DSP SOFTWARE

This chapter describes DSP software issues that are common to the boards containing DSPs. Commands and features specific to each board are discussed in the relevant chapter.

DSP5600x Description

The heart of several controller boards is either the Motorola DSP56001 or 56002, an integer digital signal processor processor with a 24-bit data word, a 16-bit address space, a fast ALU, a Reduced Instruction Set Computer (RISC) architecture that executes most instructions in one clock cycle and extensive on-chip peripheral support. These peripherals include separate address spaces for on-chip program and data memory, a synchronous serial interface, boot logic and a simple interface to an external data bus. A block diagram of the DSP is shown in Fig. 1. The timing board uses the DSP to write encoded 24-bit data words to its external data bus, which are decoded by external circuitry to control the CCD clocks and video processor. Documentation and software is now available from Motorola on CDROM. The technical documentation is free, and the development tools costs $99.

DSP Technical Documentation	CDDSP3/D, available by calling 1-800-441-2447
DSP Development Tools	DSPTOOLSCD-6.1-01273 (or later Rev.)

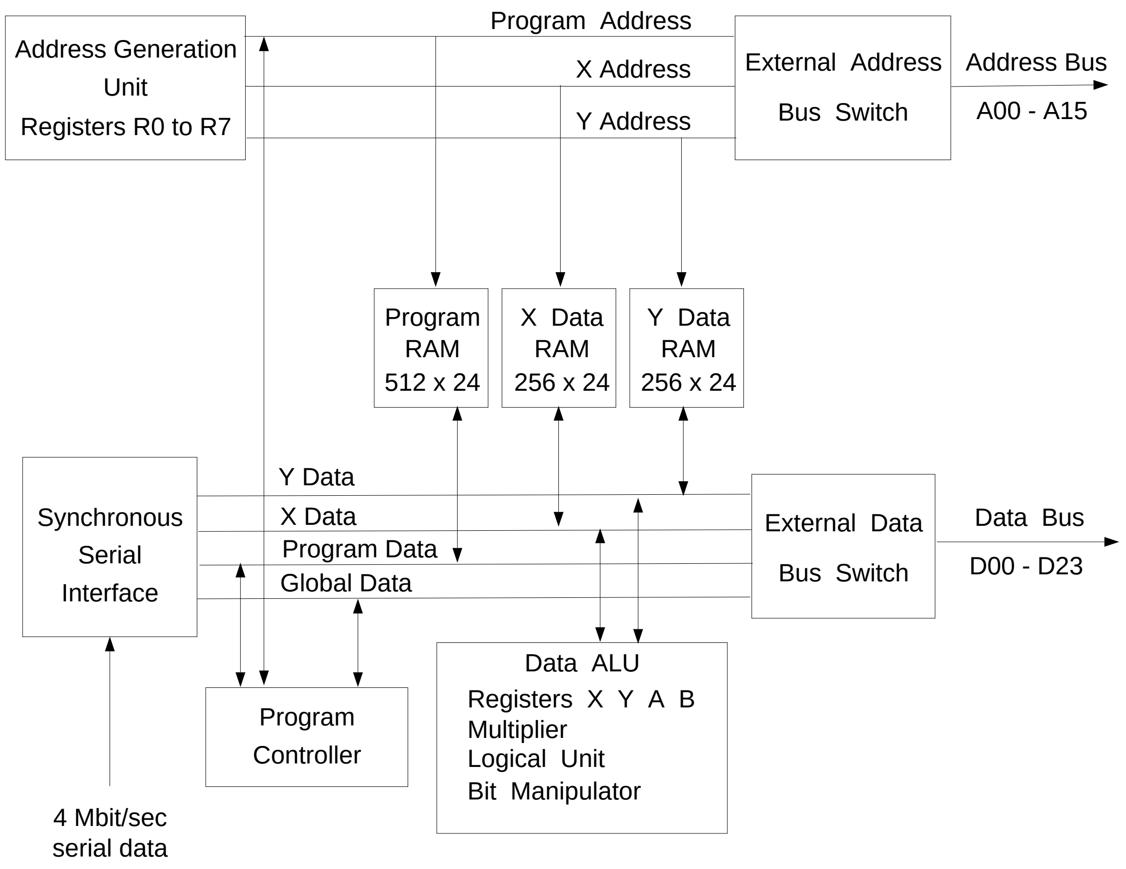

Figure 7-1: Block diagram of most of the components of the DSP56001.

