do when starting is to set the pir register to a zero. This lets the PC know that the DSP32C is busy on a calculation. When the DSP32C is done, the pir register will be set to a 1. With this scheme, the C program can poll the pir register to know when the DSP32C is done. After the processor is done, it is halted to leave it in its power down state, and the C program prints out the results.

This sequence of programming steps is typical for PC applications controlling the DSP32C boards. Download the code, set the input data, start execution, poll the DSP32C to see if done, halt, and get the results back. Of course, there are variations on this basic control flow that can be useful. An example would be that the PC could be off doing other tasks, perhaps plotting graphics or saving a result to disk, rather than doing nothing in a polling loop.

Finally, the DSP32C portion of the program reads as follows:

```
constant
   RESET         = 0x000000,
   INPUT         = 0x000100,
   OUTPUT        = 0x000200,
   BUSY          = 0,
   DONE          = 1;

<INPUT> ieee A[9], x[3]; <OUTPUT> ieee y[3];

<RESET> main {
```

```
    /* Set pir to BUSY. */
    r1 = BUSY; pir = r1;

    /* Convert A and x to DSP32C floats. */
    r1 = A;
        do 0C { *r1++ = a0 = dsp( *r1); }

    /* Compute y = A*x, using in line code. */
    r1 = A; r3 = y;

    r2 = x;         /* first row */
                 a0 =          *r1++ * *r2++;
                 a0 = a0 +     *r1++ * *r2++;
       *r3++ =   a0 = a0 +     *r1++ * *r2++;

    r2 = x;       /* second row */
                 a0 =          *r1++ * *r2++;
                 a0 = a0 +     *r1++ * *r2++;
       *r3++ =   a0 = a0 +     *r1++ * *r2++;

    r2 = x;       /* third row */
                 a0 =          *r1++ * *r2++;
                 a0 = a0 +     *r1++ * *r2++;
       *r3++ =   a0 = a0 +     *r1++ * *r2++;

    /* Convert y back to IEEE floats. */
    r1 = y;
        do 3 { *r1++ = a0 = ieee( *r1); }

    /* Set the pir register to DONE, and wait for the PC. */
    r1 = DONE; pir = r1; wait: goto wait;
}
```

Note that the location and layout of the data declared here agrees with that of the C portion of the program, where DSPASM guarantees alignment of all global data objects to 4 byte boundaries.

Finally, the floating point numbers coming from the PC are in IEEE format. This input is converted to the internal floating point format of the DSP32C in the do loop at the beginning, and the computed results are converted back to IEEE format in the do loop at the end. It is your responsibility to make sure that floating point numbers are always have the proper binary format for the operations being performed on the DSP32C.

SYMBOL TABLES AND FORWARD REFERENCING

The previous example showed how to control the DSP32C under programmed PC control. It was complicated by having to make sure the constants `INPUT` and `OUTPUT` agreed in both the C and DSPASM portions of the source code. In addition, the DSPASM source was written to be assembled in one pass, forcing the use of location directives. This example shows how to make maintaining the previous example easier with the use of symbol tables and forward referencing.

Instead of maintaining common symbols by hand, the DSPASM assembler can be made to emit its symbol table in C include file format. When included in your C programs, these symbols can then automatically provide your C programs with the DSP32C program locations you need for downloading and exchanging data. Additionally, DSPASM can be run in two pass mode to resolve all forward references, allowing you to write DSP32C source code that reads more naturally.

For this example, suppose you wanted to xor two integers from the PC on the DSP32C board and print the result out on the screen. The DSP32C portion of the program might look like this:

```
xor {

   /* XOR x and y. */
   r1 = *x; r2 = *y; r3 = r1^r2; *z = r3;

   /* Set pir to done, and wait for the PC. */
   r1 = 1; pir = r1; wait: goto wait;
}

long x, y, z;
```

Assuming this has been saved in the file xor.dsp, it should be assembled with the DSPASM /sc option to emit the symbol table to the C include file xor.dsc, and the /2 option to force two passes. The .dsc include file will communicate the addresses of x, y, and z to the C portion of the program, while two passes allow the forward referencing to x, y, and z. Also note that because the function xor is listed first in the source it will be at location 0 by default, and executed on reset.

