DSP56303EVM

User's Manual

Motorola, Incorporated
Semiconductor Products Sector
DSP Division
6501 William Cannon Drive West
Austin, TX 78735-8598

Order this document by:
DSP56303EVMUM/AD

Introduction

This document supports the DSP56303 Evaluation Module (DSP56303EVM) including a description of its basic structure and operation, the equipment required to use it, the specifications of the key components, the provided software (such as the demonstration code, the self-test code, and the software required to develop and debug sophisticated applications), schematic diagrams, and a parts list. Section 1 is a Quick Start Guide. Section 2 provides instructions for running the demonstration software. Section 3 provides a simple programming example. Section 4 provides detailed information about key components in the evaluation module. Appendix A has detailed schematics. Appendix B is the parts list. Appendix C includes additional notes for using the Assembler. Appendix D is a tutorial for programming the codec. This document has been designed for users experienced with DSP development tools. For users with little or no DSP experience, detailed information is provided in the additional documents supplied with this kit.

How to reach us:

USA/Europe/Locations Not Listed:
Motorola Literature Distribution
P.O. Box 20912
Phoenix, Arizona 85036
1 (800) 441-2447 or
1 (602) 303-5454

Mfax™
RMFAX0@email.sps.mot.com
TOUCHTONE (602) 244-6609

Asia/Pacific:
Motorola Semiconductors H.K. Ltd.
8B Tai Ping Industrial Park
51 Ting Kok Road
Tai Po, N.T., Hong Kong
852-2662928

Technical Resource Center:
1 (800) 521-6274

DSP Helpline
dsphelp@dsp.sps.mot.com

Japan:
Nippon Motorola Ltd.
Tatsumi-SPD-JLDC
6F Seibu-Butsuryu-Center
3-14-2 Tatsumi Koto-Ku
Tokyo 135, Japan
03-3521-8315

Internet:
http://www.motorola-dsp.com

MOTOROLA

CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION 1

QUICK START GUIDE

1.1 OVERVIEW

This section provides a summary description of the evaluation module contents, additional requirements, and quick installation and test information. Detailed information about the DSP56303EVM design and operation is provided in the remaining sections of this manual.

1.2 EQUIPMENT

The following section gives a brief summary of the equipment required to use the DSP56303 Evaluation Module (DSP56303EVM), some of which will be supplied with the module, and some of which must be supplied by the user.

1.2.1 What You Get with the DSP56303EVM

The following material is provided with the DSP56303EVM:

- DSP56303 Evaluation Module board
- DSP56303EVM Product Information
- DSP56303 Technical Data (preliminary)
- DSP56303 Chip Errata
- DSP56300 Family Manual
- DSP56303 Product Specifications Revision 1.02
- DSP56303EVM User’s Manual (this document)
- Crystal Semiconductor CS4215 16-Bit Multimedia Audio Codec Data Sheet
- Domain Technologies Debug-56K Debugger manual for Motorola 16- and 24-bit DSPs
- Set of diskettes (3-1/2 inch) containing required software:
  - GUI Debugger from Domain Technologies (1 diskette)
  - Assembler/linker/example software from Motorola (1 diskette)

1.2.2 What You Need to Supply

- PC (386 class or higher) with:
  – Windows 3.1 or higher (including Windows 95)
  – 4 Mbytes of memory minimum
  – 3-1/2-inch, high density, diskette drive
  – Hard drive with 4 Mbytes of free disk space
  – Mouse
  – RS-232 serial port supporting 9,600–57,600 bit-per-second transfer rates
- RS-232 interface cable (DB9 male to DB9 female)
- Power supply, 7–9 V ac or dc input into a 2.1 mm power connector
- Audio source (tape player, radio, CD player, etc.)
- Headphones
- Audio interface cable with 1/8-inch stereo plugs

1.3 INSTALLATION PROCEDURE

Installation requires four basic steps:

1. Preparing the DSP56303EVM board
2. Connecting the board to the PC and power
3. Installing the software
4. Testing the installation

1.3.1 Preparing the DSP56303EVM

CAUTION

Because all electronic components are sensitive to the effects of electrostatic discharge (ESD) damage, correct procedures should be used when handling all components in this kit and inside the supporting personal computer. Use the following procedures to minimize the likelihood of damage due to ESD:

- **Always handle all static-sensitive components only in a protected area, preferably a lab with conductive (anti-static) flooring and bench surfaces.**
- **Always use grounded wrist straps when handling sensitive components.**
- **Never remove components from anti-static packaging until required for installation.**
- **Always transport sensitive components in anti-static packaging.**

Locate jumper blocks J4, J7, and J9, as shown in **Figure 1-1**. Make sure that, for blocks J4 and J7, all six positions on each block are jumpered, and for J9, there is a jumper connecting pins 2 and 3. These jumpers perform the following functions:

- J4 controls the interface between the DSP56302 JTAG / OnCE port and DSP56002 Synchronous Serial Interface (SSI).
- J7 controls the interface between the audio codec and the DSP56302 Enhanced Synchronous Serial Interface (ESSI0).
- J9 defines the external memory map.

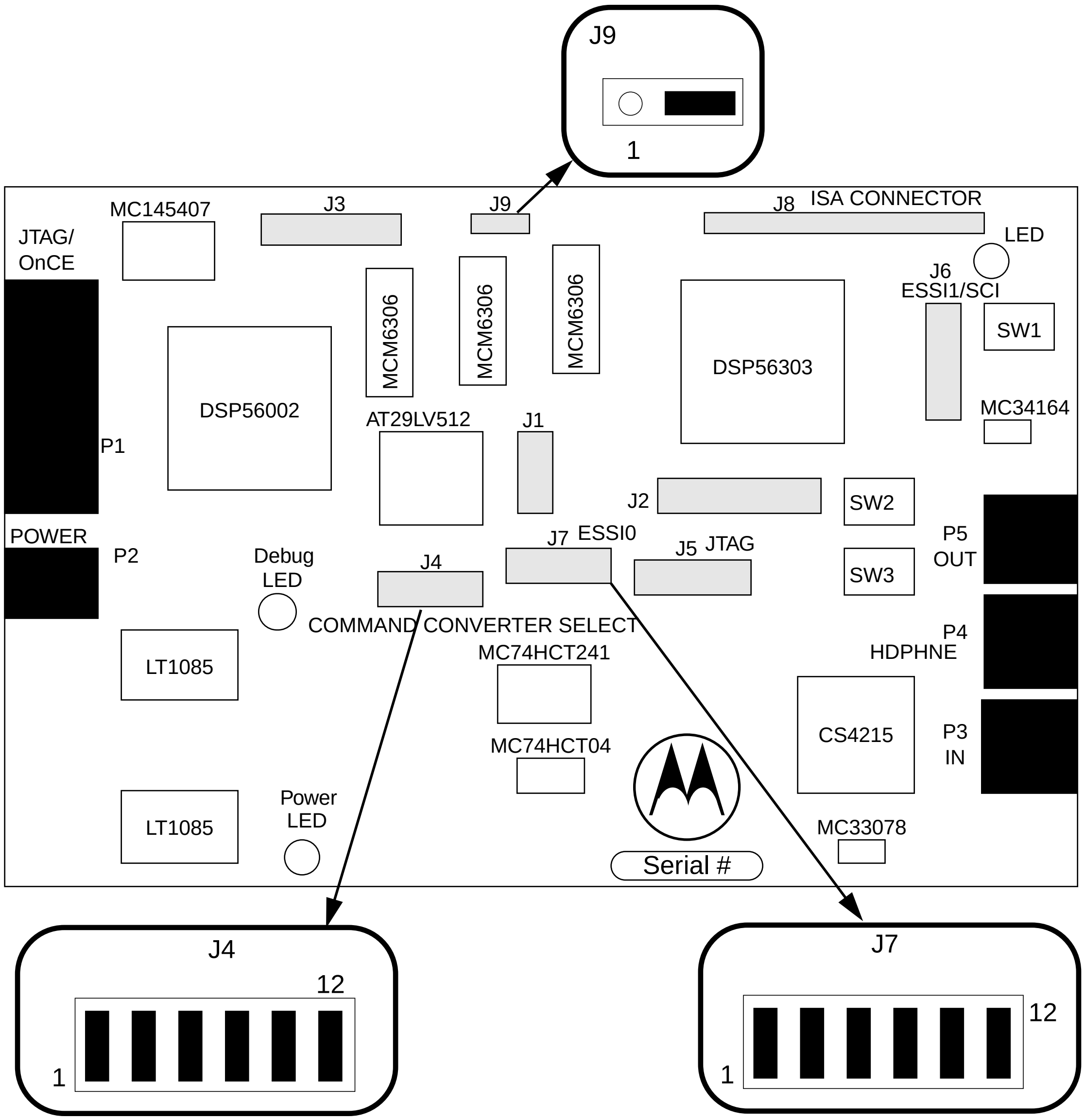


Figure 1-1 DSP56303EVM Component Layout

1.3.2 Connecting the DSP56303EVM to the PC and Power

Figure 1-2 shows the interconnection diagram for connecting the PC and the external power supply to the DSP56303EVM board. Use the following steps to complete cable connections:

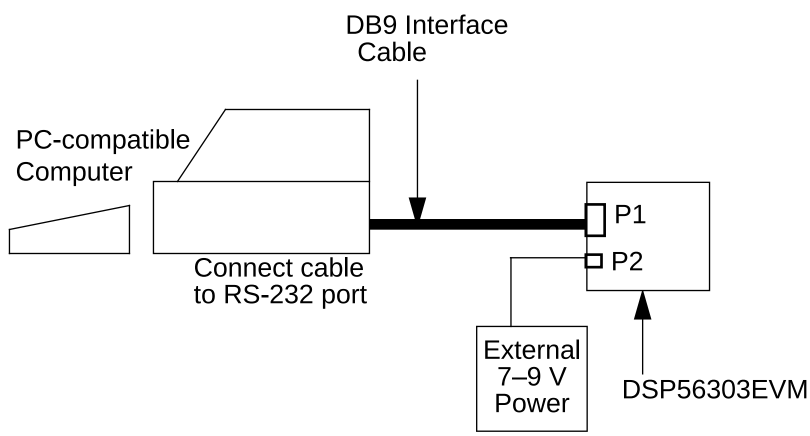


Figure 1-2 Connecting the DSP56303EVM Cables

1. Connect the DB9P end of the RS-232 interface cable to the RS-232 port connection on the PC.
2. Connect the DB9S end of the cable to P1, shown in **Figure 1-1**, on the DSP56303EVM board. This provides the connection to allow the PC to control the board function.
3. Make sure that the external 7–9 V power supply does not have power supplied to it.
4. Connect the 2.1 mm output power plug into P2, shown in **Figure 1-1**, on the DSP56303EVM board.
5. Apply power to the power supply. The green Power LED will light when power is correctly applied.

1.3.3 Installing the Software

The DSP56303EVM software includes the following:

- Motorola diskette containing:
  - Assembler
  - Linker
  - Test code
  - Demonstration code
- Domain Technologies diskette containing the windowed user interface debug software

Use the following steps to install the software:

1. Insert the Motorola software diskette into the PC diskette drive.
2. If the system is not already running in Windows, start Windows.
3. From Windows, select a DOS window and run the install program. The general format for the command line of the install program is ***install <source>***. For example, if your diskette drive is a, type in ***a:install a:.*** The program installs the software in the default destination c:\evm30xw.
4. Close the DOS window and remove the Motorola software diskette from the diskette drive. Insert the Domain Technologies diskette labeled Debug-56K into the drive.
5. From Windows, run the Debugger installation program install.exe on the diskette. This can be done from the Microsoft Windows Program Manager by pulling down the File menu, choosing Run, entering ***a:install*** on the command line, and clicking OK.
6. Read the README.TXT if you are installing the Debugger program for the first time. This provides information in addition to that provided by the Domain Technologies manual included with this kit.
7. The install program creates a program group called “evm5630x” and a program icon called “evm30xw” within Windows. This step completes the software installation.

1.3.4 Testing the DSP56303EVM

The following sections describe the DSP56303EVM self-test and analysis programs. The self-test allows the user to determine if the DSP56303EVM is properly connected and ready for operation. The analysis program allows the user to examine the DSP56303EVM operation in more detail to diagnose problems that may occur.

1.3.4.1 DSP56303EVM Self-test

This section describes how to run the DSP56303EVM self-test. The self-test comes pre-loaded into the Flash PEROM of the DSP56303EVM, and is therefore very easy to use. There are two parts to the self-test, as follows:

- The first part of the self-test examines the external memory of the DSP56303EVM by writing to, and reading from, the external SRAM. It also examines the analog circuitry of the DSP56303EVM by analyzing the response of the analog circuitry to a sequence of tones. This part of the self-test begins running immediately when the DSP56303 boots from the Flash PEROM and, after startup, by invoking external Interrupt Request A ($\overline{\text{IRQA}}$). $\overline{\text{IRQA}}$ can be activated by pressing the switch labelled SW2.
- The second part of the self-test analyzes the audio circuitry of the DSP56303EVM by moving an audio signal through the codec. The second part of the self-test begins following a receipt of external Interrupt Request D ($\overline{\text{IRQD}}$). $\overline{\text{IRQD}}$ is activated by pressing the switch labelled SW3.

The following steps describe how to run the self-test code for the DSP56303EVM.

1. Install the cable with the 1/8-inch stereo plugs between the line input jack labelled P3/IN on the DSP56303EVM and the line output jack labelled P5/OUT on the DSP56303EVM. Refer to **Figure 1-1** on page 1-6 for the location of the line input and output jacks.
2. Press the switch labelled SW1 on the DSP56303EVM. Refer to **Figure 1-1** on page 1-6 for the location of SW1. This switch resets the DSP56303 and allows the DSP56303 to boot from the Flash PEROM, which is preloaded with the self-test code, and begins the first part of the self-test. The red LED at D12 lights when the test begins running. When this test is complete, the LED turns off if the DSP56303EVM passed the test. If the DSP56303EVM fails the test, the LED flashes. To repeat the test, press SW2 to invoke $\overline{\text{IRQA}}$.
3. Now run the second part of the self-test code to test the analog circuitry of the DSP56303EVM. Using the cable with the 1/8-inch stereo plugs, connect the phone output of the audio source to the line input jack labelled P3/IN on the DSP56303EVM. Also connect a pair of headphones to the headphone jack labelled P4/HDPHNE on the DSP56303EVM. Refer to **Figure 1-1** on page 1-6

for the locations of the line input and headphone jacks. Start the audio source and put on the headphones.

4. Press the switch labelled SW3 on the DSP56303EVM. Refer to **Figure 1-1** on page 1-6 for the location of SW3. You should hear the audio through the headphones with a slight echo added.

To rerun the first part of the self-test, perform steps 1 and 2 above. To repeat the second part of the test, perform steps 3 and 4 above. You can switch between the parts at any time.

If both parts of the self-test complete correctly, the DSP56303EVM is correctly installed, operational, and ready for use. If either part of the self-test fails, double-check the jumper settings and cable connections (power, RS-232, and audio) and repeat the test. If the DSP56303EVM continues to fail either test, run the DSP56303EVM analysis program described in **Section 1.3.4.2** to try to determine the cause of the problem.

1.3.4.2 DSP56303EVM Analysis Program

This section describes how to run the DSP56303EVM analysis program. The analysis program functions in the same way as the first part of the self-test, but provides a list of output results. This allows the user to determine exactly how the DSP56303EVM is failing the tests.

The following steps describe how to run the analysis program for the DSP56303EVM.

1. Run the program *evmtst2.exe*. This can be done from the Microsoft Windows Program Manager by pulling down the File menu, choosing Run, entering ***c:\evm30xw\evmtst2.exe*** on the command line, and clicking OK.
2. When prompted, install the cable with the 1/8” stereo plugs between the line input jack labelled P3/IN on the DSP56303EVM and the line output jack labelled P5/OUT on the DSP56303EVM. Refer to **Figure 1-1** on page 1-6 for the location of the line input and output jacks. Press any key to continue the test.
3. Wait for the test to run.

Note: When the test is complete, the test program window will go to the background. Click on the test window to bring it to the foreground.

4. After the test is complete, the test asks for the DSP56303EVM serial number. The serial number can be found at the bottom of the DSP56303EVM board as shown in **Figure 1-1** on page 1-6. Type in the serial number from the board and press the return key.

The test program will output a set of diagnostics to the screen, similar to those shown in **Figures 1-3** and **1-4**, showing the results of the analog circuitry and memory tests. The analog circuitry results show the DC offset, the noise level, and the response of the analog circuitry to the sequence of tones listed in the first column. The second two columns contain the raw data received by the DSP56303 from the left and right channels of the analog circuitry. The raw data is evaluated in decibels relative to the maximum value and placed in the next two columns, labelled dB below MAX.

-- DSP56303EVM Performance Analysis Ver.2.00--
EVM Serial No.: 1234 Tue Jun 04 17:45:47 1996
DSP56303 Chip Revision No. 000303

	LEFT	RIGHT	dB below MAX				
DC Offset:	65535	65535	6.02	6.02	10.00		
NOISE:	0	0	0.00	0.00	10.00		
24 kHz:	3086	2773	-20.52	-21.45	-20.00		
6 kHz:	20	24014	-64.29	-2.70	-6.00	*	
1.5 kHz:	24422	22	-2.55	-63.46	-6.00		*
375 Hz:	20	19	-64.29	-64.73	-6.00	*	*
94 Hz:	22772	22553	-3.16	-3.24	-6.00		
23 Hz:	15563	15218	-6.47	-6.66	0.00		
12 Hz:	8627	8401	-11.59	-11.82	0.00		
6 Hz:	2701	2567	-21.68	-22.12	-20.00		

Pass: 6	Address:	X:8200	Y:8200
	Expected:	000000	000000
	Received:	000000	FF0000

MEMORY ERRORS FOUND! (check J9 is in correct position)

AUDIO FAIL

*** FAIL *** FAIL *** FAIL *** FAIL ***

Figure 1-3 DSP56303EVM Test Sample Output—Fail

```
-- DSP56303EVM Performance Analysis    Ver.2.00--
EVM Serial No.: 1234                        Tue Jun 04 17:45:47 1996
DSP56303 Chip Revision No. 000303
                    LEFT        RIGHT     ————dB below MAX—————
```

	LEFT	RIGHT			
DC Offset:	65535	65535	6.02	6.02	10.00
NOISE:	0	0	0.00	0.00	10.00
24 kHz:	3086	2773	-20.52	-21.45	-20.00
6 kHz:	24481	24014	-2.53	-2.70	-6.00
1.5 kHz:	24422	24140	-2.55	-2.65	-6.00
375 Hz:	24040	23868	-2.69	-2.75	-6.00
94 Hz:	22772	22553	-3.16	-3.24	-6.00
23 Hz:	15563	15218	-6.47	-6.66	0.00
12 Hz:	8627	8401	-11.59	-11.82	0.00
6 Hz:	2701	2567	-21.68	-22.12	-20.00

```
Pass: 6     Address:    X:0000    Y:0000
            Expected:   000000    000000
            Received:   000000    000000
-------- PASS --------
```

Figure 1-4 DSP56303EVM Test Sample Output—Pass

The last column shows the acceptable responses, in dB, of the analog circuitry for the various tones. The dB levels for each channel are compared to the acceptable responses to determine if the DSP56303EVM passed the analog circuitry test. The dB levels must be below the acceptable responses for DC offset, noise, 24 kHz, 23 Hz, 12 Hz, and 6 Hz and above the acceptable responses for 6 kHz, 1.5 kHz, 375 Hz, and 94 Hz for the DSP56303EVM to pass the analog circuitry test. If either channel's dB levels do not satisfy the acceptable responses, an asterisk will be located at the end of the row that did not pass the analog circuitry test. For example, in **Figure 1-3**, the left channel did not satisfy the acceptable response for 6 kHz, the right channel did not satisfy the acceptable response for 1.5 kHz, and both channels did not satisfy the acceptable response for 375 Hz. If the DSP56303EVM passed the analog circuitry test, no asterisks will be seen at the end of the rows, as in **Figure 1-4**.

The results of the memory test are shown below the results of the analog circuitry test. These results tell how many passes of the external memory test were run and if the DSP56303EVM passed the test. If all the values in the Address, Expected, and Received fields are zero, as in **Figure 1-4**, the DSP56303EVM passed the external memory test. If the DSP56303EVM failed the external memory test, these fields will tell which memory location caused the failure, the value that was expected to be read, and the value that was actually read from that memory location, as in **Figure 1-3**.

The last line of the diagnostics tells if the DSP56303EVM passed or failed the test. If the last line says PASS, as in **Figure 1-4**, then the DSP56303EVM passed the test and is ready for use. If the last line says FAIL, as in **Figure 1-3**, double check the jumpers and the power, RS-232, and stereo connections and repeat the test. If the DSP56303EVM continues to fail the test, email the DSP help line at dsphelp@dsp.sps.mot.com.

Now the test is complete and the DSP56303EVM is ready for operation.

SECTION 2

RUNNING THE DEMONSTRATION FILE

2.1 OVERVIEW

This section describes the demonstration that is included with the DSP56303EVM. The demonstration is designed to show the advantage of 24-bit DSPs over 16-bit DSPs. Even if the system input data is limited by a 16-bit data converter, the system can still benefit greatly from using the 24-bit DSP architecture.

This demonstration takes an input signal, digitizes it, and adds a 60 Hz tone to simulate the noise that can be generated by a 60 Hz AC power line. The signal is then filtered using one of three sets of coefficients.

The first set of coefficients performs no filtering, simply allowing the corrupted signal to pass through. The second set of coefficients use 24-bit coefficients to correspond to a 60 Hz notch filter that removes the 60 Hz portion of the corrupted signal. The final set of coefficients are the same as those used for the 24-bit filtering, only rounded to 16 bits.

The following sections give step-by-step instructions on how to run this demo.

2.2 SETTING UP THE DEMONSTRATION

Perform the following steps to set up the demonstration:

1. Set up the audio source, preferably a music source with a headphone output, such as a portable radio, tape player, or CD player.
2. Make sure the DSP56303EVM has been installed and tested, as described in **Section 1**.
3. Using the cable with the 1/8-inch stereo plugs, connect the output of the audio source to the stereo input jack labelled P3/IN on the DSP56303EVM.
4. Connect a pair of headphones to the stereo headphone jack labelled P4/HDPHNE on the DSP56303EVM. Refer to **Figure 2-1** for the locations of the line input and headphone jacks.

Start the audio source and put on the headphones.

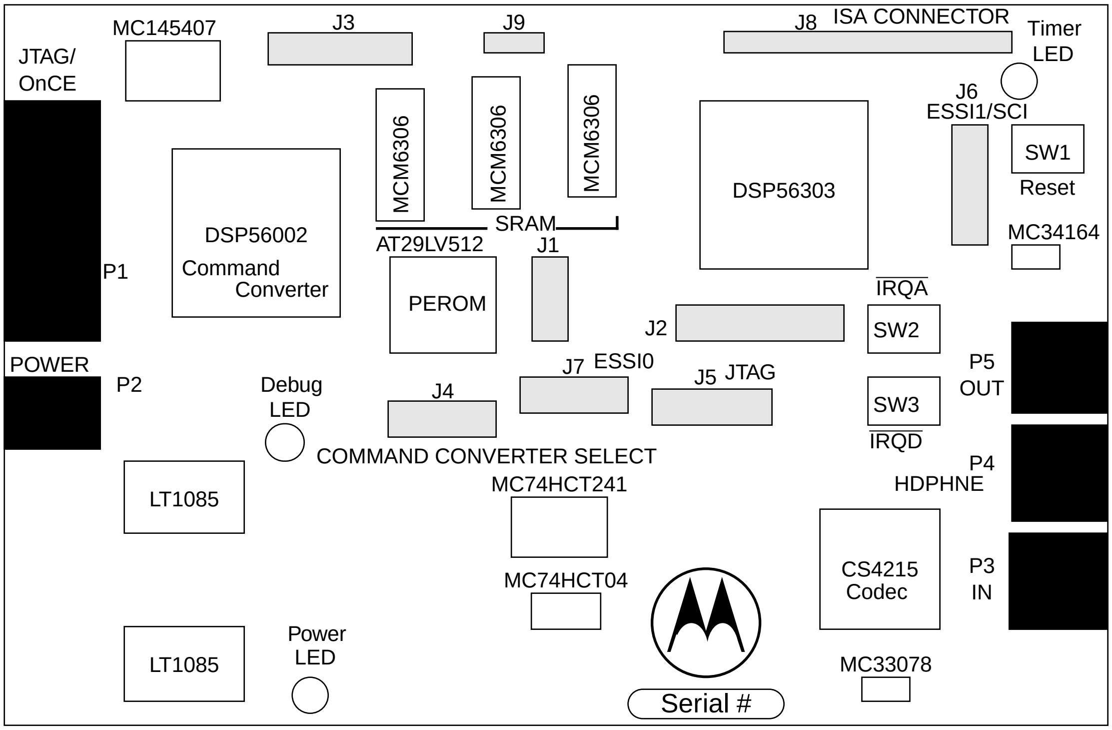


Figure 2-1 DSP56303EVM Component Layout

2.3 RUNNING THE DEMONSTRATION

To run the demonstration:

1. Start Windows.
2. Invoke the Debugger by double-clicking on the icon labelled *evm30xw* in the *evm5630x* program group.
3. Click on the command window and type ***force r*** to reset the DSP56303 and enter the debug mode.
4. Type in ***load 60hzdemo*** to load the demonstration file into the DSP56303.
5. Finally, type ***go*** to begin the demonstration. The first set of coefficients (no filtering) is used when the demonstration begins. The first set of coefficients performs no filtering, simply allowing the corrupted signal to pass. Thus, the output contains the input signal with the added 60 Hz tone when the demo begins.

6. Briefly press SW2 to invoke external internal request A ($\overline{\text{IRQA}}$). Invoking $\overline{\text{IRQA}}$ causes the DSP to use the 24-bit filter coefficients. Refer to **Figure 2-1** for the location of SW2.
7. To hear the results of using the 16-bit coefficients, briefly press the switch labeled SW3. This invokes the external interrupt request D ($\overline{\text{IRQD}}$), which causes the DSP to use the 16-bit filter coefficients. Refer to **Figure 2-1** for the location of SW3.

Note: You will hear a significant difference between filtering with the 24-bit and 16-bit coefficients. The reason for this contrast is that it is impossible to place the notch of the filter exactly on 60 Hz with 16-bit coefficients. The 16-bit filter notch is slightly offset and, therefore, misses the frequency of the noise. With the 24-bit coefficients, it is possible to place the filter notch exactly on 60 Hz.

8. To stop the execution of the demo, type ***force r*** in the command window in the bottom left corner of the screen.
9. To exit the Debugger, type ***quit***.

Running the Demonstration

SECTION 3

EXAMPLE TEST PROGRAM

3.1 OVERVIEW

This section contains an example that illustrates how to develop a very simple program for the DSP56303. This example has been designed for users who have little or no experience with the DSP development tools. The example demonstrates the form of assembly programs, gives instructions on how to assemble programs, and shows how the Debugger can be used to verify the operation of programs.

Figure 3-1 shows the development process flow for assembly programs. The rounded blocks represent the assembly and object files. The white blocks represent software programs to assemble and link the assemble programs. The gray blocks represent hardware products. The following sections give basic information regarding the assembly program, the Assembler, the Linker and the object files. Detailed information about these subjects can be found in the Assembler and Linker manuals provided with the Motorola DSP CLAS software package available through your Motorola sales office or distributor. The documentation is also available through the Motorola DSP internet URL *http://www.motorola-dsp.com*.

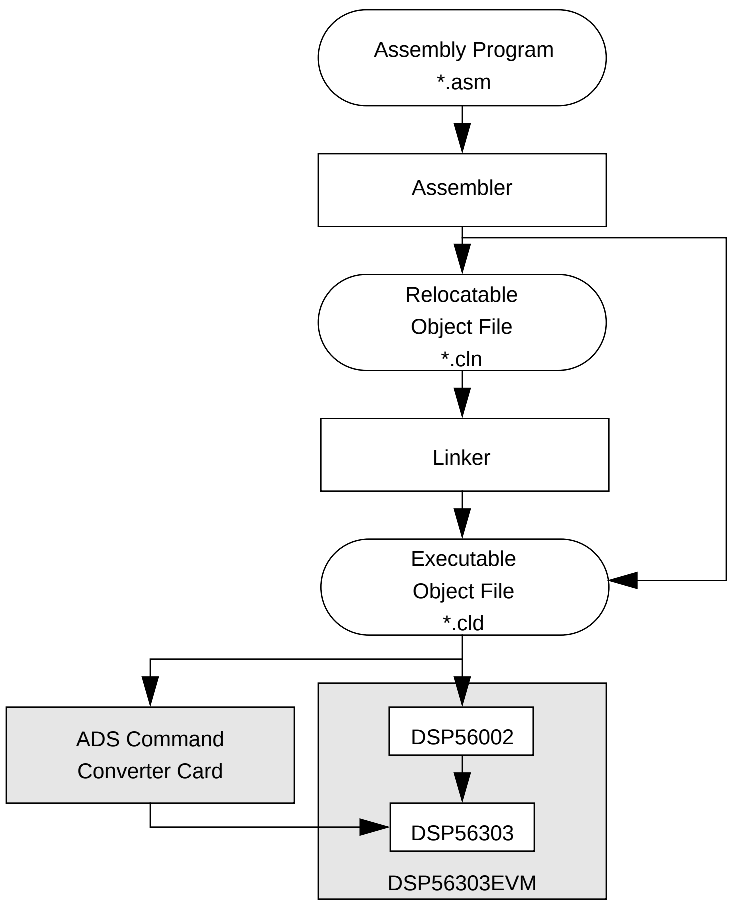


Figure 3-1 Development Process Flow

3.2 WRITING THE PROGRAM

The following sections describe the format of assembly language source statements and give an example assembly program.