Educational users may obtain the software products through University Support, HW 68, PO Box 2953, Phoenix, AZ 85062 (602-952-3857) at a substantial discount. At least an assembler and linker need to be obtained by users.

DSP Source Code Structure

DSP source code files are written in Motorola assembler language. There is a C compiler available, but it has not been used. Each board containing a DSP requires a boot file and an application file to operate. The boot file contains code to initialize the DSP after reset and provide communication and memory maintenance functions. The boot files are pretty similar from one board to the next and interpret the same set of commands. The boot file is written to the EEPROM or UVPROM by a ROM burner, and it is intended that a typical user will not have to heavily modify this code.

Application files contain software that is specific to the particular function and set of desired tasks for each board. Application programs are loaded into internal DSP memory by the boot code either from on-board ROM or by a downloading procedure from the host computer. Several application programs per board have been written to reflect different system configurations, readout modes and hardware testing needs. The application programs are intended to be heavily modified by users to optimize them to their particular needs. Internal addresses are somewhat different between the application program intended to be written to ROM and for downloading directly from the host computer to the DSP, and a command line parameter selects the desired mode during assembly of the application program.

The boot code is located in the upper half of ROM memory starting at address $4000 for the timing and VME interface boards, and address $6000 for the utility board.

Generating ROMs with a programmer

It may not necessary for a typical user to program the ROMs with a burner, but the following section details how it is done should the need arise. The following is a UNIX script for generating a Motorola S-record containing both boot and application programs for the timing board that can easily be sent to a ROM programmer.

```
#!/bin/csh
asm56000 -b -ltimboot.ls timboot.asm
asm56000 -b -ltim.ls -d DOWNLOAD 0 tim.asm
asm56000 -b -ltimtest.ls -d DOWNLOAD 0 timtest.asm
dsplnk -btim.cld -v timboot.cln tim.cln timtest.cln
rm tim.lod
cldlod tim.cld > tim.lod
srec -bs tim.lod
```

The file "timboot.asm" contains the boot source code, and the files "tim1.asm" and "timtest.asm" contain application programs. They are all separately assembled into relocatable files with the DOWNLOAD command line switch cleared to zero to indicate that code for ROM programming should be generated. The linker links together the two relocatable files and resolves all relative address, after which a tim.lod format file and a Motorola S-record file are generated. Both these files contains only ASCII characters, and the tim.lod file is particularly useful for determining the DSP and ROM addresses where code fragments will be located.

The development environment this is written for consists of a SUN running the Motorola DSP development software with a ROM programmer attached to it via an RS-232 serial link. The Motorola S-records contain the assembler output that the ROM programmer program takes as input. A trick is needed to get the boot code properly placed beginning at address $4000 for the timing and VME interface boards since they have only one byte-wide ROM. The idea is that the S-records are generated by this script for a starting address of $C000 which is the address from which the DSP normally boots, but is not mapped to anything on the controller boards since the address space of the ROMs is limited to 32k and the address line A15 is unused. This procedure will depend on the ROM programmer used, but in our case consists of giving a base address of $8000 to the programmer to subtract from all incoming addresses, so $C000 will be converted to the correct $4000. Then the ROM is programmed in two steps. First the boot program is programmed with the base address of $8000 by restricting the range of ROM addresses that can be programmed to lie between $4000 and $7FFF, then the application programs are programmed with a base address of zero but with a programming range restricted to be zero to $3FFF so the boot program (which is contained in the same S-record file) does not get programmed in this second pass. The instructions for our particular ROM programmer are as follows -

```
BASE        $8000               ; ROM programmer command
PROGRAM     $4000 $7FFF         ; ROM programmer command
cat tim.s                       ; UNIX command to send file
BASE        $0000               ; Now program the application program
PROGRAM     $0000 $3FFF
cat tim.s
```