The C include file xor.dsc created with the /sc option will read as follows:

```
/* global symbol values from xor.dsp */

#define DSP_wait              0x00000028L   /* GOTO */
#define DSP_x                 0x00000030L   /* LONG */
```

```
#define DSP_y                   0x00000034L   /* LONG */
#define DSP_z                   0x00000038L   /* LONG */
#define DSP_xor                 0x00000000L   /* FUNC */
```

Each symbol from the file xor.dsp is listed along with its value. All the symbol names are given a DSP_ prefix to avoid collision with symbol names that are likely to be used in the C portion of the program. Given this file, the C part of the program would look like:

```
#include "dsptools.h"
#include "xor.dsc"

main() {

   long x, y, z;

   /* Download and initialize the DSP32C board. */
   if ( !init_32C() || !sr_load( "xor.dso", NO_REPORT) ) {
          printf( "ERROR: unable to init or download");
          exit( 1);
          }

   /* Initialize x and y, and execute the DSP32C program. */
   x = 0xFFFF;
   y = 0x1111;

   put_32C( &x, DSP_x, sizeof( x));
   put_32C( &y, DSP_y, sizeof( y));

   set_pir( 0); go(); while ( !pir() ); halt();

          get_32C( DSP_z, &z, sizeof( z));

   printf( "z (should be 0xEEEE) = 0x%04lX\n", z);
}
```

While this program demonstrates one way to communicate between the PC and DSP32C board, it is not the last word! Rather than using global data in the DSP32C portion of the code, structures could have been used. A rewrite of the above to use structures in communicating would read as:

DSPASM portion:

```
xor {
   struct input_data { long x, y, z; } input;

   /* XOR two integers together. */
   r1 = *input.x; r2 = *input.y; r3 = r1^r2; *input.z = r3;

   /* Wait for the PC. */
   r1 = 1; pir = r1; wait: goto wait;
}

long input_data[3];
```

C portion:

```
#include "dsptools.h"
#include "xor.dsc"

main() {

   struct input_data { long x, y, z; } input;

   /* Download and initialize the DSP32C board. */
   if ( !init_32C() || !sr_load( "xor.dso", NO_REPORT) ) {
          printf( "ERROR: unable to init or download");
          exit( 1);
          }

   /* Initialize x and y, and execute the DSP32C program. */
   input.x = 0xFFFF;
   input.y = 0x1111;

   put_32C( &input, DSP_input_data, sizeof( input));

   set_pir( 0); go(); while ( !pir() ); halt();

          get_32C( DSP_input_data, &input, sizeof( input));

   printf( "z (should be 0xEEEE) = 0x%04lX\n", input.z);
}
```

The advantage of using structures in the DSPASM portion of the code is that they allow access to a single area of memory in a variety of ways. DSPASM structures are local to the

function that they are declared in and their symbol names can be reused again in other functions. This technique is used extensively in the DSPMATH library.

SUBROUTINE CALL STACKS

The DSP32C architecture differs from that of many processors in that it does not have hardware support for a subroutine call stack. When a subroutine call is made, the return address is stored in a register rather than automatically pushed onto a stack. Storing the return address in a register is fine until nested subroutine calls are made, because a nested call will overwrite a previous return address and the thread of execution will be lost. The way around this problem is to maintain a return address stack in software.

One way to maintain a stack is to dedicate a CAU register to be a stack pointer, and on entry and exit to each subroutine push and pop the return address. With the auto increment and decrement of the DSP32C this is easy to implement. The usual convention on the DSP32C is for r14 to be used as the return address register in the call, and for r13 to be used as a stack pointer. In the AT&T assembler, the return address register can be specified in the call instruction. In DSPASM this is automatically taken to be r14, and does not need to be specified in the call and return instructions. DSPASM does not make any assumptions about which register is used for a stack pointer, but r13 is the usual choice.

In the example program here, the main function makes a call to a function transform, which subsequently calls the functions add and scale. The return address in r14 is saved and restored on entry and exit to the transform function so the thread of execution from main is not lost. The functions add and scale don't have to save and restore r14 because they don't make any subsequent calls. The program reads as follows:

```
constant MAIN = 0, DATA = 100, FUNCTIONS = 150;

<DATA>
dspf x[] = { 1.0, 2.0, 3.0, 4.0 }, A = 2.0, B = 1.0;

<FUNCTIONS>
scale {

   /* Scale each entry of x by A. */
   r1 = x;
   r2 = x;
   r3 = A;

       do 4 { *r1++ = a0 = *r2++ * *r3; }

   /* Wait for DAU pipeline to flush, and return. */
   nop; nop; nop; return;
}
```

```
add {
   /* Add a constant B to each entry of x. */
   r1 = x;
   r2 = x;
   r3 = B;

       do 4 { *r1++ = a0 = *r2++ + *r3; }

   /* Wait for DAU pipeline to flush, and return. */
   nop; nop; nop;
   return;
}

transform { /* scale and add a constant to each entry of x. */

   /* Save the return address, pushing r14. */
   *r13++ = r14;

   /* Make some function calls. */
   call scale; call add;

   /* Pop, restoring r14, and return. */
   r13 = r13 - 4; r14 = *r13;
   return;
}

/* Subroutine return stack */ long stack[5];

/* Main entry point after reset.  Set up stack and make call. */

<MAIN> main {

   /* Use r13 as a dedicated stack pointer. */
   r13 = stack;

   /* Call a function to transform x. */
   call transform;

   /* Wait for the PC. */ wait: goto wait;
}
```