3.2.1 Source Statement Format

Programs written in assembly language consist of a sequence of source statements. Each source statement may include up to six fields separated by one or more spaces or tabs: a label field, an operation field, an operand field, up to two data transfer fields, and a comment field. For example, the following source statement shows all six possible fields:

```
trm      mac   x0,y0,a    x:(r0)+,x0    y:(r4)+,y0      ;Text
```

Label   Operation   Operand   X Data Transfer   Y Data Transfer   Comment

3.2.1.1 Label Field

The label field is the first field of a source statement and can take one of the following forms:

- A space or tab as the first character on a line ordinarily indicates that the label files is empty and that the line has no label.
- An alphabetic character as the first character indicates that the line contains a symbol called a label.
- An underscore as the first character indicated that the label is a local label.

With the exception of some directives, a label is assigned the value of the location counter of the first word of the instruction or data being assembled. A line consisting of a label only is a valid line and has the effect of assigning the value of the location counter to the label.

3.2.1.2 Operation Field

The operation field appears after the label field and must be preceded by at least one space or tab. Entries in the operation field may be one of three types:

- **Opcode**—mnemonics that correspond directly to DSP machine instructions
- **Directive**—special operation codes known to the Assembler which control the assembly process

- **Macro call—**invocation of a previously defined macro which is to be inserted in place of the macro call

3.2.1.3 Operand Field

The interpretation of the operand field is dependent on the contents of the operation field. The operand field, if present, must follow the operation field and must be preceded by at least one space or tab.

3.2.1.4 Data Transfer Fields

Most opcodes can specify one or more data transfers to occur during the execution of the instruction. These data transfers are indicated by two addressing mode operands separated by a comma, with no embedded blanks. If two data transfers are specified, they must be separated by one or more blanks or tabs. Refer to the DSP56300 Family Manual for a complete discussion of addressing modes that are applicable to data transfer specifications.

3.2.1.5 Comment Field

Comments are not considered significant to the Assembler, but can be included in the source file for documentation purposes. A comment field is composed of any characters that are preceded by a semicolon.

3.2.2 Example Program

This program takes two lists of data, one in X memory, and one in Y memory, and calculates the sum of the products of the two lists. Calculating the sum of products is the basis for many DSP functions. Therefore, the DSP56303 has a special instruction (MAC) which multiplies two values and adds the result to the contents of an accumulator. This program is provided as example.asm on the DSP56303EVM diskette and is placed in the evm30xw directory by the installation procedure.

Example 3-1 Simple DSP56303 Code Example

```
;*************************************************************
;A SIMPLE PROGRAM: CALCULATING THE SUM OF PRODUCTS
;*************************************************************
PBASE      EQU        $100       ;instruct the assembler to replace
                                 ;every occurrence of PBASE with $200
XBASE      EQU        $0         ;used to define the position of the
                                 ;data in X memory
YBASE      EQU        $0         ;used to define the position of the
                                 ;data in Y memory
;*************************************************************
;X MEMORY
;*************************************************************
           org        x:XBASE    ;instructs the assembler that we
                                 ;are referring to X memory starting
                                 ;at location XBASE
list1      dc         $475638,$738301,$92673a,$898978,$091271,$f25067
           dc         $987153,$3A8761,$987237,$34b852,$734623,$233763
           dc         $f76756,$423423,$324732,$f40029
;*************************************************************
;Y MEMORY
;*************************************************************
           org        y:YBASE    ;instructs the assembler that we
                                 ;are referring to Y memory starting
                                 ;at location YBASE
list2      dc         $f98734,$800000,$fedcba,$487327,$957572,$369856
           dc         $247978,$8a3407,$734546,$344787,$938482,$304f82
           dc         $123456,$657784,$567123,$675634
;*************************************************************
;PROGRAM
;*************************************************************
           org        p:0        ;put following program in program
                                 ;memory starting at location 0
           jmp        begin      ;p:0 is the reset vector i.e. where
                                 ;the DSP looks for instructions
                                 ;after a reset
           org        p:PBASE    ;start the main program at p:PBASE

begin
           move       #list1,r0  ;set up pointer to start of list1
           move       #list2,r4  ;set up pointer to start of list2
           clr        a          ;clear accumulator a
           move       x:(r0)+,x0   y:(r4)+,y0
                                 ;load the value of X memory pointed
                                 ;to by the contents of r0 into x0 and
                                 ;post-increment r0
                                 ;load the value of Y memory pointed
                                 ;to by the contents of r4 into y0 and
                                 ;post-increment r4
```

Example 3-1 Simple DSP56303 Code Example (Continued)

```
          do        #15,endloop;do 15 times
          mac       x0,y0,a    x:(r0)+,x0    y:(r4)+,y0
                               ;multiply and accumulate, and load
                               ;next values
endloop   jmp       *          ;this is equivalent to
                               ;label jmp label
                               ;and is therefore a never-ending,
                               ;empty loop
;**************************************************************
;END OF THE SIMPLE PROGRAM
;**************************************************************
```

3.3 ASSEMBLING THE PROGRAM

The following sections describe the format of the Assembler command, give a list of Assembler special characters and directives, and give instructions to assemble the example program.

3.3.1 Assembler Command Format

The Motorola DSP Assembler is included with the DSP56303EVM on the Motorola 3-1/2 inch diskette and can be installed by following the instructions in **Section 1.3.3**. The Motorola DSP Assembler is a program that translates assembly language source statements into object programs compatible with the DSP56303. The general format of the command line to invoke the Assembler is:

asm56300 [options] <filenames>

where ***asm56300*** is the name of the Motorola DSP Assembler program, and ***<filenames>*** is a list of the assembly language programs to be assembled. The following section describes the Assembler options. To avoid ambiguity, the option arguments should immediately follow the option letter with no blanks between them.

3.3.2 Assembler Options

-A

This option indicates that the Assembler should run in absolute mode, generating an absolute object file when the **-B** command line option is given. By default, the Assembler produces a relocatable object file that is subsequently processed by the Motorola DSP Linker.

-B<objfil>

This option specifies that an object file is to be created for Assembler output. <objfil> can be any legal operating system filename, including an optional pathname. The type of object file produced depends on the Assembler operation mode. If the **-A** option is supplied on the command line, the Assembler operates in absolute mode and generates an absolute object (.cld) file. If there is no **-A** option on the command line, the Assembler operates in relative mode and creates a relocatable object (.cln) file. If the **-B** option is not specified, the Assembler will not generate an object file. If no <objfil> is specified, the Assembler will use the basename (filename without extension) of the first filename encountered in the source input file list and append the appropriate file type (.cln or.cld) to the basename. The **-B** option should be specified only once.

Example: ***asm56300 -Bfilter main.asm fft.asm fio.asm***

This example assembles the files main.asm, fft.asm, and fio.asm together to produce the relocatable object file filter.cln.

-D <symbol> <string>

This option replaces all occurrences of <symbol> with <string> in the source files to be assembled.

Example: ***asm56300 -DPOINTS 16 prog.asm***

This example replaces all occurrences of the symbol POINTS in the program prog.asm by the string ‘16’.

-EA<errfil> or **-EW<errfil>**

These options allow the standard error output file to be reassigned on hosts that do not support error output redirection from the command line. <errfil> must be present as an argument, but can be any legal operating system filename, including an optional pathname. The **-EA** option causes the standard error stream to be written to <errfil>; if <errfil> exists, the output stream is appended to the end of the file. The **-EW** option also writes the standard error stream to <errfil>; if <errfil> exists, it will be overwritten.

Example: ***asm56300 -EWerrors prog.asm***

This example redirects the standard output to the file errors. If the file already exists, it will be overwritten.

-F<argfil>

This option indicates that the Assembler should read command line input from <argfil>. <argfil> can be any legal operation system filename, including an optional pathname. <argfil> is a text file containing further options, arguments, and filenames to be passed to the Assembler. The arguments in the file need to be separated only by some form of white space. A semicolon on a line following white space makes the rest of the line a comment.

Example: ***asm56300 -Fopts.cmd***

This example invokes the Assembler and takes the command line options and source filenames from the command file opts.cmd.

-G

This option sends the source file line number information to the object file. This option is valid only in conjunction with the **-B** command line option. The generated line number information can be used by debuggers to provide source-level debugging.

Example: ***asm56300 -B -Gmyprog.asm***

This example assembles the file myprog.asm and sends the source file line number information to the resulting object file myprog.cln.

-I<pathname>

This option causes the Assembler to look in the directory defined by <pathname> for any include file not found in the current directory. <pathname> can be any legal operating system pathname.

Example: ***asm56300 -I\project\ testprog***

This example uses IBM PC pathname conventions, and would cause the Assembler to prefix any include files not found in the current directory with the \project\ pathname.

-L<lstfil>

This option specifies that a listing file is to be created for Assembler output. <lstfil> can be any legal operating system filename, including an optional pathname. If no <lstfil> is specified, the Assembler will use the basename (filename without extension) of the first filename encountered in the source input file list and append .lst to the basename. The **-L** option should be specified only once.

Example: ***asm56300 -L filter.asm gauss.asm***

This example assembles the files filter.asm and gauss.ams together to produce a listing file. Because no filename was given, the output file will be named using the basename of the first source file, in this case filter, and the listing file will be called filter.lst.

-M<pathname>

This option causes the Assembler to look in the directory defined by <pathname> for any macro file not found in the current directory. <pathname> can be any legal operating system pathname.

Example: ***asm56300 -Mfftlib\ trans.asm***

This example uses IBM PC pathname conventions, and would cause the Assembler to look in the fftlib subdirectory of the current directory for a file with the name of the currently invoked marco found in the source file, trans.asm.

-V

This option causes the Assembler to report assembly progress to the standard error output stream.

-Z

This option causes the Assembler to strip symbol information from the absolute load file. Normally symbol information is retained in the object file for symbolic references purposes. This option is only valid with the **-A** and **-B** options.

Note: Multiple options can be used. A typical string might be:

Example: ***asm56300 -A -B -L -G filename.asm***

3.3.3 Assembler Directives

In addition to the DSP56303 instruction set, the assembly programs can contain mnemonic directives which specify auxiliary actions to be performed by the Assembler. These are the Assembler directives. These directives are not always translated into machine language. The following sections briefly describe the various types of Assembler directives.

3.3.3.1 Assembler Significant Characters

There are several one and two character sequences that are significant to the Assembler. The Assembler significant characters are:

;	- Comment delimiter
;;	- Unreported comment delimiter
****	- Line continuation character or macro dummy argument concatenation operator
?	- Macro value substitution operator
%	- Macro hex value substitution operator
^	- Macro local label override operator
“	- Macro string delimiter or quoted string DEFINE expansion character
@	- Function delimiter
*****	- Location counter substitution
++	- String concatenation operator
[]	- Substring delimiter
<<	- I/O short addressing mode force operator
<	- Short addressing mode force operator
>	- Long addressing mode force operator
#	- Immediate addressing mode operator
#<	- Immediate short addressing mode force operator
#>	- Immediate long addressing mode force operator

3.3.3.2 Assembly Control

The directives used for assembly control are:

COMMENT - Start comment lines

DEFINE - Define substitution string

END - End of source program

FAIL - Programmer generated error message

FORCE - Set operand forcing mode

HIMEM - Set high memory bounds

INCLUDE - Include secondary file

LOMEM - Set low memory bounds

MODE - Change relocation mode

MSG - Programmer generated message

ORG - Initialize memory space and location counters

RADIX - Change input radix for constants

RDIRECT - Remove directive or mnemonic from table

SCSJMP - Set structured control branching mode

SCSREG - Reassign structured control statement registers

UNDEF - Undefine DEFINE symbol

WARN - Programmer generated warning

3.3.3.3 Symbol Definition

The directives used to control symbol definition are:

ENDSEC - End section

EQU - Equate symbol to a value

GLOBAL - Global section symbol declaration

GSET - Set global symbol to a value

LOCAL - Local section symbol declaration

SECTION - Start section

SET - Set symbol to a value

XDEF - External section symbol definition

XREF - External section symbol reference

3.3.3.4 Data Definition/Storage Allocation

The directives used to control constant data definition and storage allocation are:

BADDR - Set buffer address

BSB - Block storage bit-reverse

BSC - Block storage of constant

BSM - Block storage modulo

BUFFER - Start buffer

DC - Define constant

DCB - Define constant byte

DS - Define storage

DSM - Define modulo storage

DSR - Define reverse carry storage

ENDBUF - End buffer

3.3.3.5 Listing Control and Options

The directives used to control the output listing are:

LIST - List the assembly

LSTCOL - Set listing field widths

NOLIST - Stop assembly listing

OPT - Assembler options

PAGE - Top of page / size page

PRCTL - Send control string to printer

STITLE - Initialize program subtitle

TABS - Set listing tab stops

TITLE - Initialize program title

3.3.3.6 Object File Control

The directives used for control of the object file are:

COBJ - Comment object code

IDENT - Object code identification record

SYMOBJ - Write symbol information to object file

3.3.3.7 Macros and Conditional Assembly

The directives used for macros and conditional assembly are:

DUP	- Duplicate sequence of source lines
DUPA	- Duplicate sequence with arguments
DUPC	- Duplicate sequence with characters
DUPF	- Duplicate sequence in loop
ENDIF	- End of conditional assembly
ENDM	- End of macro definition
EXITM	- Exit macro
IF	- Conditional assembly directive
MACLIB	- Macro library
MACRO	- Macro definition
PMACRO	- Purge macro definition

3.3.3.8 Structured Programming

The directives used for structured programming are:

.BREAK	- Exit from structured loop construct
.CONTINUE	- Continue next iteration of structured loop
.ELSE	- Perform following statements when .IF false
.ENDF	- End of .FOR loop
.ENDI	- End of .IF condition
.ENDL	- End of hardware loop
.ENDW	- End of .WHILE loop
.FOR	- Begin .FOR loop
.IF	- Begin .IF condition
.LOOP	- Begin hardware loop
.REPEAT	- Begin .REPEAT loop
.UNTIL	- End of .REPEAT loop
.WHILE	- Begin .WHILE loop

3.3.4 Assembling the Example Program

The Assembler is a MS-DOS based program, thus to use the Assembler you will need to exit Windows or open a MS-DOS Prompt Window. To assemble the example program, type ***asm56300 -a -b -l -g example.asm*** in the evm303w directory created by the installation process from **Section 1.3.3**. This will create two additional files: example.cld and example.lst. The example.cld file is the absolute object file of the program, and this is what will be downloaded into the DSP56303. The example.lst file is the listing file and gives full details of where the program and data will be placed in the DSP56303 memory.

3.4 MOTOROLA DSP LINKER

Though not needed for our simple example, the Motorola DSP Linker is also included with the DSP56303EVM. The Motorola DSP Linker is a program that processes relocatable object files produced by the Motorola DSP Assembler, generating an absolute executable file which can be downloaded to the DSP56303. The Motorola DSP Linker is included on the Motorola 3-1/2 inch diskette and can be installed by following the instructions in **Section 1.3.3**. The general format of the command line to invoke the Linker is:

dsplnk [options] <filenames>

where ***dsplnk*** is the name of the Motorola DSP Linker program, and ***<filenames>*** is a list of the relocatable object files to be linked. The following section describes the Linker options. To avoid ambiguity, the option arguments should immediately follow the option letter with no blanks between them.

3.4.1 Linker Options

-A

This option auto-aligns circular buffers. Any modulo or reverse-carry buffers defined in the object file input sections are relocated independently in order to optimize placement in memory. Code and data surrounding the buffer are packed to fill the space formerly occupied by the buffer and any corresponding alignment gaps.

Example: ***dsplnk -A myprog.cln***

This example links the file myprog.cln and optimally aligns any buffers encountered in the input.

-B<objfil>

This option specifies that an object file is to be created for Linker output. <objfil> can be any legal operating system filename, including an optional pathname. If no filename is specified, or if the **-B** option is not present, the Linker will use the basename (filename without extension) of the first filename encountered in the input file list and append .cld to the basename. If the **-I** option is present (see below), an explicit filename must be given. This is because if the Linker followed the default action, it possibly could overwrite one of the existing input files. The **-B** option should be specified only once. If the file named in the **-B** option already exists, it will be overwritten.

Example: ***dsplnk -Bfilter.cld main.cln fft.cln fio.cln***

In this example, the files main.cln, fft.cln, and fio.cln are linked together to produce the absolute executable file filter.cld.

-EA<errfil> or **-EW<errfil>**

These options allow the standard error output file to be reassigned on hosts that do not support error output redirection from the command line. <errfil> must be present as an argument, but can be any legal operating system filename, including an optional pathname. The **-EA** option causes the standard error stream to be written to <errfil>; if <errfil> exists, the output stream is appended to the end of the file. The **-EW** option also writes the standard error stream to <errfil>; if <errfil> exists it will be overwritten.

Example: ***dsplnk -EWerrors myprog.cln***

This example redirects the standard error output to the file errors. If the file already exists, it will be overwritten.

-F<argfil>

This option indicates that the Linker should read command line input from <argfil>. <argfil> can be any legal operating system filename, including an optional pathname. <argfil> is a text file containing further options, arguments, and filenames to be passed to the Linker. The arguments in the file need be separated only by some form of white space. A semicolon on a line following white space makes the rest of the line a comment.

Example: ***dsplnk -Fopts.cmd***

This example invokes the Linker and takes command line options and input filenames from the command file opts.cmd.

-G

This option sends source file line number information to the object file. The generated line number information can be used by debuggers to provide source-level debugging.

Example: ***dsplnk -B -Gmyprog.cln***

This example links the file myprog.cln and sends source file line number information to the resulting object file myprog.cld.

-I

The Linker ordinarily produces an absolute executable file as output. When the **-I** option is given, the Linker combines the input files into a single relocatable object file suitable for reprocessing by the Linker. No absolute addresses are assigned and no errors are issued for unresolved external references. Note that the **-B** option must be used when performing incremental linking in order to give an explicit name to the output file. If the filename were allowed to default, it could overwrite an existing input file.

Example: ***dsplnk -I -Bfilter.cln main.cln fft.cln fio.cln***

In this example, the files main.cln, fft.cln, and fio.cln are combined to produce the relocatable object file filter.cln.

-L<library>

The Linker ordinarily processes a list of input files which each contain a single relocatable code module. If the **-L** option is encountered, the Linker treats the following argument as a library file and searches the file for any outstanding unresolved references. If a module is found in the library that resolves an outstanding external reference, the module is read from the library and included in the object file output. The Linker continues to search a library until all external references are resolved or no more references can be satisfied within the current library. The Linker searches a library only once, when it is encountered on the command line. Therefore, the position of the **-L** option on the command line is significant.

Example: ***dsplnk -B filter main fir -Lio***

This example illustrates linking with a library. The files main.cln and fir.cln are combined with any needed modules in the library io.lib to create the file filter.cld.

-M<mapfil>

This option indicates that a map file is to be created. <mapfil> can be any legal operating system filename, including an optional pathname. If no filename is specified, the Linker will use the basename (filename without extension) of the first filename encountered in the input file list and append .map to the basename. If the **-M** option is not specified, then the Linker will not generate a map file. The **-M** option should be specified only once. If the file named in the **-M** option already exists, it will be overwritten.

Example: ***dsplnk -M filter.cln gauss.cln***

In this example, the files filter.cln and gauss.cln are linked together to produce a map file. Because no filename was given with the **-M** option, the output file will be named using the basename of the first input file, in this case filter. The map file will be called filter.map.

-N

The Linker considers case significant in symbol names. When the **-N** option is given the Linker ignores case in symbol names; all symbols are mapped to lower case.

Example: ***dsplnk -N filter.cln fft.cln fio.cln***

In this example, the files filter.cln, fft.cln, and fio.cln are linked to produce the absolute executable file filetr.cld. All symbol references are mapped to lower case.

-O<mem>[<ctr>][<map>]:<origin>

By default the Linker generates instructions and data for the output file beginning at absolute location zero for all DSP memory spaces. This option allows the programmer to redefine the start address for any memory space and associated location counter. <mem> is one of the single-character memory space identifiers (X, Y, L, P). The letter may be upper or lower case. The optional <ctr> is a letter indicating the High (H) or Low (L) location counters. If no counter is specified the default counter is used. <map> is also optional and signifies the desired physical mapping for all relocatable code in the given memory space. It may be I for Internal memory, E for External memory, R for ROM, A for Port A, and B for Port B. If <map> is not supplied, then no explicit mapping is presumed. The <origin> is a hexadecimal number signifying the new relocation address for the given memory space. The **-O** option may be specified as many times as needed on the command line. This option has no effect if incremental linking is being done (see the **-I** option).

Example: ***dsplnk -Ope:200 myprog -Lmylib***

This example initializes the default P memory counter to hex 200 and maps the program space to external memory.

-P<pathname>

When the Linker encounters input files, the current directory (or the directory given in the library specification) is first searched for the file. If it is not found and the **-P** option is specified, the Linker prefixes the filename (and optional pathname) of the file specification with <pathname> and searches the newly formed directory pathname for the file.The pathname must be a legal operating system pathname. The **-P** option may be repeated as many times as desired.

Example: ***dsplnk -P\project\ testprog***

This example uses IBM PC pathname conventions, and would cause the Linker to prefix any library files not found in the current directory with the \project\ pathname.

-R<ctlfil>

This option indicates that a memory control file is to be read to determine the placement of sections in DSP memory and other Linker control functions. <ctlfil> can be any legal operating system filename, including an optional pathname. If a pathname is not specified, an attempt will be made to open the file in the current directory. If no filename is specified, the Linker will use the basename (filename without extension) of the first filename encountered in the link input file list and append .ctl to the basename. If the **-R** option is not specified, then the Linker will not use a memory control file. The **-R** option should be specified only once.

Example: ***dsplnk -Rproj filter.cln gauss.cln***

In this example, the files filter.cln and gauss.cln are linked together using the memory file proj.ctl.

-U<symbol>

This option allows the declaration of an unresolved reference from the command line. <symbol> must be specified. This option is useful for creating an undefined external reference in order to force linking entirely from a library.

Example: ***dsplnk -Ustart -Lproj.lib***

This example declares the symbol start undefined so that it will be resolved by code within the library proj.lib.

-V

This option causes the Linker to report linking progress (beginning of passes, opening and closing of input files) to the standard error output stream. This is useful to insure that link editing is proceeding normally.

Example: ***dsplnk -V myprog.cln***

This example links the file myprog.cln and sends progress lines to the standard error output.

-X<opt>[,<opt>,...,<opt>]

The **-X** option provides for link time options that alter the standard operation of the Linker. The options are described below (* means default). All options may be preceded by NO to reverse their meaning. The **-X<opt>** sequence can be repeated for as many options as desired.

Option	Meaning
ABC*	Perform address bounds checking
AEC*	Check form of address expressions
ASC	Enable absolute section bounds checking
CSL	Cumulate section length data
ESO	Do not allocate memory below ordered sections
OVLP	Warn on section overlap
RO	Allow region overlap
RSC*	Enable relative section bounds checking
SVO	Preserve object file on errors
WEX	Add warning count to exit status

Example: ***dsplnk -XWEX filter.cln fft.cln fio.cln***

This example allows the Linker to add the warning count to the exit status so that a project build will abort on warnings as well as errors.

-Z

This option allows the Linker to strip source file line number and symbol information from the output file. Symbol information normally is retained for debugging purposes. This option has no effect if incremental linking is being done (see the **-I** option).

Example: ***dsplnk*** -Zfilter.cln fft.cln fio.cln

In this example, the files filter.cln, fft.cln, and fio.cln are linked to produce the absolute object file filter.cln. The output file will contain no symbol or line number information.

3.4.2 Linker Directives

Similar to the Assembler directives, the Linker includes mnemonic directives which specify auxiliary actions to be performed by the Linker. The following sections briefly describe the Linker directives.

BALIGN -Auto-align circular buffers

BASE - Set region base address

IDENT - Object module identification

INCLUDE - Include directive file

MAP - Map file format control

MEMORY - Set region high memory address

REGION - Establish memory region

RESERVE - Reserve memory block

SBALIGN - Auto-align section buffers

SECSIZE - Pad section length

SECTION - Set section base address

SET - Set symbol value

SIZSYM - Set size symbol

START - Establish start address

SYMBOL - Set symbol value

3.5 INTRODUCTION TO THE DEBUGGER SOFTWARE

This section will give a brief introduction of the Domain Technologies Debugger, detailing only that which is required to work through this example. Full details of the Debugger and an informative tutorial can be found in the Debug-56K Manual. The Domain Technologies Debugger is a software development system for the DSP56303. The Domain Technologies Debugger is included with the DSP56303EVM on the Domain Technologies 3-1/2 inch diskette and can be installed by following the instructions in **Section 1.3.3**. To invoke the Debugger, double-click on the icon labelled EVM303W in the EVM56303 program group that was created when the Debugger was installed.

The Debugger display will be similar to **Figure 3-2** with the screen divided into four windows: the command window, the data window, the unassembly window, and the registers window. The command window is the window selected, which means that key strokes will be placed in the command window. As shown in the demonstration from **Section 2**, the command window is where commands are entered. The data window is used to display DSP56303 data. The unassembly window is used to display the DSP56303 programs. The next instruction to be executed will be highlighted. The registers window shows the contents of the DSP56303 internal registers.

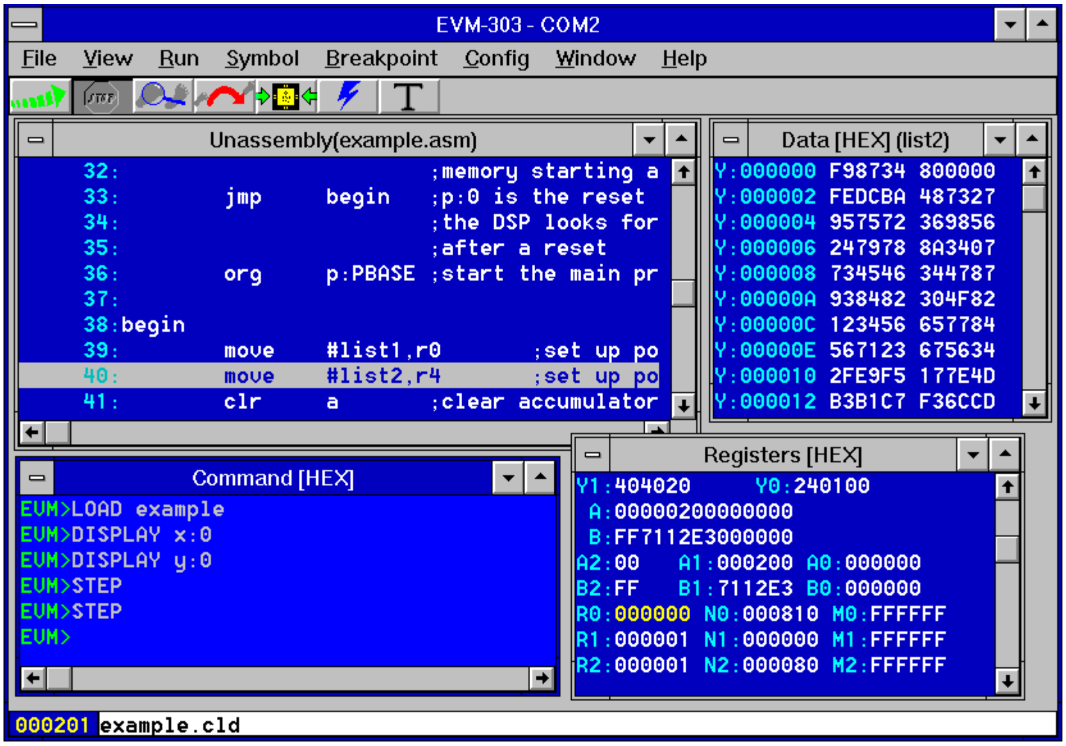


Figure 3-2 Example Debugger Window Display

When the command window is selected as in **Figure 3-2**, the tool-bar at the top of the screen contains buttons for the most often used commands. From right to left the commands are: go, stop, step, jump, automatic update, reset and radix. The go button runs the DSP56303 from the program counter. The stop button stops the DSP56303. The step button executes a single instruction. The jump button is similar to the step button, except that subroutines are treated as one instruction. The automatic update button turns the automatic screen update mode on, so that the DSP56303 is interrupted periodically to update the data and registers windows. The reset button resets the DSP56303. The radix button can be used to change the radix of the selected window. Other buttons will appear when other windows are selected, and the function of these buttons can be found in the Debug-56K Manual.