The procedure for the utility board is simpler since the board has three ROMs and executes word transfers rather than byte transfer from memory to the DSP. The ROM generating script is -

```
#!/bin/csh
asm56000 -b -lutilboot.ls utilboot
asm56000 -b -lutilappl.ls -d DOWNLOAD 0 utilappl
asm56000 -b -lutiltest.ls -d DOWNLOAD 0 utiltest
dsplnk -butil.cld -v utilboot utilappl utiltest
del util.lod
cldlod util.cld > util.lod
srec -mw util.lod
```

The -mw option on the S-record command generates word wide addresses. Three files are generated, "util.p0", "util.p1" and "util.p2" each of which is directly written to seperate ROMs.

Generating application programs for downloading

Similar scripts can be used to generate application code for downloading directly from the host computer to ROMs located on the boards, saving their having to be removed and placed in a ROM programmer. These scripts will generate *.lod format files that are described at length in the Motorola DSP56001 programming literature that can be easily interpreted by a program running on the host computer that writes to the DSP internal memory a word at a time using the WRM command. A similar procedure would be used for writing to ROM when they are installed in the boards except that the scripts described above for ROM programming would be used rather than the ones that follow since the DOWNLOAD = 0 switch in the scripts above generate correct ROM addresses. For the VME interface board the downloading script is -

```
#!/bin/csh
asm56000 -b -ltimboot.ls timboot.asm
asm56000 -b -d DOWNLOAD 1 -ltim.ls tim.asm
dsplnk -btim.cld -v timboot.cln tim.cln
del tim.lod
cldlod tim.cld > tim.lod
```

Here only one application at a time is assembled, linked and downloaded at a time. The file "tim.lod" contains the boot program as well as the application program, and the user must insure that the boot program assembled with this script is identical to the one resident in the ROM on the board of interest. Downloading of the boot program can be easily prevented by having the host program skip over any address greater than $4000, and they won’t get written to the ROM anyway.

BOOTROM commands

Several commands are executed by all the DSP boards by the boot program, as follows. Every command must be preceded by a header ID.

TDL number "Test Data Link". The DSP will read "number" and transmit it back to the source in order to test functionality of the communications path.

RDM address - "Read DSP Memory". Read from internal DSP or ROM memory. The most significant nibble of the address designates the memory space, as follows:

Bit #20 = 1 selects P: memory.
Bit #21 = 1 selects X: memory.
Bit #22 = 1 selects Y: memory.
Bit #23 = 1 selects ROM memory.

WRM address value - "Write Memory". Write "value" to internal DSP or EEPROM memory, following the same encoding of the memory space as RDM.

LDA # - "Load Application". This transfers the application program from the ROM to the DSP memory, as well as the X: and Y: memory contents. # is the number of the application to be loaded and is between 1 and 4 for the timing and VME boards, and between 0 and 10 for the utility board.

A handshaking system exists to inform the sender of commands that they are received and that processing can occur. Most commands reply with a ’DON’ when they finish executing, though there are important exceptions to this.

A timeout routine has been implemented in the command processor that requires that all words of a command be received by the command processor within about two milliseconds of receiving the header ID. The TIMEOUT parameter in the boot files can be changed by the user if desired. If the number of words specified in the header is not received in time then the words are simply discarded.

Program Notes

Following are some notes on the programming of the controller boards. The intention is not to be exhaustive or complete since examination of the source code can often be a quicker and more reliable guide, but to be helpful with some of the subtleties of the code. The user is encouraged to be careful in modifying boot code, and might find some of these remarks useful.