The stack has been programmed to grow towards *higher* addresses in memory. While it is often customary to have a stack grow toward lower memory addresses, keeping it at the end of the functions and growing towards higher memory makes it unlikely any program area would be overwritten by unexpected stack growth. Conventions about growing toward higher or lower addresses should be used as appropriate for a particular application.

USING NAMED REGISTERS, A FRACTAL PROGRAM

The named register feature of DSPASM makes it easier to write and maintain programs by allowing registers to be referred to with mnemonic names. This example shows how a fractal program can be written using this feature. The complete program can be found in the files mandel.c and mandel.dsp in the examples subdirectory on the supplied disk.

This program fragment iterates on a single pixel to see if the complex series z[n+1] = z[n]*z[n] + c is converging or not. Because complex arithmetic is involved, both the real and imaginary parts of z[n] must be manipulated, and these are pointed to by the named registers zr and zi. An advantage of programming with the named registers is that while writing your program you can concentrate on the algorithm, worrying about particular register assignments later. The fragment reads as follows:

```
/* Iterate on a single pixel for at most maxcount times. */

*zr = a0 = *zero;
*zi = a0 = *zero;

for ( count = 0 ; count < maxcount ; count++ ) {

   /* Compute the products of zr and zi. */
   a0 = *zr * *zr;
   a1 = *zi * *zi;
   a2 = *zr * *zi;
                nop; nop;

   /* Compute the norm. */
   a3 = a0 + a1;
                nop; nop;

   /* If norm >= 4.0 quit. */
   a3 = a3 - *four; nop; nop; nop;
          if ( age ) goto end_iterate;

   /* Otherwise compute new values of zr and zi. */
   a0 = a0 - a1; nop; nop;
   *zr = a0 = *cr + a0;
   *zi = a2 = *ci + a2 * *two;

   } end_iterate:
```

where the fragment must be assembled with the /2 two pass option because of the forward reference to the label end_iterate.

At the beginning of the function containing this fragment there should be a register statement to specify the assignments. A portion of it would read as:

```
register
   zr            = r1,
   zi            = r2,
   zero          = r3,
   count         = r4,
   maxcount      = r5,
   two           = r6,
   four          = r7,
   cr            = r8,
   ci            = r9, ... ;
```

Register names are local to a function, so that they can be reused again in later functions. In effect, named registers can be used like the local variables in a high level language.

Finally, in the register statement, a single register can also be given several names. The following is legal:

```
register
   i             = r1,
   j             = r1,
   k             = r2,
   l             = r3, ...
```

and so on. This can be useful when two parts of a function use the same register in a different way.

BLOCK MEMORY MOVES

This program shows how to perform a 32 bit block memory move on the DSP32C. This technique is useful for moving blocks of on board memory and also for reading and writing all 32 bits of the parallel port.

When first programming the DSP32C it is tempting to try move 32 bit data with instructions like:

```
r3 = *r1++; *r2++ = r3;
```

or

```
*r2++ = a0 = *r1++;
```

where r1 and r2 would be source and destination registers, and r3 and a0 would be temporaries. The problem with both of these techniques is that they don't accomplish a full general 32 bit move. In the first case what happens is because the CAU registers are only 24 bits wide, the top byte of the 32 bit value will be lost. In the second case, the problem is that the accumulator hardware always checks for "dirty" zeros, i.e. a floating point number with a zero exponent and nonzero mantissa. If it finds one, then the data is altered and changed to a "clean" zero with all bits zero. This means that general 32 bit values with arbitrary bit patterns could be modified. This invariably causes trouble when moving integer data or reading the parallel port.

The correct way to move general 32 bit data on the DSP32C is to use one of the DAU's tap update instructions. The tap update has a separate internal bus on the DSP32C that avoids the accumulator's "dirty" zero check. Furthermore, when placed inside a do loop, a tap update instruction can move 32 bits on each cycle without instruction fetching. This will move 32 bit words at a 12.5 Mhz rate!