3.6 RUNNING THE PROGRAM

To load the example program developed above into the Debugger, click in the command window and type ***load example***. The instruction at line 33 will be highlighted in the unassembly window as this will be the first instruction to be executed. However, before we start to execute the program, we should check that the values we expect to be in data memory are there. To do this, type ***display x:0*** and ***display y:0***. The data will be displayed in the data window.

To step through the program, type ***step*** at the command window prompt. As a shortcut, you can click on the step button or you can type the start of the command and press the space bar, and the debugger will complete the remainder of the command. To repeat the last command, simply press return. As you step through the code, you will see the registers in the registers window being changed by the instructions. After each cycle, any register that has been changed will be brightened. Once you have stepped through the program, ensure that the program has executed correctly by checking that the result in accumulator a is: $FE 9F2051 6DFCC2.

Stepping through the program like this is good for short programs, but it is impractical for large complex programs. The way to debug large programs is to set breakpoints. These are user-defined points at which execution of the code will stop, allowing the user to step through the section of interest. To set a breakpoint in the example to check that the values in r0 and r4 are correct before the do loop, type ***break p:$106*** in the command window. You will see the line before the loop brighten in the unassembly window, indicating the breakpoint has been set. To point the DSP56303 back to the start point of the program, type ***change pc 0***. This changes the program counter such that it is pointing to the reset vector. To start the program running type ***go*** or click on the go button. The DSP56303 will stop when it reaches the breakpoint, and you will be able to step through the remainder of the code.

To exit the Debugger, type ***quit*** at the command prompt.

SECTION 4

DSP56303EVM TECHNICAL SUMMARY

4.1 DSP56303EVM DESCRIPTION AND FEATURES

An overview description of the DSP56303EVM is provided in the DSP56303EVM Product Information document (order number DSP56303EVMP/D) included with this kit. The main features of the DSP56303EVM include the following:

- DSP56303 24-Bit Digital Signal Processor
- Memory
- 16-bit CD-quality audio codec
- Command Converter

4.2 DSP56303 DESCRIPTION

A full description of the DSP56303, including functionality and user information is provided in the following documents included as a part of this kit:

- **DSP56303 Technical Data (order number DSP56303/D):** Provides features list and specifications including signal descriptions, DC power requirements, AC timing requirements, and available packaging
- **DSP56303 User's Manual (order number DSP56303UM/AD):** Provides an overview description of the DSP and detailed information about the on-chip components including the memory and I/O maps, peripheral functionality, and control and status register descriptions for each subsystem

Note: The *DSP56303 User's Manual* is currently not available. The DSP56303 Product Specification, Revision 1.02 is being supplied with this kit as a replacement document.

- **DSP56300 Family Manual (order number DSP56300FM/AD):** Provides a detailed description of the core processor including internal status and control registers and a detailed description of the family instruction set.

Refer to these documents for detailed information about chip functionality and operation.

Note: A detailed list of known chip errata is also provided with this kit. Refer to the *DSP56303 Chip Errata* document for information that has changed since the publication of the reference documentation listed above. The latest version can be obtained on the Motorola DSP worldwide web site at *http://www.motorola-dsp.com*.

4.3 MEMORY

The DSP56303EVM uses the following memory:

- 32 K × 24-bit Fast Static RAM (SRAM) for expansion memory
- 64 K × 8-bit Flash Programmable Erasable Read-Only Memory (PEROM) for stand-alone operation

Refer to **Figure 4-1** for the location of the SRAM and PEROM on the DSP56303EVM. **Figure 4-2** shows a functional block diagram of the DSP56303EVM including the memory devices.

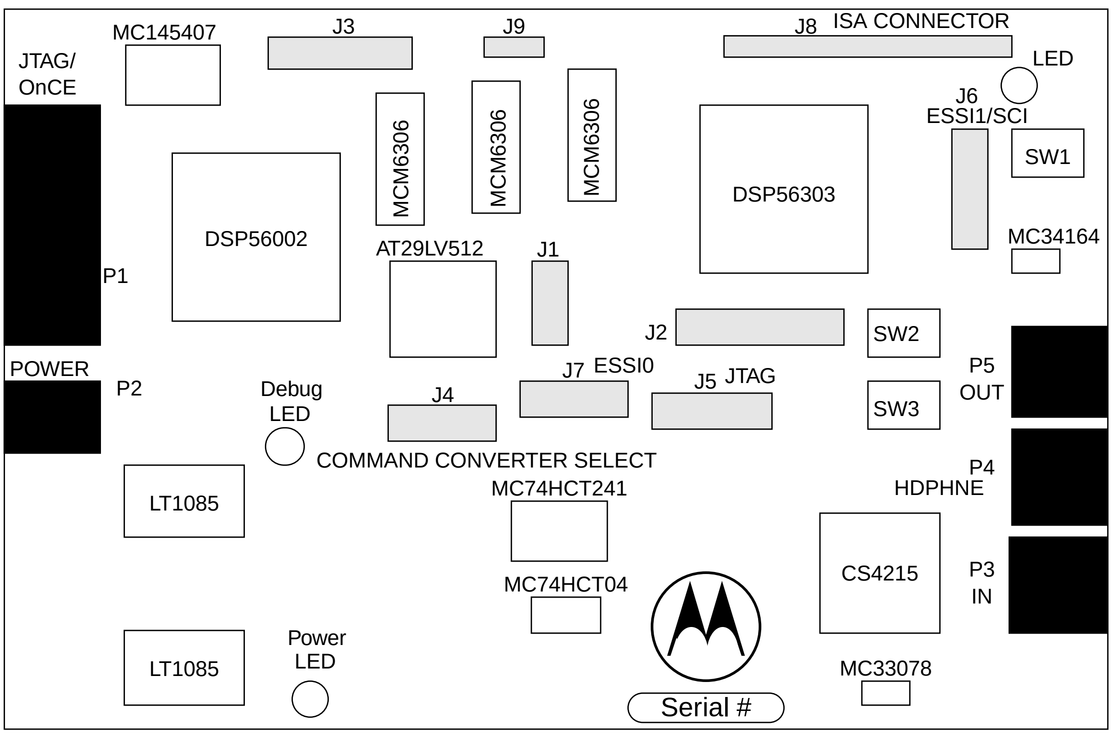


Figure 4-1 DSP56303EVM Component Layout

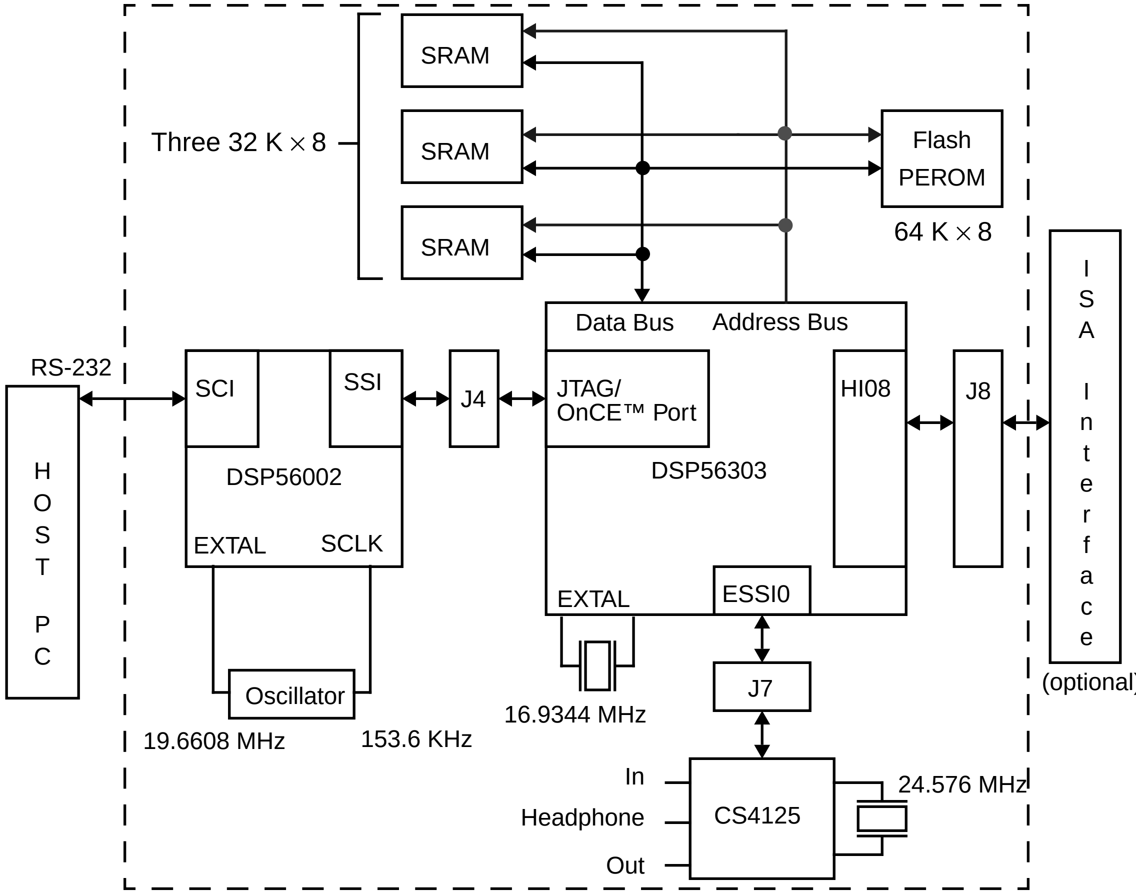


Figure 4-2 DSP56303EVM Functional Block Diagram

4.3.1 SRAM

The DSP56303EVM uses three banks of 32 K × 8-bit fast Static RAM (Motorola MCM6306, labelled U4, U5, and U6) for memory expansion.The MCM6306 uses a single 3.3 volt power supply and has an access time of 15 ns. The following sections give more details on the operation of the SRAM.

4.3.1.1 SRAM Connections

The basic connection for the SRAM is shown in **Figure 4-3**.

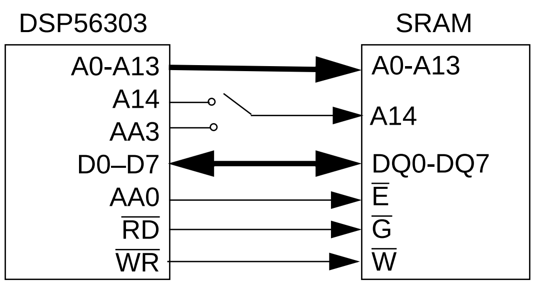


Figure 4-3 SRAM Connections to the DSP56303

The data input/output pins DQ0-DQ7 for the first bank of SRAM are connected to the DSP56303 D0–D7 pins. Similarly, pins D8–D15 and D16–D23 of the DSP56303 are connected to the second and third bank of SRAM. The SRAM Write ($\overline{W}$) and output ($\overline{G}$) enable lines are connected to the DSP56303 Write ($\overline{WR}$) and Read ($\overline{RD}$) lines, respectively. The SRAM chip Enable ($\overline{E}$) is generated by the DSP56303 Address Attribute 0 (AA0). The SRAM activity is controlled by AA0 and the corresponding Address Attribute Register 0 (AAR0). The SRAM Address input pins A0–A13 are connected to the respective Port A address pins of the DSP. The SRAM address line A14 may be connected to the DSP56303 Address Attribute 3 (AA3) or to the DSP56303 address pin A14. If the jumper on J9 connects pin 1 and 2 as in **Figure 4-4**a, then the SRAM address line A14 is connected to AA3 on the DSP56303. If the jumper on J9 connects pin 2 and 3 as in **Figure 4-4**b, then the SRAM address line A14 is connected to A14 on the DSP56303.

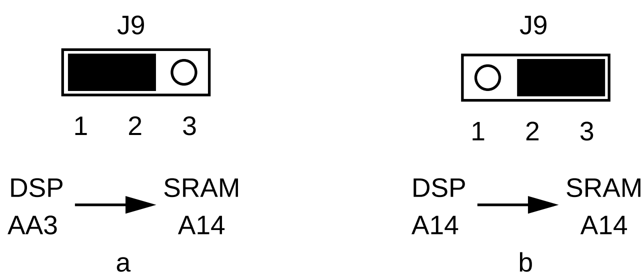


Figure 4-4 Alternate Configurations for J9

Currently, the jumper on J9 MUST connect pins 2 and 3 so that the SRAM address line A14 is connected to the DSP56303 address line A14. This configuration selects a unified memory map of 32 K words. The unified memory does not contain partitioned X data, Y data, and program memory. Thus, access to P:$1000, X:$1000, and Y:$1000 are treated as accesses to the same memory cell and 48-bit long memory data moves are not possible to or from the external SRAM.

4.3.1.2 Example: Programming AAR0

As mentioned above, the SRAM activity is controlled by the DSP56303 pin AA0 and the corresponding AAR0. AAR0 controls the external access type, the memory type, and which external memory addresses access the SRAM. **Figure 4-5** shows the memory map that is attained with the AAR0 settings described in this example. (In this example, the Memory Switch bit in the Operating Mode Register (OMR) is set and the sixteen-bit compatibility bit in the status register is cleared.) In **Figure 4-5**, the SRAM responds to the 32 K of X and Y data memory addresses between $010000 and $017FFF. However, with the unified memory map, accesses to the same external memory location are treated as accesses to the same memory cell.

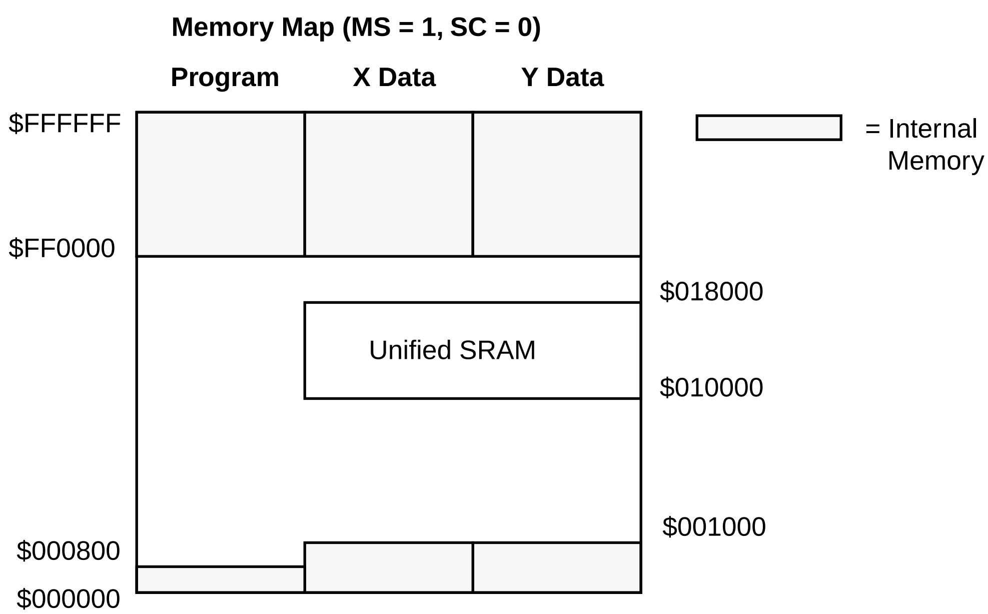


Figure 4-5 Example Memory Map with the Unified External Memory

The settings of AAR0 for this example are shown in **Figure 4-6**. The external Access Type bits (BAT1 and BAT0) are set to 0 and 1, respectively, to denote SRAM access. The Address Attribute Polarity bit (BAAP) is cleared to define AA0 as active low. Address multiplexing is not supported in the DSP56303; therefore, the address multiplexing bit BAM is cleared. Packing is not needed with the SRAM; thus, the packing enable bit BPAC is cleared to disable this option.

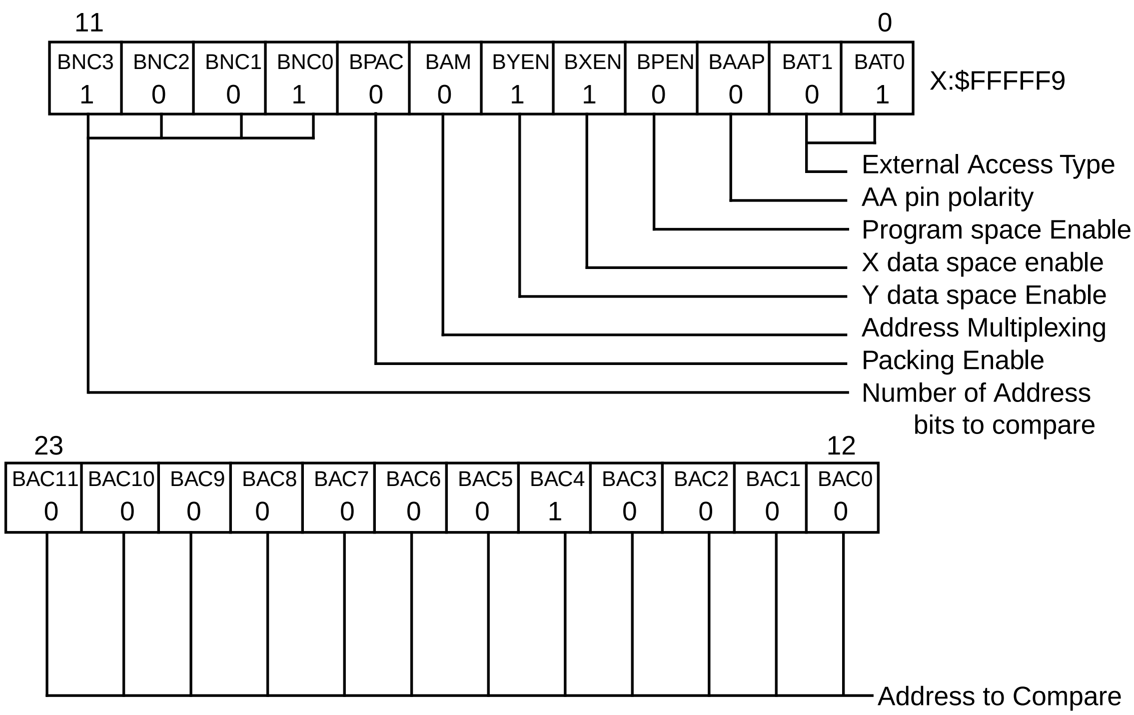


Figure 4-6 Address Attribute Register AAR0

The P, X data, and Y data space Enable bits (BPEN, BXEN, and BYEN) define whether the SRAM is activated during external P, X data, or Y data space accesses respectively. For this example, the BXEN and BYEN bits are set, and BPEN is cleared to allow the SRAM to respond to X and Y data memory accesses only.

The number of address bits to compare BNC(3:0) and the address to compare bits BAC(11:0) determine which external memory addresses access the SRAM. The BNC bits define the number of upper address bits that are compared between the BAC bits and the external address to determine if the SRAM is accessed. For this example, the SRAM is assigned to respond to addresses between $010000 and $017FFF. Thus, the BNC bits are set to $9 and the BAC bits are set to $010. If the nine Most Significant Bits of the external address are 000000010, the SRAM is accessed.

4.3.2 Flash PEROM

The DSP56303EVM uses an Atmel AT29LV512 chip (U7) to provide a 64 K $\times$ 8 bit CMOS Flash PEROM for stand-alone operation (i.e., operation without accessing the DSP56303 through the OnCE port). The AT29LV512 uses a 3.3 V power supply and has a read access time of 200 ns.

4.3.2.1 Flash PEROM Connections

The basic connection for the Flash PEROM is shown in **Figure 4-7**.

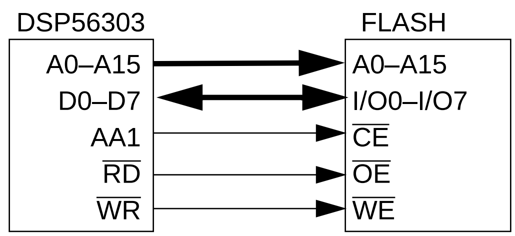


Figure 4-7 Flash PEROM Connections

The Flash PEROM Address input pins (A0–A15) are connected to the respective Port A address pins on the DSP. The Flash PEROM data Input/Output pins I/O0–I/O7 are connected to the DSP56303 D0–D7 pins. The Flash Write Enable ($\overline{WE}$) and Output Enable ($\overline{OE}$) lines are connected to the DSP56303 Write ($\overline{WR}$) and Read ($\overline{RD}$) enable lines, respectively. Address Attribute 1 (AA1) generates the Flash chip enable $\overline{CE}$.

4.3.2.2 Programming for Stand-Alone Operation

The Flash PEROM allows the DSP56303EVM to work in stand-alone operation. The DSP56303 mode pins determine the chip operating mode and start-up procedure when the DSP56303 exits the Reset state. The switch at SW1 resets the DSP56303 by asserting and then clearing the $\overline{RESET}$ pin of the DSP56303. The mode pins MODA, MODB, MODC, and MODD are sampled as the DSP56303 exits the Reset state.The mode pins for the DSP56303EVM are set so that the DSP56303 will boot from the Flash PEROM after reset (Mode 1: MODA is set, MODB and MODC are cleared, and MODD is at either state).

The *flash.asm* file, included on the DSP56303EVM diskette, allows the user to program the Flash PEROM with the code to be run by the DSP56303 in stand-alone operation. To prepare to program, the Flash PEROM stand-alone code should be written to program memory. After the stand-alone code is written into program memory, the size of the stand-alone code in words should be loaded into the register R1, and the address of the first word of stand-alone code should be loaded into the register R0. When invoked, flash.asm reads the stand-alone code from program memory starting at the address pointed to by R0 and ending at the location defined by the value in R1, and writes the stand-alone code into the Flash PEROM. When the DSP56303 is reset, the internal bootstrap program will run. The bootstrap program reads the stand-alone code from the Flash PEROM and writes it to program memory. The bootstrap program then begins execution of the stand-alone code.

4.4 AUDIO CODEC

The DSP56303EVM analog section uses Crystal Semiconductor's CS4215 for two channels of 16 bit A/D conversion and two channels of 16-bit D/A conversion. Refer to **Figure 4-1** on page 4-4 for the location of the codec on the DSP56303EVM and to **Figure 4-2** on page 4-5 for a functional diagram of the codec within the evaluation module. The codec is very flexible, offering software selectable sampling frequencies of 8, 9.6, 16, 32, and 48 kHz. Other frequencies are achievable by changing the crystal or by adding a second crystal to the codec XTAL2I and XTAL2O pins. The CS4215 uses a 5 volt power supply. Thus, Motorola's MC74HCT241A is used to convert the voltage levels to and from the 3.3 volt DSP56303. Refer to the CS4215 data sheet included with this kit for more information.

The ada_init.asm and echo.asm files included on the Motorola diskette give examples on how to program the codec. The ada_init.asm file contains the initialization code for the codec and the basic interrupt service routines for the ESSI transmit and receive interrupts. The echo.asm file is an example program that moves audio through the codec and adds a noticeable echo to the audio. The echo.asm file uses the ada_init.asm file. The ada_init.asm and echo.asm files have been set up such that the parameters can be changed by the user easily by changing one of a few control words.

The codec is connected to the DSP56303 ESSI0 through the shorting jumpers on J7 shown in **Figure 4-1** on page 4-4. By removing these jumpers, the user has full access to the ESSI0 pins of the DSP56303. The following sections describe the connections for the analog and digital sections of the codec.

4.4.1 Codec Analog Input/Output

The DSP56303EVM contains 1/8-inch stereo jacks for stereo input, output, and headphones. **Figure 4-8** shows the analog circuitry of the codec.

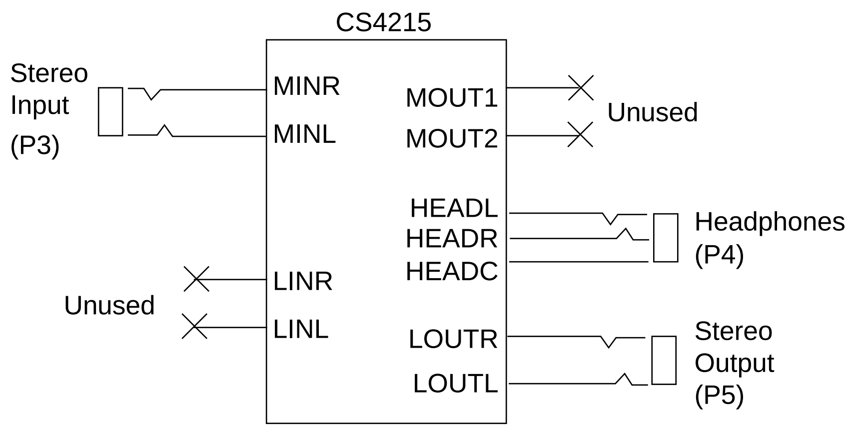


Figure 4-8 Codec Analog Input/Output Diagram

The stereo jack labelled P3/IN on the DSP56303EVM connects to the codec left and right microphone inputs MINL and MINR through an attenuating buffer stage provided by Motorola's MC33078 dual op-amp at U10. Standard line level inputs are 2 V_P and the codec requires that input levels be limited to 1 V_P. Thus, the buffer stage forms a 6 dB attenuator. Additional internal amplifiers with a programmable 20 dB gain block are provided for the microphone inputs. The 20 dB gain block may be disabled using the control mode of the codec.

The analog outputs of the codec are routed via an attenuator to a pair of line outputs, to a pair of headphone outputs, and to a pair of mono monitor speaker outputs. The Mono speaker Outputs (MOUT1 and MOUT2) are not used. The Headphone outputs (HEADL and HEADR) are connected to the stereo jack labelled P4/HDPHNE on the DSP56303EVM, which permits direct connection of stereo headphones to the DSP56303EVM. The Headphone Common return (HEADC) is the return path for large currents when driving headphones from HEADL and HEADR.

The Line Outputs (LOUTR and LOUTL) provide the output analog signal through the stereo jack labelled P5/OUT on the DSP56303EVM. This jack can be selected in software to provide output swings of 2 V_P or 2-8 V_P. Refer to the CS4215 data sheet for technical details of the programming steps required to choose the output voltage swing.

4.4.2 Codec Digital Interface

Figure 4-9 shows the digital interface to the codec.

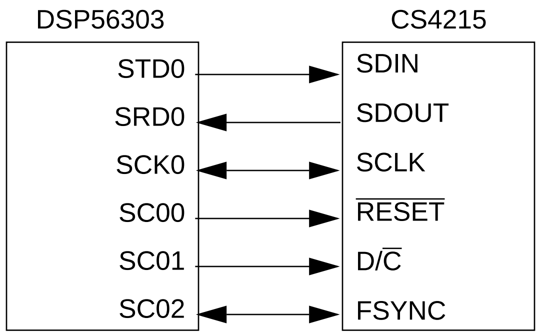


Figure 4-9 Codec Digital Interface Connections

The serial interface of the codec transfers digital audio data and control data into and out of the device. The codec communicates with the DSP56303 through the ESSI0, which consists of independent transmitter and receiver sections.

On the DSP56303 side, the Serial Transmit Data (STD0) pin transmits data to the codec. The Serial Receive Data 0 (SRD0) pin receives data from the codec. These two pins are connected to the codec Serial Data Input (SDIN) and Serial Data Output (SDOUT) pins, respectively. The DSP56303 bidirectional Serial Clock (SCK0) pin provides the serial bit rate clock for the ESSI0 interface. It is connected to the codec Serial Clock (SCLK) pin. Data is transmitted on the rising edge of SCLK and is received on the falling edge of SCLK.

The DSP56303 Serial Control 0 (SC00) pin is programmed to control the codec Reset signal $\overline{\text{RESET}}$. Serial Control 1 (SC01) pin is programmed to control the codec Data / Control (D / $\overline{\text{C}}$) select input pin . When D / $\overline{\text{C}}$ is low, SDIN and SDOUT contain control information. When D / $\overline{\text{C}}$ is high, SDIN and SDOUT contain data information. The Serial Control 2 (SC02) pin is connected to the codec Frame Sync (FSYNC) signal. A rising edge on FSYNC indicates that a new frame is about to start. FSYNC may be an input to the codec, or it may be an output from the codec.

4.4.3 Codec Clock

Figure 4-10 shows the clock generation diagram.

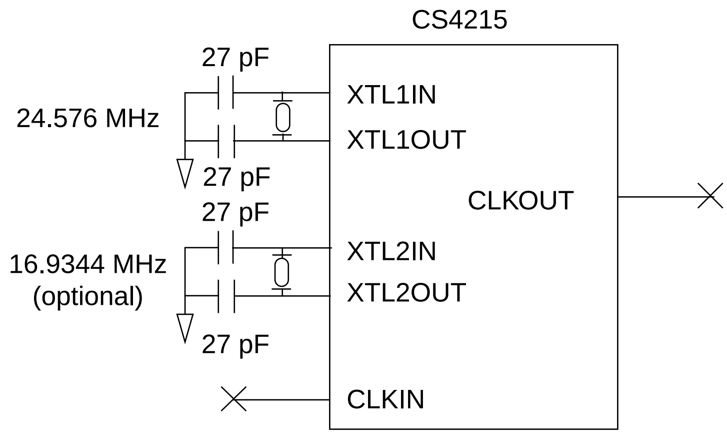


Figure 4-10 Codec Clock Generation Diagram

Two external crystals may be attached to the XTL1IN, XTL1OUT, XTL2IN, and XTL2OUT pins. The XTAL1IN oscillator is intended for use at 24.576 MHz and the XTAL2IN oscillator is intended for use at 16.9344 MHz, although other frequencies may be used. Refer to the CS4215 data sheet for information regarding selection of the correct clock source and divide ratios.