A ROM identification area is provided at addresses P:06 and P:07, and is encoded as follows:

P:06	$aabbcc	aa = institutional code
		bb = location code
		cc = instrument code
P:07	$xxyyzz	xx = ROM major revision number
		yy = ROM minor revision number
		zz = Applicable board number (1 to 3)

Institution, location and instrument codes are no longer entered by SDSU when the ROM is burned to facilitate ROM maintenance and verification, but the user can easily enter the information with with WRM command if desired. Users may access these identification codes once the ROM software is loaded into the DSP by executing the RDM command on these two memory locations to verify that the correct version of the ROM software is installed.

The boot program starts at the location P:START that is specified in an equate table at the beginning of each source code file. It is chosen to be as low as possible and uses interrupt service routine vector locations that are assigned by the DSP but unused by the particular board. As a result a warning message is generated during assembly to the effect that illegal instructions for interrupt service routines are being generated, and these warnings should be ignored.

Command processing is done by having routines linked to input data devices add incoming commands to circular buffers indexed by address registers that are permanently allocated to that task. The address register assignment is listed after the ROM_ID table in the source code. The circular buffers are set up to be 32 words long, which is enough to contain typically ten commands, and resides in the Y: memory space. This allows several commands to be sent to a given board in sequence without having to wait for each one to be executed; they will be executed in the order in which they are received. Each input device is assigned a separate circular buffer to prevent intermingling of commands. If a command is not being executed then the program is scanning the address register values for an increment caused by an incoming command. As soon as one is found a consistency check is performed to verify that its first word is a valid header. If it is correct then the last byte is read to determine when the complete command has been received. If there is more than one input data source then the command is moved to a separate buffer that contains commingled commands. A loop is entered to check that the entire command has been entered. Once the command is complete the header destination number is examined to see if the command needs to be executed by the board or passed on to another one in the chain. If it to be executed by the board then a table of valid commands is examined for a match with the second word of the command. If no match is found an 'ERR' reply is returned to the source. Otherwise execution continues at the address indicated by the command.

A reply routine, named FINISH, constructs the reply header ID by moving the source byte to the destination byte location, tacking on the correct source byte number, and adding two for the number of words in the reply. All replies are two words in length. The reply is added to the circular command buffer and then processed as any other command would be.

The serial communication interface (SCI) receiver is handled by an interrupt service routine on the utility board. Care is taken in the routines to save and restore all registers that are used by the routine, which is done by writing them to X: memory. The routines just add together the three bytes that comprise a 24-bit word by reading the bytes from three sequential memory-mapped addresses in the X: I/O memory space dedicated to SCI support. Separate interrupt service routines are maintained for error conditions to clear the SCI hardware of the error condition.

The INIT routine is executed once by the DSP on boot up. It is overwritten by application code once the LDA command is executed or code is downloaded from the host. The INIT routine initializes DSP control registers, sets up the circular buffers and registers, reads in X: memory from ROM, and initializes for the processing of interrupts. The timing board INIT routine goes on to set the DC bias DACs with safe intermediate values in preparation for the power-up sequence. A jump table exists right before the INIT routine to allow the boot code to access application commands after application code has been loaded. Before it is loaded the jump table is a simple do-nothing, and the jump table is overwritten with valid addresses by the application code.

The boot code stores all the constants it needs to operate in X: memory. The application code generally stores constants it needs in Y: memory, although its command table is still stored in X: memory as an extension onto the boot command table. When an application program is loaded the command table entries for it are added to the boot command table in X: memory by the LDA command. Many constants are stored in X: or Y: memory as a device to conserve P: memory space. As an example, the command MOVE #$FFB0,R6 occupies two locations of P: memory and takes two clock cycles of execution time, whereas the instruction MOVE X:<CFFB0,R6 takes one P: location and one X: location and takes only one instruction cycle to execute. Since P: memory is in much shorter supply than either X: or Y: memory this helps to conserve overall memory usage and execution time.