DSP56603EVM

User's Manual

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

This document (and other documents) can be viewed on the World Wide Web at http://www.motorola-dsp.com.

OnCE™ is a trademark of Motorola, Inc.

© MOTOROLA INC., 1996, 1997

Order this document by **DSP56603EMUM/AD**

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not authorized for use as components in life support devices or systems intended for surgical implant into the body or intended to support or sustain life. Buyer agrees to notify Motorola of any such intended end use whereupon Motorola shall determine availability and suitability of its product or products for the use intended. Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Employment Opportunity / Affirmative Action Employer.

TABLE OF CONTENTS

SECTION 1 QUICK START GUIDE. 1-1
1.1 OVERVIEW 1-3
1.2 EQUIPMENT 1-3
1.2.1 What You Get with the DSP56603EVM. 1-3
1.2.2 What You Need to Supply 1-4
1.3 INSTALLATION PROCEDURE 1-5
1.3.1 Preparing the DSP56603EVM 1-5
1.3.2 Connecting the DSP56603EVM to the PC and Power 1-7
1.3.3 Installing the Software 1-8
1.3.4 Testing the DSP56603EVM. 1-9
1.3.4.1 DSP56603EVM Self-Test 1-9
1.3.4.2 DSP56603EVM Analysis Program 1-10

SECTION 2 EXAMPLE TEST PROGRAM 2-1
2.1 OVERVIEW 2-3
2.2 WRITING THE PROGRAM 2-4
2.2.1 Source Statement Format 2-4
2.2.1.1 Label Field. 2-4
2.2.1.2 Operation Field 2-5
2.2.1.3 Operand Field 2-5
2.2.1.4 Data Transfer Fields 2-5
2.2.1.5 Comment Field 2-5
2.2.2 Example Program 2-6
2.3 ASSEMBLING THE PROGRAM 2-8
2.3.1 Assembler Command Format 2-8
2.3.2 Assembler Options 2-8
2.3.3 Assembler Directives 2-11
2.3.3.1 Assembler Significant Characters 2-11
2.3.3.2 Assembly Control 2-12
2.3.3.3 Symbol Definition 2-13
2.3.3.4 Data Definition/Storage Allocation 2-13
2.3.3.5 Listing Control and Options 2-14

2.3.3.6 Object File Control Directives 2-14
2.3.3.7 Macros and Conditional Assembly 2-14
2.3.3.8 Structured Programming Directives 2-15
2.3.4 Assembling the Example Program 2-15
2.4 MOTOROLA DSP LINKER 2-16
2.4.1 Linker Options 2-16
2.4.2 Linker Directives 2-22
2.5 INTRODUCTION TO THE DEBUGGER SOFTWARE 2-23
2.6 RUNNING THE PROGRAM 2-24

SECTION 3 DSP56603EVM TECHNICAL SUMMARY 3-1
3.1 DSP56603EVM DESCRIPTION AND FEATURES 3-3
3.2 DSP56603 DESCRIPTION 3-3
3.3 MEMORY 3-4
3.4 AUDIO CODEC 3-6
3.4.1 Codec Analog Input/Output 3-7
3.4.2 Codec Digital Interface 3-8
3.4.3 Codec Clock 3-9
3.5 COMMAND CONVERTER 3-9

DSP56603EVM SCHEMATICS A-1

DSP56603EVM PARTS LIST B-1
B.1 PARTS LISTING B-3

MOTOROLA ASSEMBLER NOTES C-1
C.1 INTRODUCTION C-3
C.2 ASSEMBLER SIGNIFICANT CHARACTERS C-3
C.2.1 ; Comment Delimiter Character C-3
C.2.2 ;; Unreported Comment Delimiter Characters C-4
C.2.3 \ Line Continuation Character or Macro Argument Concatenation Character Line Continuation C-4
C.2.3.1 Line Continuation C-4
C.2.3.2 Macro Argument Concatenation C-4
C.2.4 ? Return Value of Symbol Character C-5
C.2.5 % Return Hex Value of Symbol Character C-6