The codec on the DSP56303EVM is driven by a 24.576 MHz crystal between XTL1IN and XTL1OUT. The 24.576 MHz oscillator provides the master clock to run the codec. FSYNC and SCLK must be synchronous to this master clock. The external Clock Input (CLKIN), which is provided for potential use with an external AES/EBU receiver or an already existing system clock, is not used. The master Clock Output (CLKOUT) is also not used.

4.5 COMMAND CONVERTER

The DSP56303EVM uses Motorola's DSP56002 to perform OnCE/JTAG command conversion. The DSP56002 Serial Communications Interface (SCI) communicates with the host PC through an RS-232 connector. The DSP56002 SCI receives commands from the host PC. The set of commands may include read data, write data, reset OnCE, reset DSP56303 (the HA2 pin of the DSP56002 is then used to reset the DSP56303), request OnCE, or release OnCE. The DSP56002 command converter software interprets the commands received from the PC and sends a sequence of instructions to the DSP56303 OnCE/JTAG port. The DSP56303 may then continue to receive data or it may transmit data back to the DSP56002. The DSP56002 sends a reply to the host PC to give status information. The set of replies may include acknowledge good, acknowledge bad, in Debug mode, out of Debug mode, or data read. When the DP56303 is in the Debug state, the red Debug LED (D6) is lluminated.

The DSP56002 is connected to the DSP56303 OnCE port through the shorting jumpers on J4. By removing the jumpers, the user has full access to the OnCE/JTAG pins of the DSP56303. Refer to **Figure 4-1** on page 4-4 for the location of J4 on the DSP56303EVM and to **Figure 4-2** on page 4-5 for a functional diagram. **Figure 4-11** shows the RS-232 serial interface diagram.

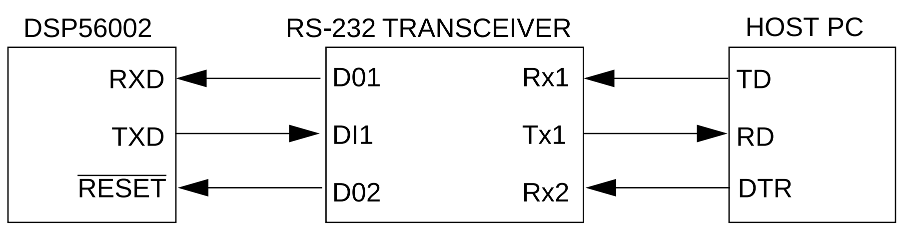
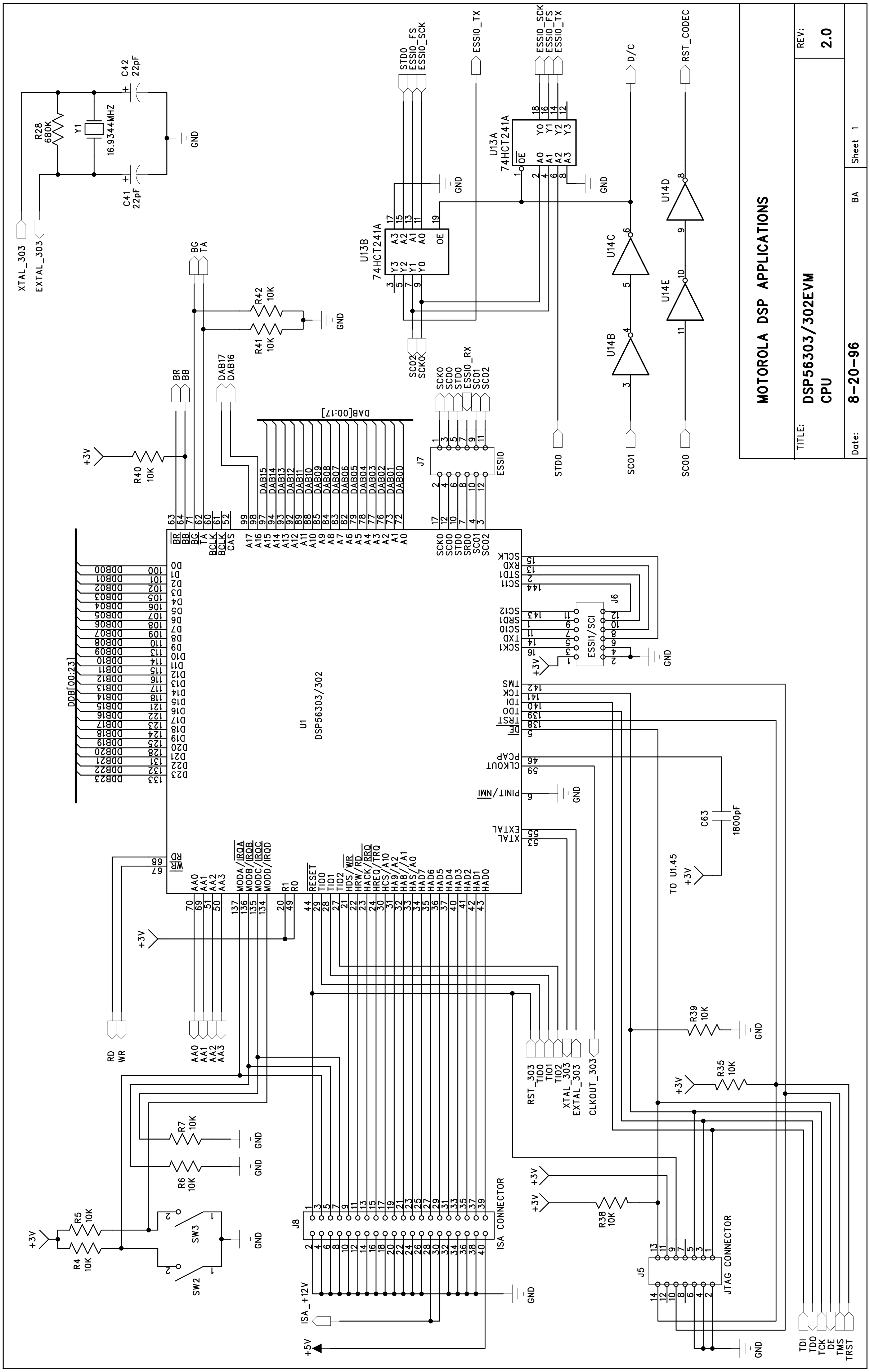


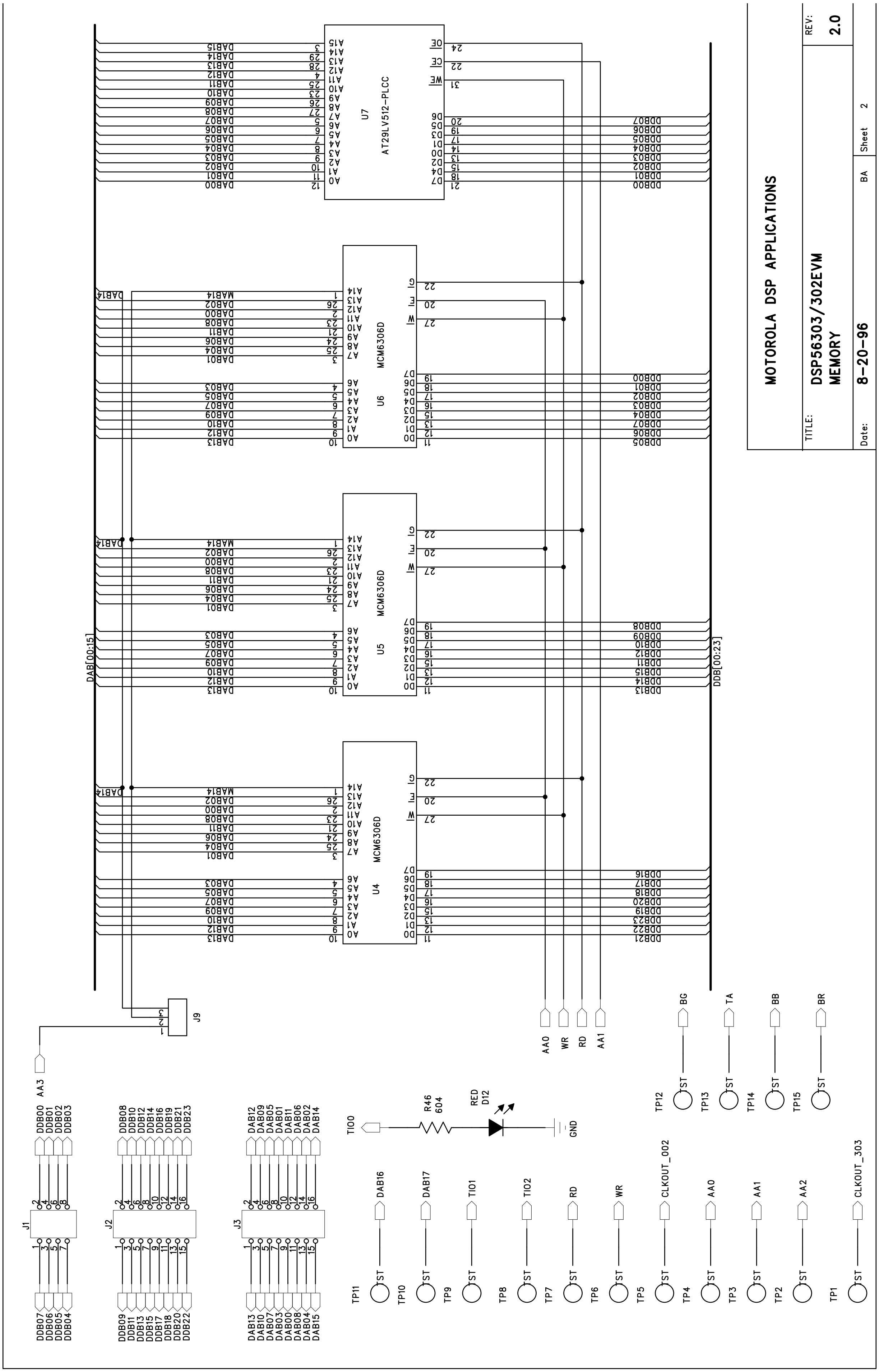
Figure 4-11 RS-232 Serial Interface

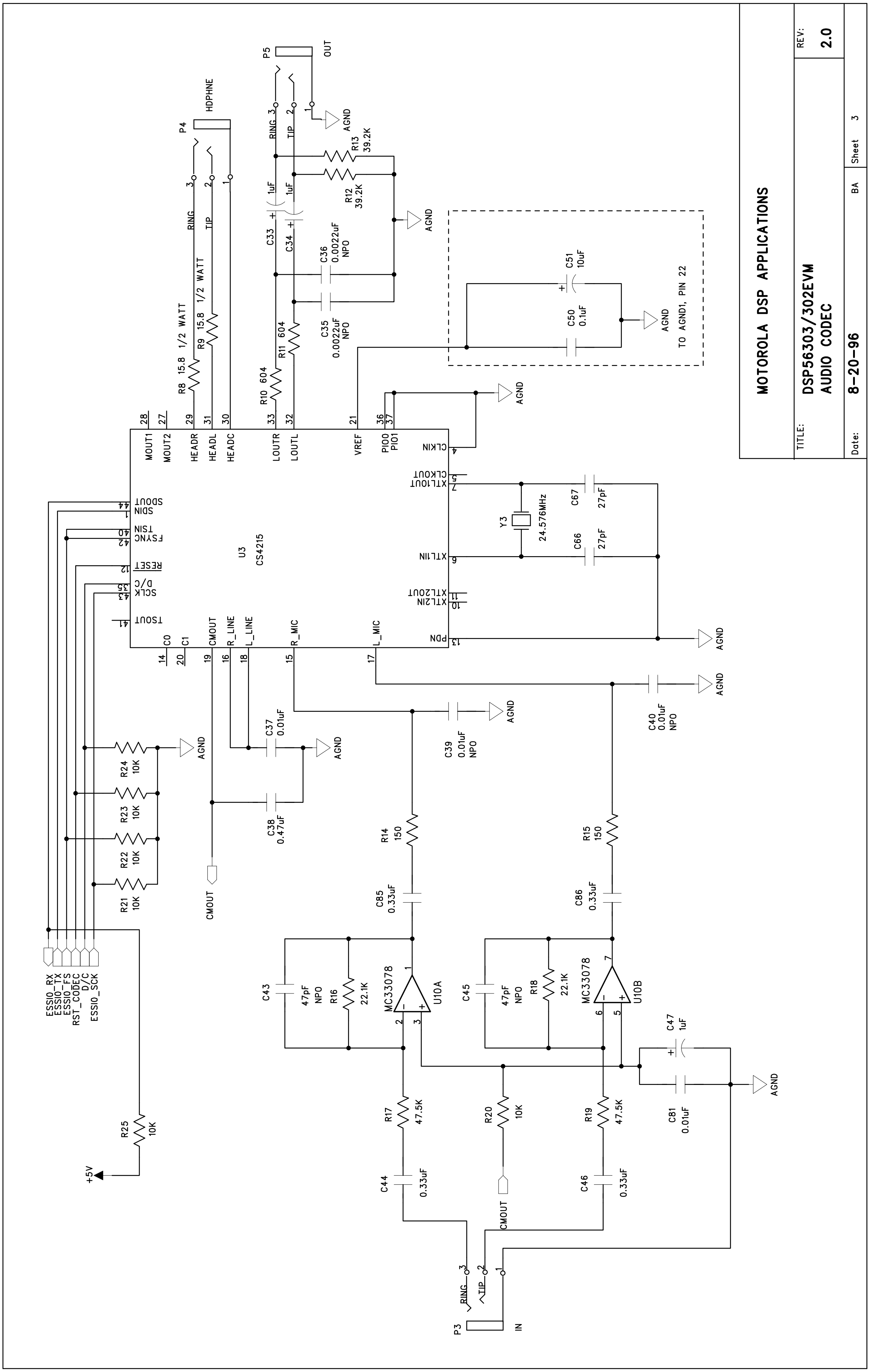
Motorola's 5 Volt-Only Driver/Receiver MC145407 is used to transmit the signals between the host PC and the DSP56002. Serial data is transmitted from the host PC Transmitted Data (TD) signal and received on the DSP56002 Receive Data (RXD) pin. Serial data is similarly transmitted from the DSP56002 Transmit Data (TXD) signal and received on the host PC Received Data (RD) signal. The Data Terminal Ready (DTR) pin asserts the $\overline{\text{RESET}}$ pin of the DSP56002. As an option, the DSP56303EVM 14-pin JTAG connector at J5 allows the user to connect an ADS command converter card directly to the DSP56303EVM if the DSP56002 command converter software is not used (J4 jumpers removed). Pin 8 has been removed from J5. The JTAG cable from the ADS command converter is similarly keyed so that the cable cannot be connected to the DSP56303EVM incorrectly.

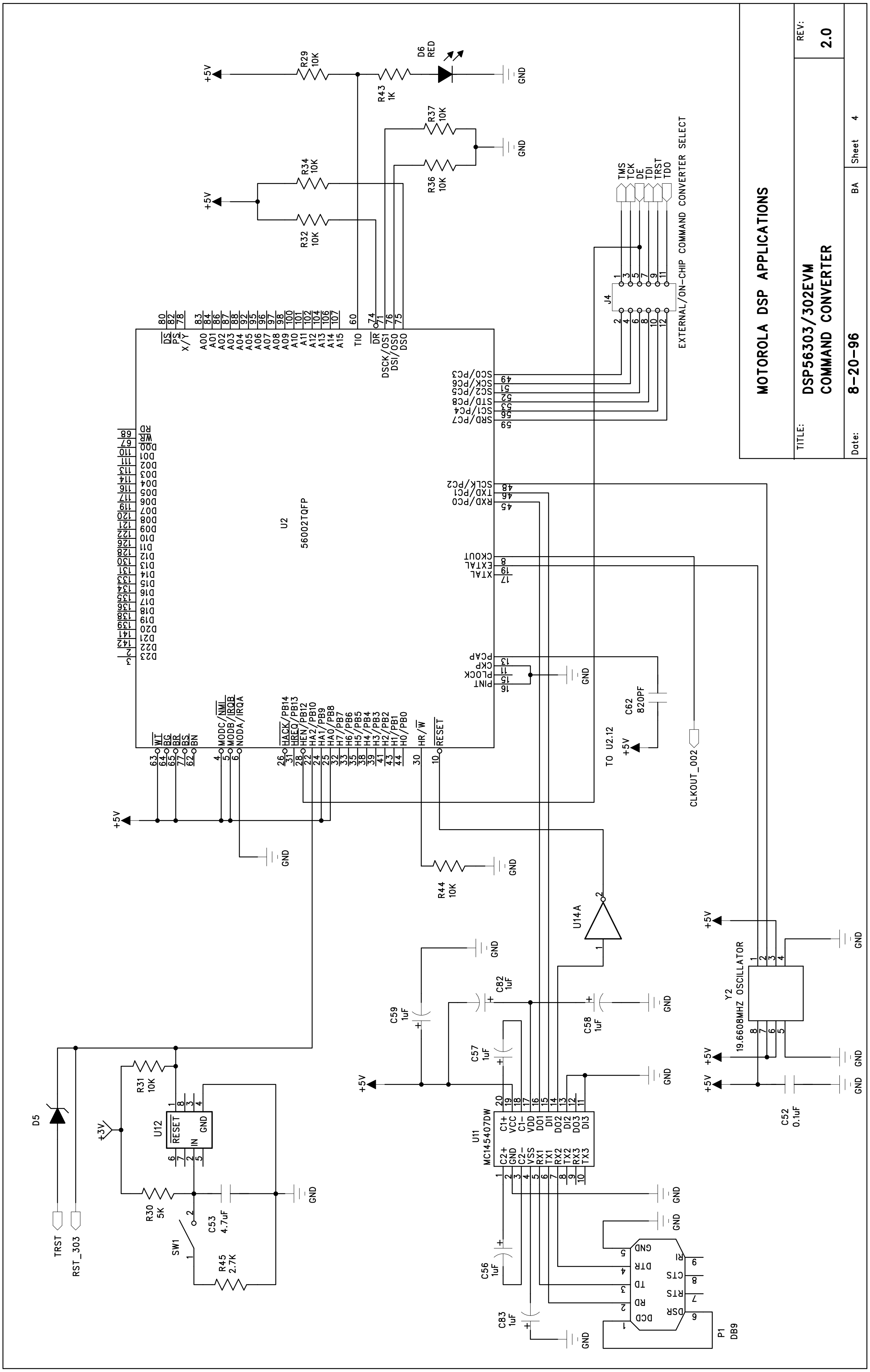
APPENDIX A

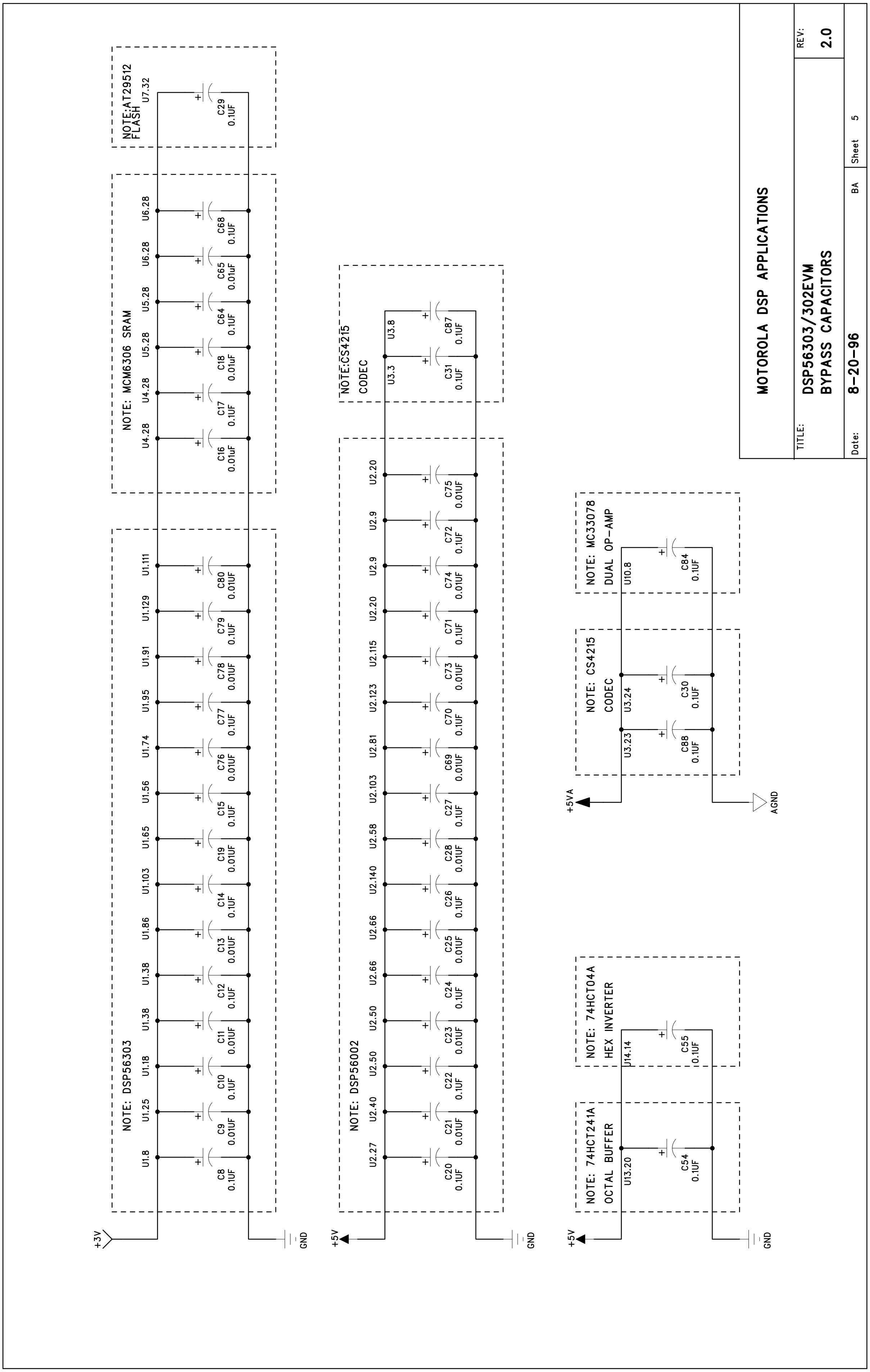
DSP56303EVM SCHEMATICS

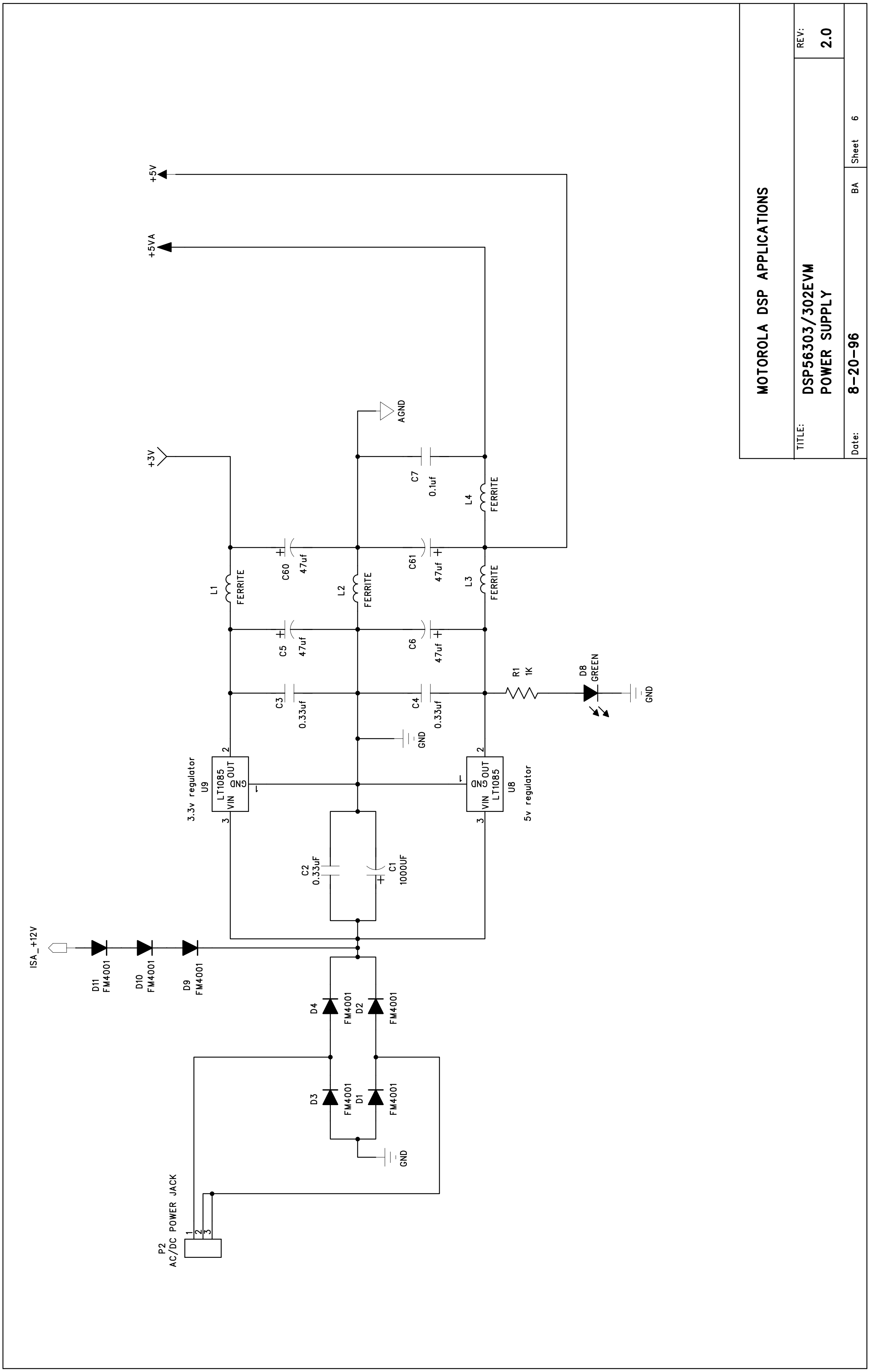

MOTOROLA DSP APPLICATIONS
TITLE: DSP56303/302EVM
CPU
Date: 8-20-96
Sheet 1
REV: 2.0
BA
U1
DSP56303/302
ISA CONNECTOR
JTAG CONNECTOR
ESSI0
ESSI1/SCI
U13A 74HCT241A
U13B 74HCT241A
16.9344MHZ


MOTOROLA DSP APPLICATIONS
TITLE: DSP56303/302EVM MEMORY
Date: 8-20-96
Sheet 2
REV: 2.0


MOTOROLA DSP APPLICATIONS
TITLE: DSP56303/302EVM
AUDIO CODEC
Date: 8-20-96
BA
Sheet 3
REV: 2.0
U3
CS4215
P4 HDPHNE
P5 OUT
P3 IN
Y3 24.576MHz
U10A MC33078
U10B MC33078
TO AGND1, PIN 22
AGND
CMOUT
+5V
ESSI0_RX
ESSI0_TX
ESSI0_FS
RST_CODEC
D/C
ESSI0_SCK


MOTOROLA DSP APPLICATIONS
TITLE:
DSP56303/302EVM
COMMAND CONVERTER
Date:
8-20-96
BA
Sheet 4
REV:
2.0


MOTOROLA DSP APPLICATIONS
TITLE:
DSP56303/302EVM
BYPASS CAPACITORS
REV:
2.0
Date:
8-20-96
BA
Sheet 5
NOTE:AT29512
FLASH
U7.32
C29 0.1UF
NOTE: MCM6306 SRAM
U4.28 U4.28 U5.28 U5.28 U6.28 U6.28
C16 0.01uF
C17 0.1UF
C18 0.01uF
C64 0.1UF
C65 0.01uF
C68 0.1UF
NOTE: DSP56303
U1.8 U1.25 U1.18 U1.38 U1.38 U1.86 U1.103 U1.65 U1.56 U1.74 U1.95 U1.91 U1.129 U1.111
C8 0.1UF
C9 0.01UF
C10 0.1UF
C11 0.01UF
C12 0.1UF
C13 0.01UF
C14 0.1UF
C19 0.01UF
C15 0.1UF
C76 0.01UF
C77 0.1UF
C78 0.01UF
C79 0.1UF
C80 0.01UF
+3V
GND
NOTE:CS4215
CODEC
U3.3 U3.8
C31 0.1UF
C87 0.1UF
NOTE: DSP56002
U2.27 U2.40 U2.50 U2.50 U2.66 U2.66 U2.140 U2.58 U2.103 U2.81 U2.123 U2.115 U2.20 U2.9 U2.9 U2.20
C20 0.1UF
C21 0.01UF
C22 0.1UF
C23 0.01UF
C24 0.1UF
C25 0.01UF
C26 0.1UF
C28 0.01UF
C27 0.1UF
C69 0.01UF
C70 0.1UF
C73 0.01UF
C71 0.1UF
C74 0.01UF
C72 0.1UF
C75 0.01UF
+5V
GND
NOTE: MC33078
DUAL OP-AMP
U10.8
C84 0.1UF
NOTE: CS4215
CODEC
U3.24 U3.23
C30 0.1UF
C88 0.1UF
+5VA
AGND
NOTE: 74HCT04A
HEX INVERTER
U14.14
C55 0.1UF
NOTE: 74HCT241A
OCTAL BUFFER
U13.20
C54 0.1UF
+5V
GND


MOTOROLA DSP APPLICATIONS
TITLE:
DSP56303/302EVM
POWER SUPPLY
REV:
2.0
Date:
8-20-96
BA
Sheet 6
+5V
+5VA
+3V
AGND
GND
C7
0.1uf
L4
FERRITE
L1
FERRITE
L2
FERRITE
L3
FERRITE
C60
47uf
C61
47uf
C5
47uf
C6
47uf
C3
0.33uf
C4
0.33uf
R1
1K
D8
GREEN
3.3v regulator
U9
LT1085
VIN GND OUT
5v regulator
U8
LT1085
VIN GND OUT
C2
0.33uF
C1
1000UF
ISA_+12V
D11
FM4001
D10
FM4001
D9
FM4001
D4
FM4001
D2
FM4001
D3
FM4001
D1
FM4001
P2
AC/DC POWER JACK

APPENDIX B

DSP56303EVM PARTS LIST

B.1 PARTS LISTING

The following table contains information on the parts and devices on the DSP56303EVM.