The program below shows how to do the 32 bit moves with the tap update instruction. To run it, assemble and then use the DSPMON monitor to check the x and y arrays before and after execution.

```
constant RESET = 0, DATA = 100, N = 4;

<DATA> long xarray[N] = { 1, 2, 3, 4 }, yarray[N];

<RESET> main {

   register x = r1, y = r2;

   /* Point to the x and y arrays, and do the move. */
   x = xarray;
   y = yarray;
```

```
        do N { a0 = (*y++ = *x++) + a0; }

    /* Wait for the PC. */
    wait: goto wait;
}
```

Finally, note that the source or destination array can be one of the parallel ports on the DSP_400 or DSP_MOD boards. See the memory mapping chapter for the addressing of the parallel ports.

SETTING WAIT STATES

The DSP32C processor was designed so that wait states for external memory accesses are set by software and not by jumpers on the board. In addition, wait states are measured differently on the DSP32C than they are on most other processors, as only 1/4 of an instruction cycle is added to each memory access per wait state. This example covers both the setting of the wait states and their timing.

To set the wait states, the DSP32C pcw (processor control word) register bits 0-5 must be set, where the wait states can be programmed separately for external memory partitions A and B. This register can be written to *only* by the DSP32C, and cannot be directly accessed by the PC. The meanings of pcw bits 0-5 are as follows:

Wait state settings for pcw:

0001	enable wait state generator for partition B
0002	enable wait state generator for partition A
0000	1 wait states for partition B
0004	2 wait states for partition B
0008	3 wait states for partition B
000C	2 or more wait states controlled by SRDYN
0000	1 wait states for partition A
0010	2 wait states for partition A
0020	3 wait states for partition A
0030	2 or more wait states controlled by SRDYN

After reset, bits 0-5 are all set and the processor runs at 2 wait states for external memory accesses because the SRDYN pin is wired low on the board. The internal on chip memory always runs at 0 wait states. To run the external memory at 0 wait states, bits 0-5 of the pcw should all be set to 0. The following example shows one way to program other wait states:

```
constant

   ENABLE_B = 0x0001, ENABLE_A = 0x0002,

   B1 = 0x0000 | ENABLE_B,
   B2 = 0x0004 | ENABLE_B,
   B3 = 0x0008 | ENABLE_B,
   A1 = 0x0000 | ENABLE_A,
   A2 = 0x0010 | ENABLE_A,
   A3 = 0x0020 | ENABLE_A,
```

```
    WAITS = A1|B2;

set_waits { r1 = WAITS; pcw = r1; }
```

where 1 wait state has been selected for memory partition A, and 2 wait states for partition B.

As for the timing associated with wait states, the DSP32C adds 1/4 of an instruction cycle to a memory access for each wait state. The processor performance is degraded by only 25% per wait state whereas most processors add on one complete instruction cycle per wait state, degrading performance by 100%.

For an 80ns DSP32C CPU, a memory access at 0 wait states takes 20ns. Such fast access is required because memory may be accessed up to 3 times in a single DAU instruction, along with an access by the DMA controller. Each additional wait state adds 20ns to the basic instruction cycle, so the recommended memory speeds for the various wait states are:

0 ws	20-25 ns memories
1 ws	45 ns memories
2 ws	65 ns memories
3 ws	85 ns memories

Finally, note that the waits states for accesses to the 32 bit parallel port on the board are programmable just like they are for the memory. Because the parallel port is memory mapped, it appears to the DSP32C to be just like a memory location. You may wish to slow the processor down when it is writing to the parallel port and then speed it back up again, as a 20ns cycle time may be too fast for many devices connected to the port.

Additional information about setting wait states can be found in the AT&T DSP32C Information manual, page 7-18.

CIRCUITS

This chapter covers the detailed circuit diagrams for the DSP32C coprocessor boards. The schematics are hierarchical, first giving an overall functional view, and then giving subsheets of increasing detail.

Among the items you can find here are the DIP switches for selecting the PC/IO base addresses, and the pin outs for the headers on the boards. For users designing their own interfacing circuits, note that the header signals are all TTL compatible, and have timings consistent with those generated by the DSP32C itself.

Note that the DSP_400 and DSP_MOD boards are essentially the same, differing in only their on board memory arrays. The parallel port, serial port, and PC interfaces are identical between the two and are given only for the DSP_400 board. The on board memory array is given for the DSP_MOD board so you can see how the decoding is done. Only one copy of the memory chips or modules is listed because they are also repeated identically in the designs.

INDEX