C.2.6 ^ Macro Local Label Override C-6
C.2.7 " Macro String Delimiter or Quoted String DEFINE Expansion Character C-7
C.2.7.1 Macro String C-7
C.2.7.2 Quoted String DEFINE Expansion C-8
C.2.8 @ Function Delimiter C-8
C.2.9 * Location Counter Substitution C-8
C.2.10 ++ String Concatenation Operator C-9
C.2.11 [ ] Substring Delimiter [<string>,<offset><length>] C-9
C.2.12 << I/O Short Addressing Mode Force Operator C-9
C.2.13 < Short Addressing Mode Force Operator C-10
C.2.14 > Long Addressing Mode Force Operator C-10
C.2.15 # Immediate Addressing Mode C-11
C.2.16 #< Immediate Short Addressing Mode Force Operator . . . C-11
C.2.17 #> Immediate Long Addressing Mode Force Operator . . . C-12
C.3 ASSEMBLER DIRECTIVES C-13
C.3.1 BADDR Set Buffer Address C-13
C.3.2 BSB Block Storage Bit-Reverse C-13
C.3.3 BSC Block Storage of Constant C-14
C.3.4 BSM Block Storage Modulo C-15
C.3.5 BUFFER Start Buffer C-15
C.3.6 COBJ Comment Object File C-16
C.3.7 COMMENT Start Comment Lines C-17
C.3.8 DC Define Constant C-17
C.3.9 DCB Define Constant Byte C-18
C.3.10 DEFINE Define Substitution String C-19
C.3.11 DS Define Storage C-20
C.3.12 DSM Define Modulo Storage C-20
C.3.13 DSR Define Reverse Carry Storage C-21
C.3.14 DUP Duplicate Sequence of Source Lines C-22
C.3.15 DUPA Duplicate Sequence With Arguments C-23
C.3.16 DUPC Duplicate Sequence With Characters C-24
C.3.17 DUPF Duplicate Sequence In Loop C-25
C.3.18 END End of Source Program C-26
C.3.19 ENDBUF End Buffer C-27
C.3.20 ENDIF End of Conditional Assembly C-27

C.3.21 ENDM End of Macro Definition C-28
C.3.22 ENDSEC End Section C-28
C.3.23 EQU Equate Symbol to a Value C-29
C.3.24 EXITM Exit Macro C-29
C.3.25 FAIL Programmer Generated Error C-30
C.3.26 FORCE Set Operand Forcing Mode C-30
C.3.27 GLOBAL Global Section Symbol Declaration C-31
C.3.28 GSET Set Global Symbol to a Value C-31
C.3.29 HIMEM Set High Memory Bounds C-32
C.3.30 IDENT Object Code Identification Record C-32
C.3.31 IF Conditional Assembly Directive C-33
C.3.32 INCLUDE Include Secondary File C-34
C.3.33 LIST List the Assembly C-35
C.3.34 LOCAL Local Section Symbol Declaration C-35
C.3.35 LOMEM Set Low Memory Bounds C-36
C.3.36 LSTCOL Set Listing Field Widths C-36
C.3.37 MACLIB Macro Library C-37
C.3.38 MACRO Macro Definition C-38
C.3.39 MODE Change Relocation Mode C-39
C.3.40 MSG Programmer Generated Message C-39
C.3.41 NOLIST Stop Assembly Listing C-40
C.3.42 OPT Assembler Options C-41
C.3.42.1 Listing Format Control C-41
C.3.42.2 Reporting Options C-41
C.3.42.3 Message Control C-42
C.3.42.4 Symbol Options C-42
C.3.42.5 Assembler Operation C-43
C.3.43 ORG Initialize Memory Space and Location Counters . . . C-48
C.3.44 PAGE Top of Page/Size Page C-51
C.3.45 PMACRO Purge Macro Definition C-52
C.3.46 PRCTL Send Control String to Printer C-53
C.3.47 RADIX Change Input Radix for Constants C-53
C.3.48 RDIRECT Remove Directive or Mnemonic from Table . . . C-54
C.3.49 SCSJMP Set Structured Control Statement Branching Mode C-55