Table B-1 DSP56303EVM Parts List

Designator	Manufacturer	Part Number	Description
U1	Motorola	DSP56303	DSP
U2	Motorola	DSP56002	DSP (OnCE)
U3	Crystal Semiconductor	CS4215KL	Audio Codec
U4 U5 U6	Motorola	MCM6306DJ15	SRAM
U7	Atmel	AT29LV512-20JC	Flash PEROM
U8	Linear Technologies	LT1085CT-5	5V Regulator
U9	Linear Technologies	LT1085CT-3.3	3.3V Regulator
U10	Motorola	MC33078D	Dual Op-Amp
U11	Motorola	MC145407DW	RS-232 Transceiver
U12	Motorola	MC34164D-3	Undervolt Sensor
U13	Motorola	MC74HCT241ADW	Octal Noninverting Buffer
U14	Motorola	MC74HCT04AD	Hex Inverter
D1 D2 D3 D4 D9 D10 D11	Rectron	FM4001	Diode
D5	Rectron	FM5817	Schottky Diode
D6 D12	Quality Technologies	HLMP1700	Red LED
D8	Quality Technologies	HLMP1790	Green LED

Table B-1 DSP56303EVM Parts List (Continued)

Designator	Manufacturer	Part Number	Description
Y1	Ecliptek	EC2-169-16.9344MHZ	16.9344 MHz Crystal
Y2	ECS	OECS-196.6-3-C3X1A	19.6608 MHz Crystal
Y3	Ecliptek	EC2-246-24.576MHZ	24.576 MHz Crystal
SW1 SW2 SW3	Panasonic	EVQ-QS205K	6 mm Switch
P1	Mouser	152-3409	DB-9 Female Connector
P2	Switchcraft	RAPC-722	2.1 mm DC Power Jack
P3 P4 P5	Switchcraft	35RAPC4BHN2	3.5 mm Miniature Jack
J1	Robinson Nugent	NSH-8DB-S2-TG	Header 8 pin double row
J2 J3	Robinson Nugent	NSH-16DB-S2-TG	Header 16 pin double row
J4 J6 J7	Robinson Nugent	NSH-12DB-S2-TG	Header 12 pin double row
J5	Robinson Nugent	NSH-14DB-S2-TG	Header 14 pin double row
J8	Robinson Nugent	P2DN-40A-S1-TR	Header 40 pin double row
J9	Robinson Nugent	NSH-3SB-S2-TG	Header 3 pin single row
C51	Xicon	MR16V10	10 µF Capacitor
C33 C34 C47 C56 C57 C58 C59 C82 C83	Murata	GRM42-6Y5V105Z025BL	1.0 µF Capacitor

Table B-1 DSP56303EVM Parts List (Continued)

Designator	Manufacturer	Part Number	Description
C7 C8 C10 C12 C14 C15 C17 C20 C22 C24 C26 C27 C29 C30 C31 C50 C52 C54 C55 C64 C68 C70 C71 C72 C77 C79 C84 C87 C88	Murata	GRM40-X7R104K025BL	0.1 µF Capacitor
C9 C11 C13 C16 C18 C19 C21 C23 C25 C28 C37 C39 C40 C65 C69 C73 C74 C75 C76 C78 C80 C81	Murata	GRM40-X7R103K050BL	0.01 µF Capacitor
C53	Kemet	T491B475K016AS	4.7 µF Capacitor
C41 C42	Murata	GRM40-COG220J050BL	22 pF Capacitor
C48 C49 C66 C67	Murata	GRM40-COG270J050BL	27 pF Capacitor
C2 C3 C4 C44 C46 C85 C86	Murata	GRM42-6Y5V334Z025BL	0.33 µF Capacitor
C38	Murata	GRM42-6Y5V474Z025BL	0.47 µF Capacitor
C43 C45	Murata	GRM40-COG470J050BL	47 pF Capacitor
C35 C36	Murata	GRM40-COG222J050BL	2200 pF Capacitor
C62	Murata	GRM40-X7R821K050BL	820 pF Capacitor
C63	Murata	GRM40-X7R182K050BL	1800 pF Capacitor
C5 C6 C60 C61	Xicon	MLRL10V47	47 µF Capacitor
C1	Xicon	XAL16V1000	1000 µF Capacitor
L1 L2 L3 L4	Murata	BL01RN1-A62	Ferrite Bead
R1 R43	NIC	NRC12RF1001TR	1 K Resistor
R30	NIC	NRC12RF5001TR	5 K Resistor

Table B-1 DSP56303EVM Parts List (Continued)

Designator	Manufacturer	Part Number	Description
R4 R5 R6 R7 R20 R21 R22 R23 R24 R25 R29 R31 R32 R34 R35 R36 R37 R38 R39 R40 R41 R42 R44	NIC	NRC12RF1002TR	10 K Resistor
R8 R9	NIC	NRC25RF15R8TR	15.8 KΩ Resistor
R14 R15	NIC	NRC12RF1500TR	150 Ω Resistor
R16 R18	NIC	NRC12RF2212TR	22.1 KΩ Resistor
R12 R13	NIC	NRC12RF3922TR	39.2 KΩ Resistor
R17 R19	NIC	NRC12RF4752TR	47.5 KΩ Resistor
R45	NIC	NRC12RF5600TR	2.7 KΩ Resistor
R10 R11 R46	NIC	NRC12RF6040TR	604 Ω Resistor
R28	NIC	NRC12RF6803TR	680 KΩ Resistor

APPENDIX C

MOTOROLA ASSEMBLER NOTES

C.1 INTRODUCTION

This appendix supplements information in *Section 3* of this document and provides a detailed description of the following components used with the Motorola Assembler:

- Special characters significant to the assembler
- Assembler directives
- Structure control statements

C.2 ASSEMBLER SIGNIFICANT CHARACTERS

There are several one and two character sequences that are significant to the assembler. The following subsections define these characters and their use.

C.2.1 ; Comment Delimiter Character

Any number of characters preceded by a semicolon (;), but not part of a literal string, is considered a comment. Comments are not significant to the assembler, but they can be used to document the source program. Comments will be reproduced in the assembler output listing. Comments are normally preserved in macro definitions, but this option can be turned off (see the **OPT** directive).

Comments can occupy an entire line, or can be placed after the last assembler-significant field in a source statement. A comment starting in the first column of the source file will be aligned with the label field in the listing file. Otherwise, the comment will be shifted right and aligned with the comment field in the listing file.

Example B-1 Example of Comment Delimiter

```
; THIS COMMENT BEGINS IN COLUMN 1 OF THE SOURCE FILE
LOOP    JSR     COMPUTE ; THIS IS A TRAILING COMMENT
                        ; THESE TWO COMMENTS ARE PRECEDED
                        ; BY A TAB IN THE SOURCE FILE
```

C.2.2 ;; Unreported Comment Delimiter Characters

Unreported comments are any number of characters preceded by two consecutive semicolons (;;) that are not part of a literal string. Unreported comments are not considered significant by the assembler, and can be included in the source statement, following the same rules as normal comments. However, unreported comments are never reproduced on the assembler output listing, and are never saved as part of macro definitions.

Example B-2 Example of Unreported Comment Delimiter

```
;; THESE LINES WILL NOT BE REPRODUCED
;; IN THE SOURCE LISTING
```

C.2.3 \ Line Continuation Character or Macro Argument Concatenation Character Line Continuation

C.2.3.1 Line Continuation

The backslash character (\), if used as the last character on a line, indicates to the assembler that the source statement is continued on the following line. The continuation line will be concatenated to the previous line of the source statement, and the result will be processed by the assembler as if it were a single line source statement. The maximum source statement length (the first line and any continuation lines) is 512 characters.

Example B-3 Example of Line Continuation Character

```
; THIS COMMENT \
EXTENDS OVER \
THREE LINES
```

C.2.3.2 Macro Argument Concatenation

The backslash (\) is also used to cause the concatenation of a macro dummy argument with other adjacent alphanumeric characters. For the macro processor to recognize dummy arguments, they must normally be separated from other alphanumeric characters by a non-symbol character. However, sometimes it is desirable to concatenate the argument characters with other characters. If an argument is to be concatenated in front of or behind some other symbol characters, then it must be followed by or preceded by the backslash, respectively.

Example B-4 Example of Macro Concatenation

Suppose the source input file contained the following macro definition:

```
SWAP_REG    MACRO       REG1,REG2  ;swap REG1,REG2 using D4.L as temp
            MOVE        R\REG1,D4.L
            MOVE        R\REG2,R\REG1
            MOVE        D4.L,R\REG2
            ENDM
```

The concatenation operator (\) indicates to the macro processor that the substitution characters for the dummy arguments are to be concatenated in both cases with the character R. If this macro were called with the following statement,

```
            SWAP_REG             0,1
```

the resulting expansion would be:

```
            MOVE        R0,D4.L
            MOVE        R1,R0
            MOVE        D4.L,R1
```

C.2.4 ? Return Value of Symbol Character

The **?**<symbol> sequence, when used in macro definitions, will be replaced by an ASCII string representing the value of <symbol>. This operator may be used in association with the backslash (\) operator. The value of <symbol> must be an integer (not floating point).

Example B-5 Example of Use of Return Value Character

Consider the following macro definition:

```
SWAP_SYMMACRO  REG1,REG2;swap REG1,REG2 using D4.L as temp
        MOVE   R\?REG1,D4.L
        MOVE   R\?REG2,R\?REG1
        MOVE   D4.L,R\?REG2
        ENDM
```

If the source file contained the following SET statements and macro call,

```
AREG        SET         0
BREG        SET         1
            SWAP_SYM    AREG,BREG
```

the resulting expansion as it would appear on the source listing would be:

Example B-5 Example of Use of Return Value Character

```
        MOVE    R0,D4.L
        MOVE    R1,R0
        MOVE    D4.L,R1
```

C.2.5 % Return Hex Value of Symbol Character

The **%**<symbol> sequence, when used in macro definitions, will be replaced by an ASCII string representing the hexadecimal value of <symbol>. This operator may be used in association with the backslash (\) operator. The value of <symbol> must be an integer (not floating point).

Example B-6 Example of Return Hex Value Symbol Character

Consider the following macro definition:

```
GEN_LAB    MACRO      LAB,VAL,STMT
LAB\%VAL   STMT
           ENDM
```

If this macro were called as follows,

```
NUM        SET        10
           GEN_LAB    HEX,NUM,'NOP'
```

The resulting expansion as it would appear in the listing file would be:

```
HEXA       NOP
```

C.2.6 ^ Macro Local Label Override

The circumflex (^), when used as a unary expression operator in a macro expansion, will cause any local labels in its associated term to be evaluated at normal scope rather than macro scope. This means that any underscore labels in the expression term following the circumflex will not be searched for in the macro local label list. The operator has no effect on normal labels or outside of a macro expansion. The circumflex operator is useful for passing local labels as macro arguments to be used as referents in the macro.

Note: The circumflex is also used as the binary exclusive OR operator.

Example B-7 Example of Local Label Override Character

Consider the following macro definition:

```
LOAD        MACRO       ADDR
            MOVE        P:^ADDR,R0
            ENDM
```

If this macro were called as follows,

```
_LOCAL
            LOAD        _LOCAL
```

the assembler would ordinarily issue an error since _LOCAL is not defined within the body of the macro. With the override operator the assembler recognizes the _LOCAL symbol outside the macro expansion and uses that value in the MOVE instruction.

C.2.7 " Macro String Delimiter or Quoted String DEFINE Expansion Character

C.2.7.1 Macro String

The double quote ("), when used in macro definitions, is transformed by the macro processor into the string delimiter, the single quote ('). The macro processor examines the characters between the double quotes for any macro arguments. This mechanism allows the use of macro arguments as literal strings.

Example B-8 Example of a Macro String Delimiter Character

Using the following macro definition,

```
CSTR        MACRO       STRING
            DC          "STRING"
            ENDM
```

and a macro call,

```
CSTR    ABCD
```

the resulting macro expansion would be:

```
            DC          'ABCD'
```

C.2.7.2 Quoted String DEFINE Expansion

A sequence of characters which matches a symbol created with a **DEFINE** directive will not be expanded if the character sequence is contained within a quoted string. Assembler strings generally are enclosed in single quotes ('). If the string is enclosed in double quotes (") then **DEFINE** symbols will be expanded within the string. In all other respects usage of double quotes is equivalent to that of single quotes.

Example B-9 Example of a Quoted String DEFINE Expression

Consider the source fragment below:

```
           DEFINE     LONG       'short'
STR_MAC    MACRO      STRING
           MSG        'This is a LONG STRING'
           MSG        "This is a LONG STRING"
           ENDM
```

If this macro were invoked as follows,

```
           STR_MAC    sentence
```

then the resulting expansion would be:

```
           MSG        'This is a LONG STRING'
           MSG        'This is a short sentence'
```

C.2.8 @ Function Delimiter

All assembler built-in functions start with the **@** symbol.

Example B-10 Example of a Function Delimiter Character

```
SVAL       EQ         @SQT(FVAL) ; OBTAIN SQUARE ROOT
```

C.2.9 * Location Counter Substitution

When used as an operand in an expression, the asterisk represents the current integer value of the runtime location counter.

Example B-11 Example of a Location Counter Substitution

```
           ORG        X:$100
XBASE      EQU        *+$20      ; XBASE = $120
```

C.2.10 ++ String Concatenation Operator

Any two strings can be concatenated with the string concatenation operator (**++**). The two strings must each be enclosed by single or double quotes, and there must be no intervening blanks between the string concatenation operator and the two strings.

Example B-12 Example of a String Concatenation Operator

```
'ABC'++'DEF' = 'ABCDEF'
```

C.2.11 [ ] Substring Delimiter [<string>,<offset><length>]

Square brackets delimit a substring operation. The <string> argument is the source string. <offset> is the substring starting position within <string>. <length> is the length of the desired substring. <string> may be any legal string combination, including another substring. An error is issued if either <offset> or <length> exceed the length of <string>.

Example B-13 Example of a Substring Delimiter

```
        DEFINE    ID        ['DSP56000',3,5]      ; ID = '56000'
```

C.2.12 << I/O Short Addressing Mode Force Operator

Many DSP instructions allow an I/O short form of addressing. If the value of an absolute address is known to the assembler on pass one, then the assembler will always pick the shortest form of addressing consistent with the instruction format. If the absolute address is not known to the assembler on pass one (that is, the address is a forward or external reference), then the assembler will pick the long form of addressing by default. If this is not desired, then the I/O short form of addressing can be forced by preceding the absolute address by the I/O short addressing mode force operator (**<<**).

Example B-14 Example of an I/O Short Addressing Mode Force Operator

Since the symbol IOPORT is a forward reference in the following sequence of source lines, the assembler would pick the long absolute form of addressing by default:

```
          BTST      #4,Y:IOPORT
IOPORT    EQU       Y:$FFF3
```

Example B-14 Example of an I/O Short Addressing Mode Force Operator

Because the long absolute addressing mode would cause the instruction to be two words long instead of one word for the I/O short absolute addressing mode, it would be desirable to force the I/O short absolute addressing mode as shown below:

```
        BTST    #4,Y:<<IOPORT
IOPORT  EQU     Y:$FFF3
```

C.2.13 < Short Addressing Mode Force Operator

Many DSP instructions allow a short form of addressing. If the value of an absolute address is known to the assembler on pass one, or the **FORCE** SHORT directive is active, then the assembler will always pick the shortest form of addressing consistent with the instruction format. If the absolute address is not known to the assembler on pass one (that is, the address is a forward or external reference), then the assembler will pick the long form of addressing by default. If this is not desired, then the short absolute form of addressing can be forced by preceding the absolute address by the short addressing mode force operator (<).

Example B-15 Example of a Short Addressing Mode Force Operator

Since the symbol DATAST is a forward reference in the following sequence of source lines, the assembler would pick the long absolute form of addressing by default:

```
        MOVE    D0.L,Y:DATAST
DATAST  EQU     Y:$23
```

Because the long absolute addressing mode would cause the instruction to be two words long instead of one word for the short absolute addressing mode, it would be desirable to force the short absolute addressing mode as shown below:

```
        MOVE    D0.L,Y:<DATAST
DATAST  EQU     Y:$23
```

C.2.14 > Long Addressing Mode Force Operator

Many DSP instructions allow a long form of addressing. If the value of an absolute address is known to the assembler on pass one, then the assembler will always pick the shortest form of addressing consistent with the instruction format, unless the

FORCE LONG directive is active. If this is not desired, then the long absolute form of addressing can be forced by preceding the absolute address by the long addressing mode force operator (>).

Example B-16 Example of a Long Addressing Mode Force Operator

Since the symbol DATAST is a not a forward reference in the following sequence of source lines, the assembler would pick the short absolute form of addressing:

```
DATAST    EQU     Y:$23
          MOVE    D0.L,Y:DATAST
```

If this is not desirable, then the long absolute addressing mode can be forced as shown below:

```
DATAST    EQU     Y:$23
          MOVE    D0.L,Y:>DATAST
```

C.2.15 # Immediate Addressing Mode

The pound sign (#) is used to indicate to the assembler to use the immediate addressing mode.

Example B-17 Example of Immediate Addressing Mode

```
CNST      EQU     $5
          MOVE    #CNST,D0.L
```

C.2.16 #< Immediate Short Addressing Mode Force Operator

Many DSP instructions allow a short immediate form of addressing. If the immediate data is known to the assembler on pass one (not a forward or external reference), or the **FORCE** SHORT directive is active, then the assembler will always pick the shortest form of immediate addressing consistent with the instruction. If the immediate data is a forward or external reference, then the assembler will pick the long form of immediate addressing by default. If this is not desired, then the short form of addressing can be forced using the immediate short addressing mode force operator (**#<**).

Example B-18 Example of Immediate Short Addressing Mode Force Operator

In the following sequence of source lines, the symbol CNST is not known to the assembler on pass one, and therefore, the assembler would use the long immediate addressing form for the **MOVE** instruction.

```
        MOVE    #CNST,D0.L
CNST    EQU     $5
```

Because the long immediate addressing mode makes the instruction two words long instead of one word for the immediate short addressing mode, it may be desirable to force the immediate short addressing mode as shown below:

```
        MOVE    #<CNST,D0.L
CNST    EQU     $5
```

C.2.17 #> Immediate Long Addressing Mode Force Operator

Many DSP instructions allow a long immediate form of addressing. If the immediate data is known to the assembler on pass one (not a forward or external reference), then the assembler will always pick the shortest form of immediate addressing consistent with the instruction, unless the **FORCE** LONG directive is active. If this is not desired, then the long form of addressing can be forced using the immediate long addressing mode force operator (**#>**).

Example B-19 Example of an Immediate Long Addressing Mode Operator

In the following sequence of source lines, the symbol CNST is known to the assembler on pass one, and therefore, the assembler would use the short immediate addressing form for the **MOVE** instruction.

```
CNST    EQU     $5
        MOVE    #CNST,D0.L
```

If this is not desirable, then the long immediate form of addressing can be forced as shown below:

```
CNST    EQU     $5
        MOVE    #>CNST,D0.L
```

C.3 ASSEMBLER DIRECTIVES

C.3.1 BADDR Set Buffer Address

```
BADDR      <M | R>,<expression>
```

The **BADDR** directive sets the runtime location counter to the address of a buffer of the given type, the length of which in words is equal to the value of <expression>. The buffer type may be either **M**odulo or **R**everse-carry. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k >= <expression>. An error will be issued if there is insufficient memory remaining to establish a valid base address. Unlike other buffer allocation directives, the runtime location counter is **not** advanced by the value of the integer expression in the operand field; the location counter remains at the buffer base address. The block of memory intended for the buffer is not initialized to any value.

The result of <expression> may have any memory space attribute but must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). If a **M**odulo buffer is specified, the expression must fall within the range 2 ≤ <expression> ≤ *m*, where *m* is the maximum address of the target DSP. If a **R**everse-carry buffer is designated and <expression> is not a power of two a warning will be issued. A label is not allowed with this directive.

Note: See also **BSM**, **BSB**, **BUFFER**, **DSM**, **DSR**

Example B-20 Example BADDR Directive

```
          ORG       X:$100
M_BUF     BADDR     M,24        ; CIRCULAR BUFFER MOD 24
```

C.3.2 BSB Block Storage Bit-Reverse

```
[<label>]  BSB      <expression>[,<expression>]
```

The **BSB** directive causes the assembler to allocate and initialize a block of words for a reverse-carry buffer. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k is greater than or equal to the value of the first expression. An error will occur if the first expression contains symbols that are not yet defined (forward references) or if the expression has a value of less than or equal to zero. Also, if the first expression is not a power of two a warning will be generated. Both expressions can have any memory space attribute.

<label>, if present, will be assigned the value of the runtime location counter after a valid base address has been established.

Only one word of object code will be shown on the listing, regardless of how large the first expression is. However, the runtime location counter will be advanced by the number of words generated.

Note: See also **BSC**, **BSM**, **DC.**

Example B-21 Buffer Directive

```
BUFFER      BSB       BUFSIZ    ; INITIALIZE BUFFER TO ZEROS
```

C.3.3 BSC Block Storage of Constant

```
[<label>]   BSC       <expression>[,<expression>]
```

The **BSC** directive causes the assembler to allocate and initialize a block of words. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the first expression contains symbols that are not yet defined (forward references) or if the expression has a value of less than or equal to zero, an error will be generated. Both expressions can have any memory space attribute.

<label>, if present, will be assigned the value of the runtime location counter at the start of the directive processing.

Only one word of object code will be shown on the listing, regardless of how large the first expression is. However, the runtime location counter will be advanced by the number of words generated.

Note: See also **BSM**, **BSB**, **DC.**

Example B-22 Block Storage of Constant Directive

```
UNUSED     BSC       $2FFF-@LCV(R),$FFFFFFFF; FILL UNUSED EPROM
```

C.3.4 BSM Block Storage Modulo

```
[<label>]   BSM      <expression>[,<expression>]
```

The **BSM** directive causes the assembler to allocate and initialize a block of words for a modulo buffer. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k is greater than or equal to the value of the first expression. An error will occur if the first expression contains symbols that are not yet defined (forward references), has a value of less than or equal to zero, or falls outside the range 2 ≤ <expression> ≤ *m*, where *m* is the maximum address of the target DSP. Both expressions can have any memory space attribute.

<label>, if present, will be assigned the value of the runtime location counter after a valid base address has been established.

Only one word of object code will be shown on the listing, regardless of how large the first expression is. However, the runtime location counter will be advanced by the number of words generated.

Note: See also **BSC**, **BSB**, **DC.**

Example B-23 Block Storage Modulo Directive

```
BUFFER     BSM       BUFSIZ,$FFFFFFFF; INITIALIZE BUFFER TO ALL ONES
```

C.3.5 BUFFER Start Buffer

```
BUFFER    <M | R>,<expression>
```

The **BUFFER** directive indicates the start of a buffer of the given type. Data is allocated for the buffer until an **ENDBUF** directive is encountered. Instructions and most data definition directives may appear between the **BUFFER** and **ENDBUF** pair, although **BUFFER** directives may not be nested and certain types of directives such as **MODE**, **ORG**, **SECTION**, and other buffer allocation directives may not be used. The <expression> represents the buffer size. If less data is allocated than the size of the buffer, the remaining buffer locations will be uninitialized. If more data is allocated than the specified size of the buffer, an error is issued.

The **BUFFER** directive sets the runtime location counter to the address of a buffer of the given type, the length of which in words is equal to the value of <expression>. The buffer type may be either **M**odulo or **R**everse-carry. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k >= <expression>. An error will be issued if there is insufficient memory remaining to establish a valid base address. Unlike other buffer allocation directives, the runtime location counter is **not** advanced by the value of the integer expression in the operand field; the location counter remains at the buffer base address.

The result of <expression> may have any memory space attribute but must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). If a **M**odulo buffer is specified, the expression must fall within the range $2 \leq$ <expression> $\leq m$, where m is the maximum address of the target DSP. If a **R**everse-carry buffer is designated and <expression> is not a power of two a warning will be issued.

Note: A label is not allowed with this directive.

Note: See also **BADDR**, **BSM**, **BSB**, **DSM**, **DSR**, **ENDBUF.**

Example B-24 Buffer Directive

```
        ORG       X:$100
        BUFFER    M,24        ; CIRCULAR BUFFER MOD 24
M_BUF   DC        0.5,0.5,0.5,0.5
        DS        20          ; REMAINDER UNINITIALIZED
        ENDBUF
```

C.3.6 COBJ Comment Object File

```
COBJ     <string>
```

The **COBJ** directive is used to place a comment in the object code file. The <string> will be put in the object file as a comment.

Note: A label is not allowed with this directive.

Note: See also **IDENT.**

Example B-25 COBM Directive

```
COBJ     'Start of filter coefficients'
```

C.3.7 COMMENT Start Comment Lines

```
COMMENT   <delimiter>
.
.
<delimiter>
```

The **COMMENT** directive is used to define one or more lines as comments. The first non-blank character after the **COMMENT** directive is the comment delimiter. The two delimiters are used to define the comment text. The line containing the second comment delimiter will be considered the last line of the comment. The comment text can include any printable characters and the comment text will be reproduced in the source listing as it appears in the source file.

Note: A label is not allowed with this directive.

Example B-26 COMMENT Directive

```
COMMENT   + This is a one line comment +
COMMENT   *         This is a multiple line
                    comment. Any number of lines
                    can be placed between the two delimiters.
          *
```

C.3.8 DC Define Constant

```
<label>]   DC        <arg>[,<arg>,...,<arg>]
```

The **DC** directive allocates and initializes a word of memory for each <arg> argument. <arg> may be a numeric constant, a single or multiple character string constant, a symbol, or an expression. The **DC** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. If the **DC** directive is used in L memory, the arguments will be evaluated and stored as long word quantities. Otherwise, an error will occur if the evaluated argument value is too large to represent in a single DSP word.

<label>, if present, will be assigned the value of the runtime location counter at the start of the directive processing.

Integer arguments are stored as is; floating point numbers are converted to binary values. Single and multiple character strings are handled in the following manner:

- Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character.

Example B-27 Single Character String Definition

```
         'R'        = $000052
```

- Multiple character strings represent words whose bytes are composed of concatenated sequences of the ASCII representation of the characters in the string (unless the **NOPS** option is specified; see the **OPT** directive). If the number of characters is not an even multiple of the number of bytes per DSP word, then the last word will have the remaining characters left aligned and the rest of the word will be zero-filled. If the **NOPS** option is given, each character in the string is stored in a word whose lower seven bits represent the ASCII value of the character.

Example B-28 Multiple Character String Definition

```
'ABCD'    =         $414243
                    $440000
```

Note: See also **BSC**, **DCB.**

Example B-29 DC Directive

```
TABLE     DC        1426,253,$2662,'ABCD'
CHARS     DC        'A','B','C','D'
```

C.3.9 DCB Define Constant Byte

```
[<label>]    DCB       <arg>[,<arg>,...,<arg>]
```

The **DCB** directive allocates and initializes a byte of memory for each <arg> argument. <arg> may be a byte integer constant, a single or multiple character string constant, a symbol, or a byte expression. The **DCB** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive byte locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding byte location will be filled with zeros.

<label>, if present, will be assigned the value of the runtime location counter at the start of the directive processing.

Integer arguments are stored as is, but must be byte values (e.g. within the range 0-255); floating point numbers are not allowed. Single and multiple character strings are handled in the following manner:

- Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character. (See **Example B-27**.)
- Multiple character strings represent words whose bytes are composed of concatenated sequences of the ASCII representation of the characters in the string (unless the **NOPS** option is specified; see the **OPT** directive). If the number of characters is not an even multiple of the number of bytes per DSP word, then the last word will have the remaining characters left aligned and the rest of the word will be zero-filled. If the **NOPS** option is given, each character in the string is stored in a word whose lower seven bits represent the ASCII value of the character. (See **Example B-28**.)

Note: See also **BSC**, **DC.**

Example B-30 DCB Directive

```
TABLE     DCB      'two',0,'strings',0
CHARS     DCB      'A','B','C','D'
```

C.3.10 DEFINE Define Substitution String

```
DEFINE     <symbol>   <string>
```

The **DEFINE** directive is used to define substitution strings that will be used on all following source lines. All succeeding lines will be searched for an occurrence of <symbol>, which will be replaced by <string>. This directive is useful for providing better documentation in the source program. <symbol> must adhere to the restrictions for non-local labels. That is, it cannot exceed 512 characters, the first of which must be alphabetic, and the remainder of which must be either alphanumeric or the underscore(_). A warning will result if a new definition of a previously defined symbol is attempted. The assembler output listing will show lines after the **DEFINE** directive has been applied and therefore redefined symbols will be replaced by their substitution strings (unless the **NODXL** option in effect; see the **OPT** directive).

Macros represent a special case. **DEFINE** directive translations will be applied to the macro definition as it is encountered. When the macro is expanded any active **DEFINE** directive translations will again be applied.

DEFINE directive symbols that are defined within a section will only apply to that section. See the **SECTION** directive.

Note: A label is not allowed with this directive.

Note: See also **UNDEF.**

Example B-31 DEFINE Directive

If the following **DEFINE** directive occurred in the first part of the source program:

```
DEFINE    ARRAYSIZ   '10 * SAMPLSIZ'
```

then the source line below:

```
DS        ARRAYSIZ
```

would be transformed by the assembler to the following:

```
DS        10 * SAMPLSIZ
```

C.3.11 DS Define Storage

```
[<label>]  DS        <expression>
```

The **DS** directive reserves a block of memory the length of which in words is equal to the value of <expression>. This directive causes the runtime location counter to be advanced by the value of the absolute integer expression in the operand field. <expression> can have any memory space attribute. The block of memory reserved is not initialized to any value. The expression must be an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined).

<label>, if present, will be assigned the value of the runtime location counter at the start of the directive processing.

Note: See also **DSM**, **DSR.**

Example B-32 DS Directive

```
S_BUF     DS      12      ; SAMPLE BUFFER
```

C.3.12 DSM Define Modulo Storage

```
[<label>]  DSM     <expression>
```

The **DSM** directive reserves a block of memory the length of which in words is equal to the value of <expression>. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k >= <expression>. An error will be issued if there is insufficient memory remaining to establish a valid base address. Next the runtime location counter is advanced by the value of the integer expression in the operand field. <expression> can have any memory space attribute. The block of memory reserved is not initialized to any given value. The result of <expression> must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). The expression also must fall within the range:

$2 \leq$ <expression> $\leq m$,
where m is the maximum address of the target DSP.

<label>, if present, will be assigned the value of the runtime location counter after a valid base address has been established.

Note: See also **DS**, **DSR.**

Example B-33 DSM Directive

```
          ORG     X:$100
M_BUF     DSM     24      ; CIRCULAR BUFFER MOD 24
```

C.3.13 DSR Define Reverse Carry Storage

```
[<label>]   DSR       <expression>
```

The **DSR** directive reserves a block of memory the length of which in words is equal to the value of <expression>. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where $2^k \geq$ <expression>. An error will be issued if there is insufficient memory remaining to establish a valid base address. Next the runtime location counter is advanced by the value of the integer expression in the operand field. <expression> can have any memory space attribute. The block of memory reserved is not initialized to any given value. The result of <expression> must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). Since the **DSR** directive is useful mainly for generating FFT buffers, if <expression> is not a power of two a warning will be generated.

<label>, if present, will be assigned the value of the runtime location counter after a valid base address has been established.

Note: See also **DS**, **DSM.**

Example B-34 DSR Directive

```
         ORG        X:$100
R_BUF    DSR        8          ; REVERSE CARRY BUFFER FOR 16 POINT FFT
```

C.3.14 DUP Duplicate Sequence of Source Lines

```
[<label>]   DUP       <expression>
            .
            .
            ENDM
```

The sequence of source lines between the **DUP** and **ENDM** directives will be duplicated by the number specified by the integer <expression>. <expression> can have any memory space attribute. If the expression evaluates to a number less than or equal to 0, the sequence of lines will not be included in the assembler output. The expression result must be an absolute integer and cannot contain any forward references (symbols that have not already been defined). The **DUP** directive may be nested to any level.

<label>, if present, will be assigned the value of the runtime location counter at the start of the **DUP** directive processing.

Note: See also **DUPA**, **DUPC**, **DUPF**, **ENDM**, **MACRO**

Example B-35 DUP Directive

The sequence of source input statements,

```
COUNT     SET       3
          DUP       COUNT     ; ASR BY COUNT
          ASR       D0
          ENDM
```

would generate the following in the source listing:

```
COUNT     SET       3
          DUP       COUNT     ; ASR BY COUNT
          ASR       D0
          ASR       D0
          ASR       D0
          ENDM
```

Note that the lines

```
          DUP       COUNT     ;ASR BY COUNT
          ENDM
```

will only be shown on the source listing if the **MD** option is enabled. The lines

```
          ASR       D0
          ASR       D0
          ASR       D0
```

will only be shown on the source listing if the **MEX** option is enabled.

Note: See the **OPT** directive in this appendix for more information on the **MD** and **MEX** options.

C.3.15 DUPA Duplicate Sequence With Arguments

```
[<label>] DUPA      <dummy>,<arg>[<,<arg>,...,<arg>]
          .
          .
          ENDM
```

The block of source statements defined by the **DUPA** and **ENDM** directives will be repeated for each argument. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding argument string. If the argument string is a null, then the block is repeated with each occurrence of the dummy parameter removed. If an argument includes an embedded blank or other assembler-significant character, it must be enclosed with single quotes.

<label>, if present, will be assigned the value of the runtime location counter at the start of the **DUPA** directive processing.

Note: See also **DUP**, **DUPC**, **DUPF**, **ENDM**, **MACRO.**

Example B-36 DUPA Directive

If the input source file contained the following statements,

```
DUPA      VALUE,12,32,34
DC        VALUE
ENDM
```

then the assembled source listing would show

```
DUPA      VALUE,12,32,34
DC        12
DC        32
DC        34
ENDM
```

Note that the lines

```
DUPA      VALUE,12,32,34
ENDM
```

will only be shown on the source listing if the **MD** option is enabled. The lines

```
DC        12
DC        32
DC        34
```

will only be shown on the source listing if the **MEX** option is enabled.

Note: See the **OPT** directive in this appendix for more information on the **MD** and **MEX** options.

C.3.16 DUPC Duplicate Sequence With Characters

```
[<label>]   DUPC     <dummy>,<string>
            .
            .
            ENDM
```

The block of source statements defined by the **DUPC** and **ENDM** directives will be repeated for each character of <string>. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding character in the string. If the string is null, then the block is skipped.

<label>, if present, will be assigned the value of the runtime location counter at the start of the **DUPC** directive processing.

Note: See also **DUP**, **DUPA**, **DUPF**, **ENDM**, **MACRO.**

Example B-37 DUPC Directive

If input source file contained the following statements,

```
DUPC    VALUE,'123'
DC      VALUE
ENDM
```

then the assembled source listing would show:

```
DUPC    VALUE,'123'
DC      1
DC      2
DC      3
ENDM
```

Note that the lines

```
DUPC    VALUE,'123'
ENDM
```

will only be shown on the source listing if the **MD** option is enabled. The lines

```
DC      1
DC      2
DC      3
```

will only be shown on the source listing if the **MEX** option is enabled.

Note: See the **OPT** directive in this appendix for more information on the **MD** and **MEX** options.

C.3.17 DUPF Duplicate Sequence In Loop

```
[<label>]   DUPF     <dummy>,[<start>],<end>[,<increment>]
            .
            .
            ENDM
```

The block of source statements defined by the **DUPF** and **ENDM** directives will be repeated in general (<end> - <start>) + 1 times when <increment> is 1. <start> is the starting value for the loop index; <end> represents the final value. <increment> is the increment for the loop index; it defaults to 1 if omitted (as does the <start> value). The <dummy> parameter holds the loop index value and may be used within the body of instructions.

<label>, if present, will be assigned the value of the runtime location counter at the start of the **DUPF** directive processing.

Note: See also **DUP**, **DUPA**, **DUPC**, **ENDM**, **MACRO.**

Example B-38 DUPF Directive

If input source file contained the following statements,

```
DUPF    NUM,0,7
MOVE    #0,R\NUM
ENDM
```

then the assembled source listing would show:

```
DUPF    NUM,0,7
MOVE    #0,R0
MOVE    #0,R1
MOVE    #0,R2
MOVE    #0,R3
MOVE    #0,R4
MOVE    #0,R5
MOVE    #0,R6
MOVE    #0,R7
ENDM
```

Note that the lines

```
DUPF    NUM,0,7
ENDM
```

will only be shown on the source listing if the **MD** option is enabled. The lines

Example B-38 DUPF Directive (Continued)

```
        MOVE    #0,R0
        MOVE    #0,R1
        MOVE    #0,R2
        MOVE    #0,R3
        MOVE    #0,R4
        MOVE    #0,R5
        MOVE    #0,R6
        MOVE    #0,R7
```

will only be shown on the source listing if the **MEX** option is enabled.

Note: See the **OPT** directive in this appendix for more information on the **MD** and **MEX** options.

C.3.18 END End of Source Program

END [<expression>]

The optional **END** directive indicates that the logical end of the source program has been encountered. Any statements following the **END** directive are ignored. The optional expression in the operand field can be used to specify the starting execution address of the program. <expression> may be absolute or relocatable, but must have a memory space attribute of **P**rogram or **N**one. The **END** directive cannot be used in a macro expansion.

Note: A label is not allowed with this directive.

Example B-39 END Directive

```
        END     BEGIN     ; BEGIN is the starting execution address
```

C.3.19 ENDBUF End Buffer

ENDBUF

The **ENDBUF** directive is used to signify the end of a buffer block. The runtime location counter will remain just beyond the end of the buffer when the **ENDBUF** directive is encountered.

Note: A label is not allowed with this directive.

Note: See also **BUFFER.**

Example B-40 ENDBUF Directive

```
            ORG         X:$100
BUF         BUFFER      R,64        ; uninitialized reverse-carry buffer
            ENDBUF
```

C.3.20 ENDIF End of Conditional Assembly

```
ENDIF
```

The **ENDIF** directive is used to signify the end of the current level of conditional assembly. Conditional assembly directives can be nested to any level, but the **ENDIF** directive always refers to the most previous **IF** directive.

Note: A label is not allowed with this directive.

Note: See also **IF.**

Example B-41 ENDIF Directive

```
            IF          @REL()
SAVEPC      SET         *           ; Save current program counter
            ENDIF
```

C.3.21 ENDM End of Macro Definition

```
ENDM
```

Every **MACRO**, **DUP**, **DUPA**, and **DUPC** directive must be terminated by an **ENDM** directive.

Note: A label is not allowed with this directive.

Note: See also **DUP**, **DUPA**, **DUPC**, **MACRO.**

Example B-42 ENDM Directive

```
SWAP_SYM  MACRO    REG1,REG   ;swap REG1,REG2 using D4.L as temp
          MOVE     R\?REG1,D4.L
          MOVE     R\?REG2,R\?REG1
          MOVE     D4.L,R\?REG2
          ENDM
```

C.3.22 ENDSEC End Section

```
ENDSEC
```

Every **SECTION** directive must be terminated by an **ENDSEC** directive.

Note: A label is not allowed with this directive.

Note: See also **SECTION.**

Example B-43 ENDSEC Directive

```
          SECTION  COEFF
          ORG      Y:
VALUES    BSC      $100       ; Initialize to zero
          ENDSEC
```

C.3.23 EQU Equate Symbol to a Value

```
<label>    EQU [{X: | Y: | L: | P: | E:}]<expression>
```

The **EQU** directive assigns the value and memory space attribute of <expression> to the symbol <label>. If <expression> has a memory space attribute of **N**one, then it can optionally be preceded by any of the indicated memory space qualifiers to force a memory space attribute. An error will occur if the expression has a memory space attribute other than **N**one and it is different than the forcing memory space attribute. The optional forcing memory space attribute is useful to assign a memory space attribute to an expression that consists only of constants but is intended to refer to a fixed address in a memory space.

The **EQU** directive is one of the directives that assigns a value other than the program counter to the label. The label cannot be redefined anywhere else in the program (or section, if **SECTION** directives are being used). The <expression> may be relative or absolute, but cannot include a symbol that is not yet defined (no forward references are allowed).

Note: See also **SET.**

Example B-44 EQU Directive

```
A_D_PORT   EQU      X:$4000
```

This would assign the value $4000 with a memory space attribute of **X** to the symbol A_D_PORT.

```
COMPUTE    EQU      @LCV(L)
```

@LCV(L) is used to refer to the value and memory space attribute of the load location counter. This value and memory space attribute would be assigned to the symbol COMPUTE.

C.3.24 EXITM Exit Macro

```
EXITM
```

The **EXITM** directive will cause immediate termination of a macro expansion. It is useful when used with the conditional assembly directive **IF** to terminate macro expansion when error conditions are detected.

Note: A label is not allowed with this directive.

Note: See also **DUP**, **DUPA**, **DUPC**, **MACRO.**

Example B-45 EXITM Directive

```
CALC       MACRO    XVAL,YVAL
           IF       XVAL<0
           FAIL     'Macro parameter value out of range'
           EXITM    ; Exit macro
           ENDIF
           .
           .
           .
           ENDM
```

C.3.25 FAIL Programmer Generated Error

```
FAIL    [{<str>|<exp>}[,{<str>|<exp>},...,{<str>|<exp>}]]
```

The **FAIL** directive will cause an error message to be output by the assembler. The total error count will be incremented as with any other error. The **FAIL** directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the error has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified optionally to describe the nature of the generated error.

Note: A label is not allowed with this directive.

Note: See also **MSG**, **WARN.**

Example B-46 FAIL Directive

```
        FAIL    'Parameter out of range'
```

C.3.26 FORCE Set Operand Forcing Mode

```
FORCE   {SHORT | LONG | NONE}
```

The **FORCE** directive causes the assembler to force all immediate, memory, and address operands to the specified mode as if an explicit forcing operator were used. Note that if a relocatable operand value forced short is determined to be too large for the instruction word, an error will occur at link time, not during assembly. Explicit forcing operators override the effect of this directive.

Note: A label is not allowed with this directive.

Note: See also **<**, **>**, **#<**, **#>.**

Example B-47 FORCE Directive

```
        FORCE   SHORT   ; force operands short
```

C.3.27 GLOBAL Global Section Symbol Declaration

```
GLOBAL     <symbol>[,<symbol>,...,<symbol>]
```

The **GLOBAL** directive is used to specify that the list of symbols is defined within the current section, and that those definitions should be accessible by all sections. This directive is only valid if used within a program block bounded by the **SECTION** and **ENDSEC** directives. If the symbols that appear in the operand field are not defined in the section, an error will be generated.

Note: A label is not allowed with this directive.

Note: See also **SECTION**, **XDEF**, **XREF.**

Example B-48 GLOBAL Directive

```
SECTION    IO
GLOBAL     LOOPA      ; LOOPA will be globally accessible by other sections
.
.
.
ENDSEC
```

C.3.28 GSET Set Global Symbol to a Value

```
<label>    GSET      <expression>
           GSET      <label>    <expression>
```

The **GSET** directive is used to assign the value of the expression in the operand field to the label. The **GSET** directive functions somewhat like the **EQU** directive. However, labels defined via the **GSET** directive can have their values redefined in another part of the program (but only through the use of another **GSET** or **SET** directive). The **GSET** directive is useful for resetting a global **SET** symbol within a section, where the **SET** symbol would otherwise be considered local. The expression in the operand field of a **GSET** must be absolute and cannot include a symbol that is not yet defined (no forward references are allowed).

Note: See also **EQU**, **SET.**

Example B-49 GSET Directive

```
COUNT      GSET      0          ; INITIALIZE COUNT
```

C.3.29 HIMEM Set High Memory Bounds

```
HIMEM      <mem>[<rl>]:<expression>[,...]
```

The **HIMEM** directive establishes an absolute high memory bound for code and data generation. <mem> corresponds to one of the DSP memory spaces (**X**, **Y**, **L**, **P**, **E**). <rl> is one of the letters **R** for runtime counter or **L** for load counter. The <expression> is an absolute integer value within the address range of the machine. If during assembly the specified location counter exceeds the value given by <expression>, a warning is issued.

Note: A label is not allowed with this directive.

Note: See also **LOMEM.**

Example B-50 HIMEM Directive

```
HIMEM          XR:$7FFF,YR:$7FFF          ; SET X/Y RUN HIGH MEM BOUNDS
```

C.3.30 IDENT Object Code Identification Record

```
[<label>]  IDENT      <expression1>,<expression2>
```

The **IDENT** directive is used to create an identification record for the object module. If <label> is specified, it will be used as the module name. If <label> is not specified, then the filename of the source input file is used as the module name. <expression1> is the version number; <expression2> is the revision number. The two expressions must each evaluate to an integer result. The comment field of the **IDENT** directive will also be passed on to the object module.

Note: See also **COBJ.**

Example B-51 IDENT Directive

If the following line was included in the source file,

```
FFILTER   IDENT    1,2      ; FIR FILTER MODULE
```

then the object module identification record would include the module name (FFILTER), the version number (1), the revision number (2), and the comment field (; FIR FILTER MODULE).

C.3.31 IF Conditional Assembly Directive

```
IF        <expression>
.
.
[ELSE]    (the ELSE directive is optional)
.
.
ENDIF
```

Part of a program that is to be conditionally assembled must be bounded by an **IF-ENDIF** directive pair. If the optional **ELSE** directive is not present, then the source statements following the **IF** directive and up to the next **ENDIF** directive will be included as part of the source file being assembled only if the <expression> has a nonzero result. If the <expression> has a value of zero, the source file will be assembled as if those statements between the **IF** and the **ENDIF** directives were never encountered. If the **ELSE** directive is present and <expression> has a nonzero result, then the statements between the **IF** and **ELSE** directives will be assembled, and the statements between the **ELSE** and **ENDIF** directives will be skipped. Alternatively, if <expression> has a value of zero, then the statements between the **IF** and **ELSE** directives will be skipped, and the statements between the **ELSE** and **ENDIF** directives will be assembled.

The <expression> must have an absolute integer result and is considered true if it has a nonzero result. The <expression> is false only if it has a result of 0. Because of the nature of the directive, <expression> must be known on pass one (no forward references allowed). **IF** directives can be nested to any level. The **ELSE** directive will always refer to the nearest previous **IF** directive as will the **ENDIF** directive.

Note: A label is not allowed with this directive.

Note: See also **ENDIF.**

Example B-52 IF Directive

```
IF        @LST>0
DUP       @LST      ; Unwind LIST directive stack
NOLIST
ENDM
ENDIF
```

C.3.32 INCLUDE Include Secondary File

```
INCLUDE    <string>  |  <<string>>
```

This directive is inserted into the source program at any point where a secondary file is to be included in the source input stream. The string specifies the filename of the secondary file. The filename must be compatible with the operating system and can include a directory specification. If no extension is given for the filename, a default extension of .ASM is supplied.

The file is searched for first in the current directory, unless the <<string>> syntax is used, or in the directory specified in <string>. If the file is not found, and the **-I** option was used on the command line that invoked the assembler, then the string specified with the **-I** option is prefixed to <string> and that directory is searched. If the <<string>> syntax is given, the file is searched for only in the directories specified with the **-I** option.

Note: A label is not allowed with this directive.

Note: See also **MACLIB.**

Example B-53 INCLUDE Directive

```
INCLUDE  'headers/io.asm'; Unix example
INCLUDE  'storage\mem.asm'; MS-DOS example
INCLUDE  <data.asm>    ; Do not look in current directory
```

C.3.33 LIST List the Assembly

```
LIST
```

Print the listing from this point on. The **LIST** directive will not be printed, but the subsequent source lines will be output to the source listing. The default is to print the source listing. If the **IL** option has been specified, the **LIST** directive has no effect when encountered within the source program.

The **LIST** directive actually increments a counter that is checked for a positive value and is symmetrical with respect to the **NOLIST** directive.

Note the following sequence:

```
; Counter value currently 1
LIST                  ; Counter value = 2
LIST                  ; Counter value = 3
NOLIST                ; Counter value = 2
NOLIST                ; Counter value = 1
```

The listing still would not be disabled until another **NOLIST** directive was issued.

Note: A label is not allowed with this directive.

Note: See also **NOLIST**, **OPT.**

Example B-54 LIST Directive

```
IF      LISTON
LIST            ; Turn the listing back on
ENDIF
```

C.3.34 LOCAL Local Section Symbol Declaration

```
LOCAL       <symbol>[,<symbol>,...,<symbol>]
```

The **LOCAL** directive is used to specify that the list of symbols is defined within the current section, and that those definitions are explicitly local to that section. It is useful in cases where a symbol is used as a forward reference in a nested section where the enclosing section contains a like-named symbol. This directive is only valid if used within a program block bounded by the **SECTION** and **ENDSEC** directives. The **LOCAL** directive must appear before <symbol> is defined in the section. If the symbols that appear in the operand field are not defined in the section, an error will be generated.

Note: A label is not allowed with this directive.

Note: See also **SECTION**, **XDEF**, **XREF.**

Example B-55 LOCAL Directive

```
SECTION   IO
LOCAL     LOOPA      ; LOOPA local to this section
.
.
.
ENDSEC
```

C.3.35 LOMEM Set Low Memory Bounds

```
LOMEM     <mem>[<rl>]:<expression>[,...]
```

The **LOMEM** directive establishes an absolute low memory bound for code and data generation. <mem> corresponds to one of the DSP memory spaces (**X**, **Y**, **L**, **P**, **E**). <rl> is one of the letters **R** for runtime counter or **L** for load counter. The <expression> is an absolute integer value within the address range of the machine. If during assembly the specified location counter falls below the value given by <expression>, a warning is issued.

Note: A label is not allowed with this directive.

Note: See also **HIMEM.**

Example B-56 LOMEM Directive

```
LOMEM          XR:$100,YR:$100          ; SET X/Y RUN LOW MEM BOUNDS
```

C.3.36 LSTCOL Set Listing Field Widths

```
LSTCOL  [<labw>[,<opcw>[,<oprw>[,<opc2w>[,<opr2w>[,<xw>[,<yw>]]]]]]]
```

Sets the width of the output fields in the source listing. Widths are specified in terms of column positions. The starting position of any field is relative to its predecessor except for the label field, which always starts at the same position relative to page left margin, program counter value, and cycle count display. The widths may be expressed as any positive absolute integer expression. However, if the width is not adequate to accommodate the contents of a field, the text is separated from the next field by at least one space.

Any field for which the default is desired may be null. A null field can be indicated by two adjacent commas with no intervening space or by omitting any trailing fields altogether. If the **LSTCOL** directive is given with no arguments all field widths are reset to their default values.

Note: A label is not allowed with this directive.

Note: See also **PAGE.**

Example B-57 LSTCOL Directive

```
LSTCOL        40,,,,,20,20   ; Reset label, X, and Y data field widths
```

C.3.37 MACLIB Macro Library

```
MACLIB    <pathname>
```

This directive is used to specify the <pathname> (as defined by the operating system) of a directory that contains macro definitions. Each macro definition must be in a separate file, and the file must be named the same as the macro with the extension .ASM added. For example, BLOCKMV.ASM would be a file that contained the definition of the macro called BLOCKMV.

If the assembler encounters a directive in the operation field that is not contained in the directive or mnemonic tables, the directory specified by <pathname> will be searched for a file of the unknown name (with the .ASM extension added). If such a file is found, the current source line will be saved, and the file will be opened for input as an **INCLUDE** file. When the end of the file is encountered, the source line is restored and processing is resumed. Because the source line is restored, the processed file must have a macro definition of the unknown directive name, or else an error will result when the source line is restored and processed. However, the processed file is not limited to macro definitions, and can include any legal source code statements.

Multiple **MACLIB** directives may be given, in which case the assembler will search each directory in the order in which it is encountered.

Note: A label is not allowed with this directive.

Note: See also **INCLUDE.**

Example B-58 MACLIB Directive

```
        MACLIB  'macros\mymacs\'; IBM PC example
        MACLIB  'fftlib/'     ; UNIX example
```

C.3.38 MACRO Macro Definition

```
<label>   MACRO     [<dummy argument list>]
          .
          .
          <macro definition statements>
          .
          .
          ENDM
```

The dummy argument list has the form:

```
[<dumarg>[,<dumarg>,...,<dumarg>]]
```

The required label is the symbol by which the macro will be called. If the macro is named the same as an existing assembler directive or mnemonic, a warning will be issued. This warning can be avoided with the **RDIRECT** directive.

The definition of a macro consists of three parts: the header, which assigns a name to the macro and defines the dummy arguments; the body, which consists of prototype or skeleton source statements; and the terminator. The header is the **MACRO** directive, its label, and the dummy argument list. The body contains the pattern of standard source statements. The terminator is the **ENDM** directive.

The dummy arguments are symbolic names that the macro processor will replace with arguments when the macro is expanded (called). Each dummy argument must obey the same rules as symbol names. Dummy argument names that are preceded by an underscore are not allowed. Within each of the three dummy argument fields, the dummy arguments are separated by commas. The dummy argument fields are separated by one or more blanks.

Macro definitions may be nested but the nested macro will not be defined until the primary macro is expanded.

Note: See also **DUP**, **DUPA**, **DUPC**, **DUPF**, **ENDM**

Example B-59 MACRO Directive

```
SWAP_SYM  MACRO     REG1,REG2  ;swap REG1,REG2 using X0 as temp
          MOVE      R\?REG1,X0
          MOVE      R\?REG2,R\?REG1
          MOVE      X0,R\?REG2
          ENDM
```

C.3.39 MODE Change Relocation Mode

```
MODE      <ABS[OLUTE] | REL[ATIVE]>
```

Causes the assembler to change to the designated operational mode. The **MODE** directive may be given at any time in the assembly source to alter the set of location counters used for section addressing. Code generated while in absolute mode will be placed in memory at the location determined during assembly. Relocatable code and data are based from the enclosing section start address. The **MODE** directive has no effect when the command line **-A** option is issued.

Note: A label is not allowed with this directive.

Note: See also **ORG.**

Example B-60 MODE Directive

```
          MODE      ABS        ; Change to absolute mode
```

C.3.40 MSG Programmer Generated Message

```
MSG         [{<str>|<exp>}[,{<str>|<exp>},...,{<str>|<exp>}]]
```

The **MSG** directive will cause a message to be output by the assembler. The error and warning counts will not be affected. The **MSG** directive is normally used in conjunction with conditional assembly directives for informational purposes. The assembly proceeds normally after the message has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified optionally to describe the nature of the message.

Note: A label is not allowed with this directive.

Note: See also **FAIL**, **WARN.**

Example B-61 MSG Directive

```
        MSG        'Generating sine tables'
```

C.3.41 NOLIST Stop Assembly Listing

```
NOLIST
```

Do not print the listing from this point on (including the **NOLIST** directive). Subsequent source lines will not be printed.

The **NOLIST** directive actually decrements a counter that is checked for a positive value and is symmetrical with respect to the **LIST** directive. Note the following sequence:

```
; Counter value currently 1
LIST                    ; Counter value = 2
LIST                    ; Counter value = 3
NOLIST                  ; Counter value = 2
NOLIST                  ; Counter value = 1
```

The listing still would not be disabled until another **NOLIST** directive was issued.

Note: A label is not allowed with this directive.

Note: See also **LIST**, **OPT.**

Example B-62 NOLIST Directive

```
        IF         LISTOFF
        NOLIST                 ; Turn the listing off
        ENDIF
```

C.3.42 OPT Assembler Options

```
OPT        <option>[,<option>,...,<option>][<comment>]
```

The **OPT** directive is used to designate the assembler options. Assembler options are given in the operand field of the source input file and are separated by commas. Options also may be specified using the command line **-O** option. All options have a default condition. Some options are reset to their default condition at the end of pass one. Some are allowed to have the prefix **NO** attached to them, which then reverses their meaning.

Note: A label is not allowed with this directive.

Options can be grouped by function into five different types:

1. Listing format control
2. Reporting options
3. Message control
4. Symbol options
5. Assembler operation

C.3.42.1 Listing Format Control

These options control the format of the listing file:

- **FC—**Fold trailing comments
- **FF—** Form feeds for page ejects
- **FM—**Format messages
- **PP**—Pretty print listing
- **RC—**Relative comment spacing

C.3.42.2 Reporting Options

These options control what is reported in the listing file:

- **CEX—**Print **DC** expansions
- **CL—**Print conditional assembly directives
- **CRE**—Print symbol cross-reference
- **DXL—**Expand **DEFINE** directive strings in listing

- **HDR—**Generate listing headers
- **IL—**Inhibit source listing
- **LOC—**Print local labels in cross-reference
- **MC**—Print macro calls
- **MD—**Print macro definitions
- **MEX**—Print macro expansions
- **MU—**Print memory utilization report
- **NL**—Print conditional assembly and section nesting levels
- **S**—Print symbol table
- **U—**Print skipped conditional assembly lines

C.3.42.3 Message Control

These options control the types of assembler messages that are generated:

- **AE—**Check address expressions
- **MSW—**Warn on memory space incompatibilities
- **UR**—Flag unresolved references
- **W—**Display warning messages

C.3.42.4 Symbol Options

These options deal with the handling of symbols by the assembler:

- **DEX—**Expand **DEFINE** symbols within quoted strings
- **IC—**Ignore case in symbol names
- **NS**—Support symbol scoping in nested sections
- **SCL—**Scope structured control statement labels
- **SCO—**Structured control statement labels to listing/object file
- **SO**—Write symbols to object file
- **XLL**—Write local labels to object file
- **XR—**Recognize **XDEF**ed symbols without **XREF**

C.3.42.5 Assembler Operation

Miscellaneous options having to do with internal assembler operation:

- **CC—**Enable cycle counts
- **CK—**Enable checksumming
- **CM**—Preserve comment lines within macros
- **CONST—**Make **EQU** symbols assembly time constants
- **CONTCK**—Continue checksumming
- **DLD—**Do not restrict directives in loops
- **GL—**Make all section symbols global
- **GS—**Make all sections global static
- **INTR—**Perform interrupt location checks
- **LB**—Byte increment load counter
- **LDB**—Listing file debug
- **MI—**Scan **MACLIB** directories for include files
- **PS**—Pack strings
- **PSM**—Programmable short addressing mode
- **RP—**Generate NOP to accommodate pipeline delay
- **RSV—**Check reserve data memory locations
- **SI**—Interpret short immediate as long or sign extended
- **SVO**—Preserve object file on errors

Following are descriptions of the individual options. The parenthetical inserts specify **default** if the option is the default condition, and **reset** if the option is reset to its default state at the end of pass one.

- **AE—**(default, reset) Check address expressions for appropriate arithmetic operations. For example, this will check that only valid add or subtract operations are performed on address terms.
- **CC—**Enable cycle counts and clear total cycle count. Cycle counts will be shown on the output listing for each instruction. Cycle counts assume a full instruction fetch pipeline and no wait states.
- **CEX—**Print **DC** expansions.
- **CK—**Enable checksumming of instruction and data values and clear cumulative checksum. The checksum value can be obtained using the **@CHK()** function.

- **CL—**(default, reset) Print the conditional assembly directives.
- **CM—**(default, reset) Preserve comment lines of macros when they are defined. Note that any comment line within a macro definition that starts with two consecutive semicolons (;;) is never preserved in the macro definition.
- **CONST—EQU** symbols are maintained as assembly time constants and will not be sent to the object file. This option, if used, must be specified before the first symbol in the source program is defined.
- **CONTC—**Re-enable cycle counts. Does not clear total cycle counts. The cycle count for each instruction will be shown on the output listing.
- **CONTCK—**Re-enable checksumming of instructions and data. Does not clear cumulative checksum value.
- **CRE—**Print a cross reference table at the end of the source listing. This option, if used, must be specified before the first symbol in the source program is defined.
- **DEX—**Expand **DEFINE** symbols within quoted strings. Can also be done on a case-by-case basis using double-quoted strings.
- **DLD—**Do not restrict directives in DO loops. The presence of some directives in DO loops does not make sense, including some **OPT** directive variations. This option suppresses errors on particular directives in loops.
- **DXL—**(default, reset) Expand **DEFINE** directive strings in listing.
- **FC—**Fold trailing comments. Any trailing comments that are included in a source line will be folded underneath the source line and aligned with the opcode field. Lines that start with the comment character will be aligned with the label field in the source listing. The **FC** option is useful for displaying the source listing on 80 column devices.
- **FF—**Use form feeds for page ejects in the listing file.
- **FM—**Format assembler messages so that the message text is aligned and broken at word boundaries.
- **GL—**Make all section symbols global. This has the same effect as declaring every section explicitly GLOBAL. This option must be given before any sections are defined explicitly in the source file.
- **GS—**(default, reset in absolute mode) Make all sections global static. All section counters and attributes will be associated with the GLOBAL section. This option must be given before any sections are defined explicitly in the source file.
- **HDR—**(default, reset) Generate listing header along with titles and subtitles.

- **IC—**Ignore case in symbol, section, and macro names. This directive must be issued before any symbols, sections, or macros are defined.
- **IL—**Inhibit source listing. This option will stop the assembler from producing a source listing.
- **INTR—**(default, reset in absolute mode) Perform interrupt location checks. Certain DSP instructions may not appear in the interrupt vector locations in program memory. This option enables the assembler to check for these instructions when the program counter is within the interrupt vector bounds.
- **LB—**Increment load counter (if different from runtime) by number of bytes in DSP word to provide byte-wide support for overlays in bootstrap mode. This option must appear before any code or data generation.
- **LDB—**Use the listing file as the debug source file rather than the assembly language file. The **-L** command line option to generate a listing file must be specified for this option to take effect.
- **LOC—**Include local labels in the symbol table and cross-reference listing. Local labels are not normally included in these listings. If neither the **S** or **CRE** options are specified, then this option has no effect. The **LOC** option must be specified before the first symbol is encountered in the source file.
- **MC—**(default, reset) Print macro calls.
- **MD—**(default, reset) Print macro definitions.
- **MEX—**Print macro expansions.
- **MI—**Scan **MACLIB** directory paths for include files. The assembler ordinarily looks for included files only in the directory specified in the **INCLUDE** directory or in the paths given by the **-I** command line option. If the **MI** option is used the assembler will also look for included files in any designated **MACLIB** directories.
- **MSW—**(default, reset) Issue warning on memory space incompatibilities.
- **MU—**Include a memory utilization report in the source listing. This option must appear before any code or data generation.
- **NL—**Display conditional assembly (**IF-ELSE-ENDIF**) and section nesting levels on listing.
- **NOAE—**Do not check address expressions.
- **NOCC—**(default, reset) Disable cycle counts. Does not clear total cycle count.
- **NOCEX—**(default, reset) Do not print **DC** expansions.

- **NOCK—**(default, reset) Disable checksumming of instruction and data values.
- **NOCL—**Do not print the conditional assembly directives.
- **NOCM—**Do not preserve comment lines of macros when they are defined.
- **NODEX—**(default, reset) Do not expand **DEFINE** symbols within quoted strings.
- **NODLD—**(default, reset) Restrict use of certain directives in DO loop.
- **NODXL—**Do not expand **DEFINE** directive strings in listing.
- **NOFC—**(default, reset) Inhibit folded comments.
- **NOFF—**(default, reset) Use multiple line feeds for page ejects in the listing file.
- **NOFM—**(default, reset) Do not format assembler messages.
- **NOGS—**(default, reset in relative mode) Do not make all sections global static.
- **NOHDR—**Do not generate listing header. This also turns off titles and subtitles.
- **NOINTR—**(default, reset in relative mode) Do not perform interrupt location checks.
- **NOMC—**Do not print macro calls.
- **NOMD—**Do not print macro definitions.
- **NOMEX**—(default, reset) Do not print macro expansions.
- **NOMI—**(default, reset) Do not scan **MACLIB** directory paths for include files.
- **NOMSW—**Do not issue warning on memory space incompatibilities.
- **NONL—**(default, reset) Do not display nesting levels on listing.
- **NONS—**Do not allow scoping of symbols within nested sections.
- **NOPP—**Do not pretty print listing file. Source lines are sent to the listing file as they are encountered in the source, with the exception that tabs are expanded to spaces and continuation lines are concatenated into a single physical line for printing.
- **NOPS—**Do not pack strings in **DC** directive. Individual bytes in strings will be stored one byte per word.
- **NORC—**(default, reset) Do not space comments relatively.

- **NORP—**(default, reset) Do not generate instructions to accommodate pipeline delay.
- **NOSCL—**Do not maintain the current local label scope when a structured control statement label is encountered.
- **NOU—**(default, reset) Do not print the lines excluded from the assembly due to a conditional assembly directive.
- **NOUR—**(default, reset) Do not flag unresolved external references.
- **NOW—**Do not print warning messages.
- **NS—**(default, reset) Allow scoping of symbols within nested sections.
- **PP—**(default, reset) Pretty print listing file. The assembler attempts to align fields at a consistent column position without regard to source file formatting.
- **PS—**(default, reset) Pack strings in **DC** directive. Individual bytes in strings will be packed into consecutive target words for the length of the string.
- **RC—**Space comments relatively in listing fields. By default, the assembler always places comments at a consistent column position in the listing file. This option allows the comment field to float: on a line containing only a label and opcode, the comment would begin in the operand field.
- **RP—**Generate NOP instructions to accommodate pipeline delay. If an address register is loaded in one instruction then the contents of the register is not available for use as a pointer until after the next instruction. Ordinarily when the assembler detects this condition it issues an error message. The **RP** option will cause the assembler to output a NOP instruction into the output stream instead of issuing an error.
- **S—**Print symbol table at the end of the source listing. This option has no effect if the **CRE** option is used.
- **SCL—**(default, reset) Structured control statements generate non-local labels that ordinarily are not visible to the programmer. This can create problems when local labels are interspersed among structured control statements. This option causes the assembler to maintain the current local label scope when a structured control statement label is encountered.
- **SCO—**Send structured control statement labels to object and listing files. Normally the assembler does not externalize these labels. This option must appear before any symbol definition.
- **SO—**Write symbol information to object file. This option is recognized but performs no operation in COFF assemblers.

- **SVO—**Preserve object file on errors. Normally any object file produced by the assembler is deleted if errors occur during assembly. This option must be given before any code or data is generated.
- **U—**Print the unassembled lines skipped due to failure to satisfy the condition of a conditional assembly directive.
- **UR—**Generate a warning at assembly time for each unresolved external reference. This option works only in relocatable mode.
- **W—**(default, reset) Print all warning messages.
- **WEX—**Add warning count to exit status. Ordinarily the assembler exits with a count of errors. This option causes the count of warnings to be added to the error count.
- **XLL—**Write underscore local labels to object file. This is primarily used to aid debugging. This option, if used, must be specified before the first symbol in the source program is defined.
- **XR—**Causes **XDEF**ed symbols to be recognized within other sections without being **XREF**ed. This option, if used, must be specified before the first symbol in the source program is encountered.

Example B-63 OPT Directive

```
        OPT         CEX,MEX     ; Turn on DC and macro expansions
        OPT         CRE,MU      ; Cross reference, memory utilization
```

C.3.43 ORG Initialize Memory Space and Location Counters

```
ORG        <rms>[<rlc>][<rmp>]:[<exp1>][,<lms>[<llc>][<lmp>]:[<exp2>]]
ORG        <rms>[<rmp>][(<rce>)]:[<exp1>][,<lms>[<lmp>][(<lce>)]:[<exp2>]]
```

The **ORG** directive is used to specify addresses and to indicate memory space and mapping changes. It also can designate an implicit counter mode switch in the assembler and serves as a mechanism for initiating overlays.

Note: A label is not allowed with this directive.

The parameters used with the ORG directive are:

- <rms>—Which memory space (X, Y, L, P, or E) will be used as the runtime memory space. If the memory space is **L**, any allocated datum with a value greater than the target word size will be extended to two words; otherwise, it is truncated. If the memory space is **E**, then depending on the memory space qualifier, any generated words will be split into bytes, one byte per word, or a 16/8-bit combination.

- <rlc>—Which runtime counter H, L, or default (if neither H or L is specified), that is associated with the <rms> will be used as the runtime location counter.
- <rmp>—Indicates the runtime physical mapping to DSP memory: I - internal, E - external, R - ROM, A - port A, B - port B. If not present, no explicit mapping is done.
- <rce>—Non-negative absolute integer expression representing the counter number to be used as the runtime location counter. Must be enclosed in parentheses. Should not exceed the value 65535.
- <exp1>—Initial value to assign to the runtime counter used as the <rlc>. If <exp1> is a relative expression the assembler uses the relative location counter. If <exp1> is an absolute expression the assembler uses the absolute location counter. If <exp1> is not specified, then the last value and mode that the counter had will be used.
- <lms>—Which memory space (X, Y, L, P, or E) will be used as the load memory space. If the memory space is **L**, any allocated datum with a value greater than the target word size will be extended to two words; otherwise, it is truncated. If the memory space is **E**, then depending on the memory space qualifier, any generated words will be split into bytes, one byte per word, or a 16/8-bit combination.
- <llc>—Which load counter, H, L, or default (if neither H or L is specified), that is associated with the <lms> will be used as the load location counter.
- <lmp>—Indicates the load physical mapping to DSP memory: I - internal, E - external, R - ROM, A - port A, B - port B. If not present, no explicit mapping is done.
- <lce>—Non-negative absolute integer expression representing the counter number to be used as the load location counter. Must be enclosed in parentheses. Should not exceed the value 65535.
- <exp2>—Initial value to assign to the load counter used as the <llc>. If <exp2> is a relative expression the assembler uses the relative location counter. If <exp2> is an absolute expression the assembler uses the absolute location counter. If <exp2> is not specified, then the last value and mode that the counter had will be used.

If the last half of the operand field in an **ORG** directive dealing with the load memory space and counter is not specified, then the assembler will assume that the load memory space and load location counter are the same as the runtime memory space and runtime location counter. In this case, object code is being assembled to be loaded into the address and memory space where it will be when the program is run, and is not an overlay.

If the load memory space and counter are given in the operand field, then the assembler always generates code for an overlay. Whether the overlay is absolute or relocatable depends upon the current operating mode of the assembler and whether the load counter value is an absolute or relative expression. If the assembler is running in absolute mode, or if the load counter expression is absolute, then the overlay is absolute. If the assembler is in relative mode and the load counter expression is relative, the overlay is relocatable. Runtime relocatable overlay code is addressed relative to the location given in the runtime location counter expression. This expression, if relative, may not refer to another overlay block.

Note: See also **MODE.**

Example B-64 ORG Directive

ORG P:$1000

Sets the runtime memory space to P. Selects the default runtime counter (counter 0) associated with P space to use as the runtime location counter and initializes it to $1000. The load memory space is implied to be P, and the load location counter is assumed to be the same as the runtime location counter.

ORG PHE:

Sets the runtime memory space to P. Selects the H load counter (counter 2) associated with P space to use as the runtime location counter. The H counter will not be initialized, and its last value will be used. Code generated hereafter will be mapped to external (E) memory. The load memory space is implied to be P, and the load location counter is assumed to be the same as the runtime location counter.

ORG PI:OVL1,Y:

Indicates code will be generated for an overlay. The runtime memory space is P, and the default counter is used as the runtime location counter. It will be reset to the value of OVL1. If the assembler is in absolute mode via the **-A** command line option then OVL1 must be an absolute expression. If OVL1 is an absolute expression the assembler uses the absolute runtime location counter. If OVL1 is a relocatable value the assembler uses the relative runtime location counter. In this case OVL1 must not itself be an overlay symbol (e.g. defined within an overlay block). The load memory space is Y. Since neither H, L, nor any counter expression was specified as the load counter, the default load counter (counter 0) will be used as the load location counter. The counter value and mode will be whatever it was the last time it was referenced.

Example B-64 ORG Directive (Continued)

```
ORG  XL:,E8:
```

Sets the runtime memory space to X. Selects the L counter (counter 1) associated with X space to use as the runtime location counter. The L counter will not be initialized, and its last value will be used. The load memory space is set to E, and the qualifier 8 indicates a bytewise RAM configuration. Instructions and data will be generated eight bits per output word with byte-oriented load addresses. The default load counter will be used and there is no explicit load origin.

```
ORG  P(5):,Y:$8000
```

Indicates code will be generated for an absolute overlay. The runtime memory space is P, and the counter used as the runtime location counter is counter 5. It will not be initialized, and the last previous value of counter 5 will be used. The load memory space is Y. Since neither H, L, nor any counter expression was specified as the load counter, the default load counter (counter 0) will be used as the load location counter. The default load counter will be initialized to $8000.

C.3.44 PAGE Top of Page/Size Page

```
PAGE        [<exp1>[,<exp2>...,<exp5>]]
```

The **PAGE** directive has two forms:

1. If no arguments are supplied, then the assembler will advance the listing to the top of the next page. In this case, the **PAGE** directive will not be output.
2. The **PAGE** directive with arguments can be used to specify the printed format of the output listing. Arguments may be any positive absolute integer expression. The arguments in the operand field (as explained below) are separated by commas. Any argument can be left as the default or last set value by omitting the argument and using two adjacent commas. The **PAGE** directive with arguments will not cause a page eject and will be printed in the source listing.

Note: A label is not allowed with this directive.

The arguments in order are:

- **PAGE_WIDTH <exp1>—**Page width in terms of number of output columns per line (default 80, min 1, max 255).

- **PAGE_LENGTH <exp2>—**Page length in terms of total number of lines per page (default 66, min 10, max 255). As a special case a page length of 0 (zero) turns off all headers, titles, subtitles, and page breaks.
- **BLANK_TOP <exp3>**—Blank lines at top of page. (default 0, min 0, max see below).
- **BLANK_BOTTOM <exp4>—**Blank lines at bottom of page. (default 0, min 0, max see below).
- **BLANK_LEFT <exp5>—**Blank left margin. Number of blank columns at the left of the page. (default 0, min 0, max see below).

The following relationships must be maintained:

```
BLANK_TOP + BLANK_BOTTOM <= PAGE_LENGTH - 10
BLANK_LEFT < PAGE_WIDTH
```

Note: See also **LSTCOL.**

Example B-65 PAGE Directive

```
PAGE        132,,3,3    ; Set width to132, 3 line top/bottom margins
PAGE                    ; Page eject
```

C.3.45 PMACRO Purge Macro Definition

```
PMACRO    <symbol>[,<symbol>,...,<symbol>]
```

The specified macro definition will be purged from the macro table, allowing the macro table space to be reclaimed.

Note: A label is not allowed with this directive.

Note: See also **MACRO.**

Example B-66 PMACRO Directive

```
PMACRO    MAC1,MAC2
```

This statement would cause the macros named MAC1 and MAC2 to be purged.

C.3.46 PRCTL Send Control String to Printer

```
PRCTL      <exp>I<string>,...,<exp>I<string>
```

PRCTL simply concatenates its arguments and ships them to the listing file (the directive line itself is not printed unless there is an error). <exp> is a byte expression and <string> is an assembler string. A byte expression would be used to encode non-printing control characters, such as ESC. The string may be of arbitrary length, up to the maximum assembler-defined limits.

PRCTL may appear anywhere in the source file and the control string will be output at the corresponding place in the listing file. However, if a **PRCTL** directive is the last line in the last input file to be processed, the assembler insures that all error summaries, symbol tables, and cross-references have been printed before sending out the control string. This is so a **PRCTL** directive can be used to restore a printer to a previous mode after printing is done. Similarly, if the **PRCTL** directive appears as the first line in the first input file, the control string will be output before page headings or titles.

The **PRCTL** directive only works if the **-L** command line option is given; otherwise it is ignored.

Note: A label is not allowed with this directive.

Example B-67 PRCTL Directive

```
PRCTL      $1B,'E'    ; Reset HP LaserJet printer
```

C.3.47 RADIX Change Input Radix for Constants

```
RADIX      <expression>
```

Changes the input base of constants to the result of <expression>. The absolute integer expression must evaluate to one of the legal constant bases (2, 10, or 16). The default radix is 10. The **RADIX** directive allows the programmer to specify constants in a preferred radix without a leading radix indicator. The radix prefix for base 10 numbers is the grave accent (`). Note that if a constant is used to alter the radix, it must be in the appropriate input base at the time the **RADIX** directive is encountered.

Note: A label is not allowed with this directive.

Example B-68 RADIX Directive

```
_RAD10    DC       10       ; Evaluates to hex A
          RADIX    2
_RAD2     DC       10       ; Evaluates to hex 2
          RADIX    `16
_RAD16    DC       10       ; Evaluates to hex 10
          RADIX    3        ; Bad radix expression
```

C.3.48 RDIRECT Remove Directive or Mnemonic from Table

```
RDIRECT   <direc>[,<direc>,...,<direc>]
```

The **RDIRECT** directive is used to remove directives from the assembler directive and mnemonic tables. If the directive or mnemonic that has been removed is later encountered in the source file, it will be assumed to be a macro. Macro definitions that have the same name as assembler directives or mnemonics will cause a warning message to be output unless the **RDIRECT** directive has been used to remove the directive or mnemonic name from the assembler's tables. Additionally, if a macro is defined through the **MACLIB** directive which has the same name as an existing directive or opcode, it will not automatically replace that directive or opcode as previously described. In this case, the **RDIRECT** directive must be used to force the replacement.

Since the effect of this directive is global, it cannot be used in an explicitly-defined section (see **SECTION** directive). An error will result if the **RDIRECT** directive is encountered in a section.

Note: A label is not allowed with this directive.

Example B-69 RDIRECT Directive

```
          RDIRECT  PAGE,MOVE
```

This would cause the assembler to remove the **PAGE** directive from the directive table and the **MOVE** mnemonic from the mnemonic table.

C.3.49 SCSJMP Set Structured Control Statement Branching Mode

```
SCSJMP     {SHORT | LONG | NONE}
```

The **SCSJMP** directive is analogous to the **FORCE** directive, but it only applies to branches generated automatically by structured control statements (See **Section C.4** on page C-64). There is no explicit way, as with a forcing operator, to force a branch short or long when it is produced by a structured control statement. This directive will cause all branches resulting from subsequent structured control statements to be forced to the specified mode.

Just like the **FORCE** pseudo-op, errors can result if a value is too large to be forced short. For relocatable code, the error may not occur until the linking phase.

Note: See also **FORCE**, **SCSREG.**

Note: A label is not allowed with this directive.

Example B-70 SCSJMP Directive

```
        SCSJMP    SHORT     ; force all subsequent SCS jumps short
```

C.3.50 SCSREG Reassign Structured Control Statement Registers

```
SCSREG     [<srcreg>[,<dstreg>[,<tmpreg>[,<extreg>]]]]
```

The **SCSREG** directive reassigns the registers used by structured control statement (SCS) directives. It is convenient for reclaiming default SCS registers when they are needed as application operands within a structured control construct. <srcreg> is ordinarily the source register for SCS data moves. <dstreg> is the destination register. <tmpreg> is a temporary register for swapping SCS operands. <extreg> is an extra register for complex SCS operations. With no arguments **SCSREG** resets the SCS registers to their default assignments.

The **SCSREG** directive should be used judiciously to avoid register context errors during SCS expansion. Source and destination registers may not necessarily be used strictly as source and destination operands. The assembler does no checking of reassigned registers beyond validity for the target processor. Errors can result when a structured control statement is expanded and an improper register reassignment has occurred. It is recommended that the **MEX** option (see the **OPT** directive) be used to examine structured control statement expansion for relevant constructs to determine default register usage and applicable reassignment strategies.

Note: See also **OPT** (**MEX**), **SCSJMP.**

Note: A label is not allowed with this directive.

Example B-71 SCSREG Directive

```
        SCSREG     Y0,B       ; reassign SCS source and dest. registers
```

C.3.51 SECTION Start Section

```
SECTION    <symbol>   [GLOBAL | STATIC | LOCAL]
.
.
<section source statements>
.
.
ENDSEC
```

The **SECTION** directive defines the start of a section. All symbols that are defined within a section have the <symbol> associated with them as their section name. This serves to protect them from like-named symbols elsewhere in the program. By default, a symbol defined inside any given section is private to that section unless the **GLOBAL** or **LOCAL** qualifier accompanies the **SECTION** directive.

Any code or data inside a section is considered an indivisible block with respect to relocation. Code or data associated with a section is independently relocatable within the memory space to which it is bound, unless the **STATIC** qualifier follows the **SECTION** directive on the instruction line.

Symbols within a section are generally distinct from other symbols used elsewhere in the source program, even if the symbol name is the same. This is true as long as the section name associated with each symbol is unique, the symbol is not declared public (**XDEF** / **GLOBAL**), and the **GLOBAL** or **LOCAL** qualifier is not used in the section declaration. Symbols that are defined outside of a section are considered global symbols and have no explicit section name associated with them. Global symbols may be referenced freely from inside or outside of any section, as long as the global symbol name does not conflict with another symbol by the same name in a given section.

If the **GLOBAL** qualifier follows the <section name> in the **SECTION** directive, then all symbols defined in the section until the next **ENDSEC** directive are considered global. The effect is as if every symbol in the section were declared with **GLOBAL**. This is useful when a section needs to be independently relocatable, but data hiding is not desired.

If the **STATIC** qualifier follows the <section name> in the **SECTION** directive, then all code and data defined in the section until the next **ENDSEC** directive are relocated in terms of the immediately enclosing section. The effect with respect to relocation is as if all code and data in the section were defined within the parent section. This is useful when a section needs data hiding, but independent relocation is not required.

If the **LOCAL** qualifier follows the <section name> in the **SECTION** directive, then all symbols defined in the section until the next **ENDSEC** directive are visible to the immediately enclosing section. The effect is as if every symbol in the section were defined within the parent section. This is useful when a section needs to be independently relocatable, but data hiding within an enclosing section is not required.

The division of a program into sections controls not only labels and symbols, but also macros and **DEFINE** directive symbols. Macros defined within a section are private to that section and are distinct from macros defined in other sections even if they have the same macro name. Macros defined outside of sections are considered global and may be used within any section. Similarly, **DEFINE** directive symbols defined within a section are private to that section and **DEFINE** directive symbols defined outside of any section are globally applied. There are no directives that correspond to **XDEF** for macros or **DEFINE** symbols, and therefore, macros and **DEFINE** symbols defined in a section can never be accessed globally. If global accessibility is desired, the macros and **DEFINE** symbols should be defined outside of any section.

Sections can be nested to any level. When the assembler encounters a nested section, the current section is stacked and the new section is used. When the **ENDSEC** directive of the nested section is encountered, the assembler restores the old section and uses it. The **ENDSEC** directive always applies to the most previous **SECTION** directive. Nesting sections provides a measure of scoping for symbol names, in that symbols defined within a given section are visible to other sections nested within it. For example, if section B is nested inside section A, then a symbol defined in section A can be used in section B without **XDEF**ing in section A or **XREF**ing in section B. This scoping behavior can be turned off and on with the **NONS** and **NS** options respectively (see the **OPT** directive, this chapter).

Sections may also be split into separate parts. That is, <section name> can be used multiple times with **SECTION** and **ENDSEC** directive pairs. If this occurs, then these separate (but identically named) sections can access each others symbols freely without the use of the **XREF** and **XDEF** directives. If the **XDEF** and **XREF** directives are used within one section, they apply to all sections with the same section name. The reuse of the section name is allowed to permit the program source to be arranged in an arbitrary manner (for example, all statements that reserve X space storage locations grouped together), but retain the privacy of the symbols for each section.

When the assembler operates in relative mode (the default), sections act as the basic grouping for relocation of code and data blocks. For every section defined in the source a set of location counters is allocated for each DSP memory space. These counters are used to maintain offsets of data and instructions relative to the beginning of the section. At link time sections can be relocated to an absolute address, loaded in a particular order, or linked contiguously as specified by the programmer. Sections which are split into parts or among files are logically recombined so that each section can be relocated as a unit.

Sections may be relocatable or absolute. In the assembler absolute mode (command line **-A** option) all sections are considered absolute. A full set of locations counters is reserved for each absolute section unless the **GS** option is given (see the **OPT** directive, this chapter). In relative mode, all sections are initially relocatable. However, a section or a part of a section may be made absolute either implicitly by using the **ORG** directive, or explicitly through use of the **MODE** directive.

Note: A label is not allowed with this directive.

Note: See also **MODE**, **ORG**, **GLOBAL**, **LOCAL**, **XDEF**, **XREF.**

Example B-72 SECTION Directive

```
        SECTION    TABLES     ; TABLES will be the section name
```

C.3.52 SET Set Symbol to a Value

```
<label>   SET   <expression>
          SET   <label>   <expression>
```

The **SET** directive is used to assign the value of the expression in the operand field to the label. The **SET** directive functions somewhat like the **EQU** directive. However, labels defined via the **SET** directive can have their values redefined in another part of the program (but only through the use of another **SET** directive). The **SET** directive is useful in establishing temporary or reusable counters within macros. The expression in the operand field of a **SET** must be absolute and cannot include a symbol that is not yet defined (no forward references are allowed).

Note: See also **EQU**, **GSET.**

Example B-73 SET Directive

```
COUNT    SET    0        ; INITIALIZE COUNT
```

C.3.53 STITLE Initialize Program Sub-Title

```
         STITLE   [<string>]
```

The **STITLE** directive initializes the program subtitle to the string in the operand field. The subtitle will be printed on the top of all succeeding pages until another **STITLE** directive is encountered. The subtitle is initially blank. The **STITLE** directive will not be printed in the source listing. An **STITLE** directive with no string argument will cause the current subtitle to be blank.

Note: A label is not allowed with this directive.

Note: See also **TITLE.**

Example B-74 STITLE Directive

```
         STITLE   'COLLECT SAMPLES'
```

C.3.54 SYMOBJ Write Symbol Information to Object File

```
SYMOBJ      <symbol>[,<symbol>,...,<symbol>]
```

The **SYMOBJ** directive causes information for each <symbol> to be written to the object file. This directive is recognized but currently performs no operation in COFF assemblers.

Note: A label is not allowed with this directive.

Example B-75 SYMOBJ

```
SYMOBJ      XSTART,HIRTN,ERRPROC
```

C.3.55 TABS Set Listing Tab Stops

```
TABS        <tabstops>
```

The **TABS** directive allows resetting the listing file tab stops from the default value of 8.

Note: A label is not allowed with this directive.

Note: See also **LSTCOL.**

Example B-76 TABS Directive

```
TABS    4       ; Set listing file tab stops to 4
```

C.3.56 TITLE Initialize Program Title

```
TITLE       [<string>]
```

The **TITLE** directive initializes the program title to the string in the operand field. The program title will be printed on the top of all succeeding pages until another **TITLE** directive is encountered. The title is initially blank. The **TITLE** directive will not be printed in the source listing. A **TITLE** directive with no string argument will cause the current title to be blank.

Note: A label is not allowed with this directive.

Note: See also **STITLE.**

Example B-77 TITLE Directive

```
        TITLE   'FIR FILTER'
```

C.3.57 UNDEF Undefine DEFINE Symbol

```
UNDEF       [<symbol>]
```

The **UNDEF** directive causes the substitution string associated with <symbol> to be released, and <symbol> will no longer represent a valid **DEFINE** substitution. See the **DEFINE** directive for more information.

Note: A label is not allowed with this directive.

Note: See also **DEFINE.**

Example B-78 UNDEF Directive

```
        UNDEF       DEBUG       ; UNDEFINES THE DEBUG SUBSTITUTION STRING
```

C.3.58 WARN Programmer Generated Warning

```
WARN        [{<str>|<exp>}[,{<str>|<exp>},...,{<str>|<exp>}]]
```

The **WARN** directive will cause a warning message to be output by the assembler. The total warning count will be incremented as with any other warning. The **WARN** directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the warning has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified optionally to describe the nature of the generated warning.

Note: A label is not allowed with this directive.

Note: See also **FAIL**, **MSG.**

Example B-79 WARN Directive

```
        WARN        'parameter too large'
```

C.3.59 XDEF External Section Symbol Definition

```
XDEF        <symbol>[,<symbol>,...,<symbol>]
```

The **XDEF** directive is used to specify that the list of symbols is defined within the current section, and that those definitions should be accessible by sections with a corresponding **XREF** directive. This directive is only valid if used within a program section bounded by the **SECTION** and **ENDSEC** directives. The **XDEF** directive must appear before <symbol> is defined in the section. If the symbols that appear in the operand field are not defined in the section, an error will be generated.

Note: A label is not allowed with this directive.

Note: See also **SECTION**, **XREF.**

Example B-80 XDEF Directive

```
SECTION   IO
XDEF      LOOPA      ; LOOPA will be accessible by sections with XREF
.
.
.
ENDSEC
```

C.3.60 XREF External Section Symbol Reference

```
XREF        <symbol>[,<symbol>,...,<symbol>]
```

The **XREF** directive is used to specify that the list of symbols is referenced in the current section, but is not defined within the current section. These symbols must either have been defined outside of any section or declared as globally accessible within another section using the **XDEF** directive. If the **XREF** directive is not used to specify that a symbol is defined globally and the symbol is not defined within the current section, an error will be generated, and all references within the current section to such a symbol will be flagged as undefined. The **XREF** directive must appear before any reference to <symbol> in the section.

Note: A label is not allowed with this directive.

Note: See also **SECTION**, **XDEF.**

Example B-81 XREF Directive

```
        SECTION   FILTER
        XREF      AA,CC,DD    ; XDEFed symbols within section
        .
        .
        .
        ENDSEC
```

C.4 STRUCTURED CONTROL STATEMENTS

An assembly language provides an instruction set for performing certain rudimentary operations. These operations in turn may be combined into control structures such as loops (FOR, REPEAT, WHILE) or conditional branches (IF-THEN, IF-THEN-ELSE). The assembler, however, accepts formal, high-level directives that specify these control structures, generating the appropriate assembly language instructions for their efficient implementation. This use of structured control statement directives improves the readability of assembly language programs, without compromising the desirable aspects of programming in an assembly language.

C.4.1 Structured Control Directives

The following directives are used for structured control. Note the leading period, which distinguishes these keywords from other directives and mnemonics. Structured control directives may be specified in either upper or lower case, but they must appear in the opcode field of the instruction line (e.g. they must be preceded either by a label, a space, or a tab).

.BREAK	**.ENDI**	**.LOOP**
.CONTINUE	**.ENDL**	**.REPEAT**
.ELSE	**.ENDW**	**.UNTIL**
.ENDF	**.FOR**	**.WHILE**
.IF		

In addition, the following keywords are used in structured control statements:

AND	**DOWNTO**	**TO**
BY	**OR**	
DO	**THEN**	

Note: AND, DO, and OR are reserved assembler instruction mnemonics.

C.4.2 Syntax

The formats for the **.BREAK**, **.CONTINUE**, **.FOR**, **.IF**, **.LOOP**, **.REPEAT**, and **.WHILE** statements are given in sections C.4.2.1 through C.4.2.7. Syntactic variables used in the formats are defined as follows:

- **<expression>—**A simple or compound expression (section C.4.3).
- **<stmtlist>—**Zero or more assembler directives, structured control statements, or executable instructions.

Note: An assembler directive occurring within a structured control statement is examined exactly once—at assembly time. Thus the presence of a directive within a **.FOR**, **.LOOP**, **.REPEAT**, or **.WHILE** statement does not imply repeated occurrence of an assembler directive; nor does the presence of a directive within an **.IF-THEN-.ELSE** structured control statement imply conditional assembly.

- **<op1>—**A user-defined operand whose register/memory location holds the **.FOR** loop counter. The effective address must use a memory alterable addressing mode (e.g., it cannot be an immediate value).
- **<op2>—**The initial value of the **.FOR** loop counter. The effective address may be any mode, and may represent an arbitrary assembler expression.
- **<op3>—**The terminating value of the **.FOR** loop counter. The effective address may be any mode, and may represent an arbitrary assembler expression.
- **<op4>—**The step (increment/decrement) of the **.FOR** loop counter each time through the loop. If not specified, it defaults to a value of #1. The effective address may be any mode, and may represent an arbitrary assembler expression.
- **<cnt>—**The terminating value in a **.LOOP** statement. This can be any arbitrary assembler expression.

All structured control statements may be followed by normal assembler comments on the same logical line.

C.4.2.1 .BREAK Statement

.BREAK

The **.BREAK** statement causes an immediate exit from the innermost enclosing loop construct (**.WHILE**, **.REPEAT**, **.FOR**, **.LOOP**). A **.BREAK** statement does not exit an **.IF-THEN-.ELSE** construct. If a **.BREAK** is encountered with no loop statement active, a warning is issued.

Note: **.BREAK** should be used with care near **.ENDL** directives or near the end of DO loops. It generates a jump instruction which is illegal in those contexts.

Example B-82 .BREAK Statement

```
.WHILE     x:(r1)+ <GT> #0;loop until zero is found
.
.
.
.IF        <cs>
.BREAK                 ;causes exit from WHILE loop
.ENDI
.
.                      ;any instructions here are skipped
.
.ENDW
;execution resumes here after .BREAK
```

C.4.2.2 .CONTINUE Statement

.CONTINUE

The **.CONTINUE** statement causes the next iteration of a looping construct (**.WHILE**, **.REPEAT**, **.FOR**, **.LOOP**) to begin. This means that the loop expression or operand comparison is performed immediately, bypassing any subsequent instructions. If a **.CONTINUE** is encountered with no loop statement active, a warning is issued.

Note: **.CONTINUE** should be used with care near **.ENDL** directives or near the end of DO loops. It generates a jump instruction which is illegal in those contexts.

Note: One or more **.CONTINUE** directives inside a **.LOOP** construct will generate a NOP instruction just before the loop address.

Example B-83 .CONTINUE Statement

```
.REPEAT
.
.
.
.IF       <cs>
.CONTINUE           ;causes immediate jump to .UNTIL
.ENDI
.
.                   ;any instructions here are skipped
.
.UNTIL    x:(r1)+ <EQ> #0;evaluation here after .CONTINUE
```

C.4.2.3 .FOR Statement

```
.FOR <op1> = <op2>  {TO | DOWNTO} <op3> [BY <op4>] [DO]
<stmtlist>
.ENDF
```

Initialize <op1> to <op2> and perform <stmtlist> until <op1> is greater (**TO**) or less than (**DOWNTO**) <op3>. Makes use of a user-defined operand, <op1>, to serve as a loop counter. **.FOR-TO** allows counting upward, while **.FOR-DOWNTO** allows counting downward. The programmer may specify an increment/decrement step size in <op4>, or elect the default step size of #1 by omitting the **BY** clause. A **.FOR-TO** loop is not executed if <op2> is greater than <op3> upon entry to the loop. Similarly, a **.FOR-DOWNTO** loop is not executed if <op2> is less than <op3>.

<op1> must be a writable register or memory location. It is initialized at the beginning of the loop, and updated at each pass through the loop. Any immediate operands must be preceded by a pound sign (**#**). Memory references must be preceded by a memory space qualifier (X:, Y:, or P:). L memory references are not allowed. Operands must be or refer to single-word values.

The logic generated by the **.FOR** directive makes use of several DSP data registers. In fact, two data registers are used to hold the step and target values, respectively, throughout the loop; they are never reloaded by the generated code. It is recommended that these registers not be used within the body of the loop, or that they be saved and restored prior to loop evaluation.

Note: The **DO** keyword is optional.

Example B-84 .FOR Statement

```
    .FOR      X:CNT  =  #0  TO  Y:(targ*2)+114; loop on X:CNT
    .
    .
    .
    .ENDF
```

C.4.2.4 .IF Statement

```
.IF       <expression>[THEN]
<stmtlist>
[.ELSE
<stmtlist>]
.ENDI
```

If <expression> is true, execute <stmtlist> following **THEN** (the keyword **THEN** is optional); if <expression> is false, execute <stmtlist> following **.ELSE**, if present; otherwise, advance to the instruction following **.ENDI**.

Note: In the case of nested **.IF-THEN-.ELSE** statements, each **.ELSE** refers to the most recent **.IF-THEN** sequence.

Example B-85 .IF Statement

```
    .IF       <EQ>       ; zero bit set?
    .
    .
    .
    .ENDI
```

C.4.2.5 .LOOP Statement

```
.LOOP  <cnt>
<stmtlist>
.ENDL
```

Execute <stmtlist> <cnt> times. This is similar to the **.FOR** loop construct, except that the initial counter and step value are implied to be #1. It is actually a shorthand method for setting up a hardware DO loop on the DSP, without having to worry about addressing modes or label placement.

Since the **.LOOP** statement generates instructions for a hardware DO loop, the same restrictions apply as to the use of certain instructions near the end of the loop, nesting restrictions, etc. One or more **.CONTINUE** directives inside a **.LOOP** construct will generate a NOP instruction just before the loop address.

Example B-86 .LOOP Statement

```
.LOOP      LPCNT      ; hardware loop LPCNT times
.
.
.
.ENDL
```

C.4.2.6 .REPEAT Statement

```
.REPEAT
<stmtlist>
.UNTIL  <expression>
```

<stmtlist> is executed repeatedly until <expression> is true. When expression becomes true, advance to the next instruction following **.UNTIL**. The <stmtlist> is executed at least once, even if <expression> is true upon entry to the **.REPEAT** loop.

Example B-87 .REPEAT Statement

```
.REPEAT
.
.
.
.UNTIL     x:(r1)+ <EQ> #0; loop until zero is found
```

C.4.2.7 .WHILE Statement

```
.WHILE    <expression>[DO]
<stmtlist>
.ENDW
```

The <expression> is tested before execution of <stmtlist>. While <expression> remains true, <stmtlist> is executed repeatedly. When <expression> evaluates false, advance to the instruction following the **.ENDW** statement. If <expression> is false upon entry to the **.WHILE** loop, <stmtlist> is not executed; execution continues after the **.ENDW** directive.

Note: The **DO** keyword is optional.

Example B-88 .WHILE Statement

```
        .WHILE     x:(r1)+ <GT> #0; loop until zero is found
        .
        .
        .
        .ENDW
```

C.4.3 Simple and Compound Expressions

Expressions are an integral part of **.IF**, **.REPEAT**, and **.WHILE** statements. Structured control statement expressions should not be confused with the assembler expressions. The latter are evaluated at assembly time and will be referred to here as "assembler expressions"; they can serve as operands in structured control statement expressions. The structured control statement expressions described below are evaluated at run time and will be referred to in the following discussion simply as “expressions”.

A structured control statement expression may be simple or compound. A compound expression consists of two or more simple expressions joined by either **AND** or **OR** (but not both in a single compound expression).

C.4.3.1 Simple Expressions

Simple expressions are concerned with the bits of the Condition Code Register (CCR). These expressions are of two types. The first type merely tests conditions currently specified by the contents of the CCR (section C.4.3.2). The second type sets up a comparison of two operands to set the condition codes, and afterwards tests the codes.

C.4.3.2 Condition Code Expressions

A variety of tests (identical to those in the Jcc instruction) may be performed, based on the CCR condition codes. The condition codes, in this case, are preset by either a user-generated instruction or a structured operand-comparison expression. Each test is expressed in the structured control statement by a mnemonic enclosed in angle brackets.

When processed by the assembler, the expression generates an inverse conditional jump to beyond the matching **.ENDx/.UNTIL** directive.

Example B-89 Condition Code Expression

```
          .IF         <EQ>       ;zero bit set?
+         bne         Z_L00002   ;code generated by assembler
          CLR         D1         ;user code
          .ENDI
+         Z_L00002               ;assembler-generated label
          .REPEAT                ;subtract until D0 < D7
+         Z_L00034               ;assembler-generated label
          SUB         D7,D0      ;user code
          .UNTIL      <LT>
+         bge         Z_L00034   ;code generated by assembler
```

C.4.3.3 Operand Comparison Expressions

Two operands may be compared in a simple expression, with subsequent transfer of control based on that comparison. Such a comparison takes the form:

```
<op1>  <cc>  <op2>
```

where <cc> is a condition mnemonic enclosed in angle brackets (as described in section C.4.3.2), and <op1> and <op2> are register or memory references, symbols, or assembler expressions. When processed by the assembler, the operands are arranged such that a compare/jump sequence of the following form always results:

```
CMP         <reg1>,<reg2>
(J|B)cc     <label>
```

where the jump conditional is the inverse of <cc>. Ordinarily <op1> is moved to the <reg1> data register and <op2> is moved to the <reg2> data register prior to the compare. This is not always the case, however: if <op1> happens to be <reg2> and <op2> is <reg1>, an intermediate register is used as a scratch register. In any event, worst case code generation for a given operand comparison expression is generally two moves, a compare, and a conditional jump.

Jumps or branches generated by structured control statements are forced long because the number and address of intervening instructions between a control statement and its termination are not known by the assembler. The programmer may circumvent this behavior by use of the **SCSJMP** directive.

Any immediate operands must be preceded by a pound sign (**#**). Memory references must be preceded by a memory space qualifier (X:, Y:, or P:). L memory references are not allowed. Operands must be or refer to single-word values.

Note that values in the <reg1> and <reg2> data registers are not saved before expression evaluation. This means that any user data in those registers will be overwritten each time the expression is evaluated at runtime. The programmer should take care either to save needed contents of the registers, reassign data registers using the **SCSREG** directive, or not use them at all in the body of the particular structured construct being executed.

C.4.3.4 Compound Expressions

A compound expression consists of two or more simple expressions (section C.4.3.1) joined by a logical operator (**AND** or **OR**). The boolean value of the compound expression is determined by the boolean values of the simple expressions and the nature of the logical operator. Note that the result of mixing logical operators in a compound expression is undefined:

```
.IF     X1  <GT>  B  AND  <LS>  AND  R1  <NE>  R2;this is OK
.IF     X1  <LE>  B  AND  <LC>  OR   R5  <GT>  R6;undefined
```

The simple expressions are evaluated left to right. Note that this means the result of one simple expression could have an impact on the result of subsequent simple expressions, because of the condition code settings stemming from the assembler-generated compare.

If the compound expression is an **AND** expression and one of the simple expressions is found to be false, any further simple expressions are not evaluated. Likewise, if the compound expression is an **OR** expression and one of the simple expressions is found to be true, any further simple expressions are not evaluated. In these cases, the compound expression is either false or true, respectively, and the condition codes reflect the result of the last simple expression evaluated.

C.4.3.5 Statement Formatting

The format of structured control statements differs somewhat from normal assembler usage. Whereas a standard assembler line is split into fields separated by blanks or tabs, with no white space inside the fields, structured control statement formats vary depending on the statement being analyzed. In general, all structured control directives are placed in the opcode field (with an optional label in the label field) and white space separates all distinct fields in the statement. Any structured control statement may be followed by a comment on the same logical line.

C.4.3.6 Expression Formatting

Given an expression of the form:

```
<op1>  <LT>  <op2>  OR  <op3>  <GE>  <op4>
```

there must be white space (blank, tab) between all operands and their associated operators, including boolean operators in compound expressions. Moreover, there must be white space between the structured control directive and the expression, and between the expression and any optional directive modifier (**THEN**, **DO**). An assembler expression used as an operand in a structured control statement expression must not have white space in it, since it is parsed by the standard assembler evaluation routines:

```
.IF     #@CVI(@SQT(4.0))  <GT>  #2; no white space in first operand
```

C.4.3.7 .FOR/.LOOP Formatting

The **.FOR** and **.LOOP** directives represent special cases. The **.FOR** structured control statement consists of several fields:

```
.FOR  <op1>  =  <op2>  TO  <op3>  BY  <op4>  DO
```

There must be white space between all operands and other syntactic entities such as **=**, **TO**, **BY**, and **DO**. As with expression formatting, an assembler expression used as an operand must not have white space in it:

```
.FOR  X:CNT  =  #0  TO  Y:(targ*2)+1  BY  #@CVI(@POW(2.0,@CVF(R)))
```

In the example above, the **.FOR** loop operands represented as assembler expressions (symbol, function) do not have embedded white space, whereas the loop operands are always separated from structured control statement keywords by white space.

The count field of a **.LOOP** statement must be separated from the **.LOOP** directive by white space. The count itself may be any arbitrary assembler expression, and therefore must not contain embedded blanks.

C.4.4 Assembly Listing Format

Structured control statements begin with the directive in the opcode field; any optional label is output in the label field. The rest of the statement is left as is in the operand field, except for any trailing comment; the X and Y data movement fields are ignored. Comments following the statement are output in the comment field (unless the unreported comment delimiter is used).

Statements are expanded using the macro facilities of the assembler. Thus the generated code can be sent to the listing by specifying the **MEX** assembler option, either via the **OPT** directive or the **-O** command line option.

C.4.5 Effects on the Programmer's Environment

During assembly, global labels beginning with "**Z_L**" are generated. They are stored in the symbol table and should not be duplicated in user-defined labels. Because these non-local labels ordinarily are not visible to the programmer there can be problems when local (underscore) labels are interspersed among structured control statements. The **SCL** option (see the **OPT** directive) causes the assembler to maintain the current local label scope when a structured control statement label is encountered.

In the**.FOR** loop, <op1> is a user-defined symbol. When exiting the loop, the memory / register assigned to this symbol contains the value which caused the exit from the loop.

A compare instruction is produced by the assembler whenever two operands are tested in a structured statement. At runtime, these assembler-generated instructions set the condition codes of the CCR (in the case of a loop, the condition codes are set repeatedly). Any user-written code either within or following a structured statement that references CCR directly (move) or indirectly (conditional jump / transfer) should be attentive to the effect of these instructions.

Jumps or branches generated by structured control statements are forced long because the number and address of intervening instructions between a control statement and its termination are not known by the assembler. The programmer may circumvent this behavior by use of the SCSJMP directive. In all structured control statements except those using only a single condition code expression, registers are used to set up the required counters and comparands. In some cases, these registers are effectively reserved; the.FOR loop uses two data registers to hold the step and target values, respectively, and performs no save / restore operations on these registers. The assembler, in fact, does no save / restore processing in any structured control operation; it simply moves the operands into appropriate registers to execute the compare. The SCSREG directive may be used to reassign structured control statement registers. The MEX assembler option (see the OPT directive may be used to send the assembler-generated code to the listing file for examination of possible register use conflicts.

APPENDIX D

CODEC PROGRAMMING TUTORIAL

D.1 INTRODUCTION

OK, you've got a new toy and you can't wait to start developing application. But first you've got the task of reading and understanding all the seemingly endless pages of specifications, manuals, and related documentation. How are you going to get that killer application ready quickly and sold in time to make this month's mortgage? Well, we can't promise help with your banker, but we can save you some time by easing that initial foray into the codec documentation.

D.2 PROGRAMMING THE CODEC

The good news is that the Crystal Semiconductor CS4215 codec ("COder-DECoder") used on the DSP56303EVM offers a myriad of options that are all user-programmable via software. The bad news is that you have to learn how to program it.

A first scan of the CS4215 codec literature might lead you to conclude that the greatest obstacle between you and your goals is in learning to program the myriad of user-programmable features. To help you begin, there is an example program named ECHO.ASM that is included in the software shipped with the DSP56303EVM kit. This program is our effort to isolate the user from the details of programming the various CS4215 codec options. This appendix is provided to lead a new user through ECHO.ASM and help to explain how to use the tools developed to let you communicate most expediently with the analog world. So, let's get started.

D.3 ECHO.ASM PROGRAM DESCRIPTION

The ECHO.ASM program actually creates one of two versions from the same source code file. One version is the stand-alone program that is examined in this appendix. The other a version is loaded into the FLASH EPROM as a part of the code shipped with the self-test program.

A batch file MAKEECHO.BAT included with the software allows the user to create the stand-alone demonstration. A command line entry uses parameters that invoke the DSP56300 assembler program, selects the stand-alone version of the routine, and generates an executable module. The command line, which can either be typed in or executed from a batch file, is:

```
asm56300 -d STANDALONE 1 -a -b -l -g echo.asm
```

The command line entry generates the ECHO.ASM file. The following sections described the contents and format of this file. Begin by viewing the contents of the ECHO.ASM file. Set up the listing format as 132 columns and 60 line per page and display the file, or print it out on a printer.

D.3.1 Source Code Description

The ECHO.ASM file starts with a banner followed by a general description of the program and copyright notice:

Example B-90 Program Description

```
            page    132,60
;*****************************************************************************
;           ECHO.ASM Ver.2.0
;           Example program to move audio through CS4215
;                This program uses 2k samples of delay
;                to add a noticeable echo to the audio.
;
;           Copyright (c) MOTOROLA 1995, 1996
;                            Semiconductor Products Sector
;                            Digital Signal Processing Division
;
;*****************************************************************************
```

This is followed by the following sections of code:

- Included files
- Constant definitions
- Interrupt buffers
- Sample program listing

D.3.1.1 Included Files

The file includes references to four included files:

Example B-91 Included Files

```
;--------------------------------------------
          IF (STANDALONE==1)

          nolist
          include 'ioequ.asm'
          include 'intequ.asm'
          include 'ada_equ.asm'
          include 'vectors.asm'
          list

;*****************************************************************************
;         the following EQUates will define the operational parameters
;         of the codec. Please refer to the ADA_EQU.ASM source file
;         for a description of the parameters selections available. The
;         variables defined by the EQUates are sent to the codec via
;         the transmit buffer, TX_BUFF.
;
;         Since the parameters are defined in ADA_EQU.ASM, these lines must
;         follow the include statement.
;*****************************************************************************
```

These files perform the following functions:

- IOEQU.ASM defines a standard set of symbolic names for the addresses of the on-chip peripheral registers.
- INTEQU.ASM sets up the interrupt definitions.
- ADA_EQU.ASM defines the registers, parameters and bit mapping of the CS4215. This is the key to facilitating the selection of the codec's functions.
- VECTORS.ASM defines the interrupt vectors required by the interrupt handlers.

D.3.1.2 Constant Definitions

The next 4 lines of the program construct the constants that define the feature selections which are made during initialization of the codec.

Example B-92 Constant Definitions

```
CTRL_WD_12 equ NO_PREAMP+HI_PASS_FILT+SAMP_RATE_48+STEREO+DATA_16;CLB=0
CTRL_WD_34 equ IMMED_3STATE+XTAL1_SELECT+BITS_64+CODEC_MASTER
CTRL_WD_56 equ $000000
CTRL_WD_78 equ $000000

TONE_OUTPUT EQU HEADPHONE_EN+LINEOUT_EN+(0*LEFT_ATTN)+(0*RIGHT_ATTN)
TONE_INPUT EQU MIC_IN_SELECT+(15*MONITOR_ATTN)+(8*LEFT_GAIN)+(8*RIGHT_GAIN)
```

Among the features initialized at this time are sample rate, data format, clock selection and interface mechanism. These are features which generally require re-initialization when altered. By referring to the ADA_EQU.ASM file, the user can view the constants available for specifying the different feature options. The two lines that follow construct ‘tone_output’ and ‘tone_input’ are the constants that define those feature selections available when the analog subsystem is running. This includes such features as gain, mixer and attenuator settings.

D.3.1.3 Interrupt Buffers

The CS4215 interrupt routines used by this software are based on 2 four word buffers, one each for transmit data and receive data. The lines shown in allocate eight words of data memory, starting at address x:000000. These are the buffers used by the interrupt handler.

Example B-93 Interrupt Buffers

```
;---Buffer for talking to the CS4215

      org    x:0
          RX_BUFF_BASE equ *
          RX_data_1_2 ds 1      ;data time slot 1/2 for RX ISR
          RX_data_3_4 ds 1      ;data time slot 3/4 for RX ISR
          RX_data_5_6 ds 1      ;data time slot 5/6 for RX ISR
          RX_data_7_8 ds 1      ;data time slot 7/8 for RX ISR

          TX_BUFF_BASE equ *
          TX_data_1_2 ds 1      ;data time slot 1/2 for TX ISR
          TX_data_3_4 ds 1      ;data time slot 3/4 for TX ISR
          TX_data_5_6 ds 1      ;data time slot 5/6 for TX ISR
          TX_data_7_8 ds 1      ;data time slot 7/8 for TX ISR

          RX_PTR ds 1           ;Pointer for rx buffer
          TX_PTR ds 1           ;Pointer for tx buffer
```

D.3.1.4 Sample Program

This portion of the code starts at P:000000. This code initializes the DSP and codec and performs the example operations. It begins by setting the on-chip PLL multiplier to run the DSP at a higher frequency. The number of external bus wait states is set up (1 one Wait State in all external spaces is selected here). The program runs initializes all the internal registers and buffers. Then a separate program initializes the codec. Finally, the last part of the program processes the analog data as specified. This last part ends the conditional assembly code (ENDIF).

Example B-94 Sample Program Listing

```
        org P:$100
START
main
           movep #$040005,x:M_PCTL          ;PLL = 6 x 16.9344Mhz = 101.6Mhz
           movep #012421,x:M_BCR
           ENDIF
;------------------------------


           org p:
echo
           movec #0,SP                      ; clear stack pointer
           move #0,SC                       ; clear stack counter
           ori #3,mr                        ; disable interrupts
           movep #$040005,x:M_PCTL          ; PLL = 16.9344 x 6 = 101.6Mhz
           movep #$11,x:M_TCSR0             ; turn LED off
           move #$40,r6                     ; initialize stack pointer
           move #-1,m6                      ; linear addressing
           move #RX_BUFF_BASE,x0
           move x0,x:RX_PTR                 ; Initialize the rx pointer
           move #TX_BUFF_BASE,x0
           move x0,x:TX_PTR                 ; Initialize the tx pointer

; --- INIT THE CODEC ---
           jsr ada_init                     ; Jump to initialize the codec
           move #$0400,r4                   ; echo buffer starts at $400
           move #$03FF,m4                   ;...and is 1024 deep
```

Example B-94 Sample Program Listing (Continued)

```
;=========================================================================
;       this is where the work gets done...
;
loop_1
          jset #2,x:M_SSISR0,*              ; Wait for frame sync to pass.
          jclr #2,x:M_SSISR0,*              ; Wait for frame sync.

          move x:RX_BUFF_BASE,a             ; get new samples
          move x:RX_BUFF_BASE+1,b
          asr a x:(r4),x0                   ; divide them by 2 and get oldest
          asr b y:(r4),y0                   ;   samples from buffer
          add x0,a                          ; add the new samples and the old
          add y0,b
          asr a                             ; reduce magnitude of new data
          asr b
          move a,x:(r4)                     ; save the altered samples
          move b,y:(r4)+                    ; and bump the pointer

          move a,x:TX_BUFF_BASE             ; Put value in left channel tx.
          move b,x:TX_BUFF_BASE+1           ; Put value in right channel tx.
          move #TONE_OUTPUT,y0              ; headphones, line out, mute spkr, no attn.
          move y0,x:TX_BUFF_BASE+2
          move #TONE_INPUT,y0               ; no input gain, monitor mute
          move y0,x:TX_BUFF_BASE+3
          jmp loop_1                        ; Loop back.

          IF  (STANDALONE==1)
          nolist
          include 'ada_init.asm'            ; load the code init routines
          list
          ENDIF
```

INDEX

Symbols

A

E

F

P