C.3.50 SCSREG Reassign Structured Control Statement Registers C-55
C.3.51 SECTION Start Section C-56
C.3.52 SET Set Symbol to a Value C-58
C.3.53 STITLE Initialize Program Sub-Title C-59
C.3.54 SYMOBJ Write Symbol Information to Object File C-59
C.3.55 TABS Set Listing Tab Stops C-59
C.3.56 TITLE Initialize Program Title C-60
C.3.57 UNDEF Undefine DEFINE Symbol C-60
C.3.58 WARN Programmer Generated Warning C-61
C.3.59 XDEF External Section Symbol Definition C-61
C.3.60 XREF External Section Symbol Reference C-62
C.4 STRUCTURED CONTROL STATEMENTS C-62
C.4.1 Structured Control Directives C-62
C.4.2 Syntax C-63
C.4.2.1 .BREAK Statement C-64
C.4.2.2 .CONTINUE Statement C-65
C.4.2.3 .FOR Statement C-65
C.4.2.4 .IF Statement C-66
C.4.2.5 .LOOP Statement C-67
C.4.2.6 .REPEAT Statement C-67
C.4.2.7 .WHILE Statement C-68
C.4.3 Simple and Compound Expressions C-68
C.4.3.1 Simple Expressions C-68
C.4.3.2 Condition Code Expressions C-69
C.4.3.3 Operand Comparison Expressions C-69
C.4.3.4 Compound Expressions C-70
C.4.3.5 Statement Formatting C-70
C.4.3.6 Expression Formatting C-71
C.4.3.7 .FOR/.LOOP Formatting C-71
C.4.4 Assembly Listing Format C-71
C.4.5 Effects on the Programmer’s Environment C-72

CODEC PROGRAMMING TUTORIAL D-1
D.1 INTRODUCTION D-3
D.2 PROGRAMMING THE CODEC D-3

D.3 ECHO.ASM PROGRAM DESCRIPTION D-3
D.3.1 Source Code Description D-4
D.3.1.1 Included Files D-5
D.3.1.2 Constant Definitions D-6
D.3.1.3 Interrupt Buffers D-6
D.3.1.4 Sample Program D-7

LIST OF FIGURES

Figure 1-1 DSP56603EVM Component Layout 1-6

Figure 1-2 Connecting the DSP56603EVM Cables 1-7

Figure 1-3 DSP56603EVM Test Sample Output—Fail 1-12

Figure 1-4 DSP56603EVM Test Sample Output—Pass 1-13

Figure 2-1 Development Process Flow 2-3

Figure 2-2 Example Debugger Display Window 2-24

Figure 3-1 DSP56603EVM Functional Block Diagram 3-4

Figure 3-2 DSP56603EVM Component Layout 3-5

Figure 3-3 SRAM Connections to the DSP56603 3-5

Figure 3-4 Configuration for J9 3-6

Figure 3-5 Codec Analog Input/Output Diagram 3-7

Figure 3-6 Codec Digital Interface Connections 3-8

Figure 3-7 Codec Clock Generation Diagram 3-9

Figure 3-8 RS-232 Serial Interface 3-10

LIST OF TABLES

Table 2-1 Link Time Options 2-21

Table B-1 DSP56603EVM Parts List B-3

LIST OF EXAMPLES

Example 2-1 Sample Source Statement. 2-4

Example 2-2 Simple DSP56603 Code Example. 2-6

Example C-1 Example of Comment Delimiter. C-3

Example C-2 Example of Unreported Comment Delimiter C-4

Example C-3 Example of Line Continuation Character. C-4

Example C-4 Example of Macro Concatenation C-5

Example C-5 Example of Use of Return Value Character C-5

Example C-6 Example of Return Hex Value Symbol Character C-6

Example C-7 Example of Local Label Override Character C-7

Example C-8 Example of a Macro String Delimiter Character C-7

Example C-9 Example of a Quoted String DEFINE Expression C-8

Example C-10 Example of a Function Delimiter Character. C-8

Example C-11 Example of a Location Counter Substitution C-8

Example C-12 Example of a String Concatenation Operator C-9

Example C-13 Example of a Substring Delimiter C-9

Example C-14 Example of an I/O Short Addressing Mode Force Operator C-9

Example C-15 Example of a Short Addressing Mode Force Operator C-10

Example C-16 Example of a Long Addressing Mode Force Operator C-11

Example C-17 Example of Immediate Addressing Mode C-11

Example C-18 Example of Immediate Short Addressing Mode Force Operator. . . C-11

Example C-19 Example of an Immediate Long Addressing Mode Operator C-12
Example C-20 Example BADDR Directive. C-13
Example C-21 BSB Directive C-14
Example C-22 Block Storage of Constant Directive. C-14
Example C-23 Block Storage Modulo Directive C-15
Example C-24 BUFFER Directive C-16
Example C-25 COBM Directive C-16
Example C-26 COMMENT Directive C-17
Example C-27 Single Character String Definition C-18
Example C-28 Multiple Character String Definition C-18
Example C-29 DC Directive C-18
Example C-30 DCB Directive. C-19
Example C-31 DEFINE Directive C-20
Example C-32 DS Directive C-20
Example C-33 DSM Directive C-21
Example C-34 DSR Directive. C-21
Example C-35 DUP Directive. C-22
Example C-36 DUPA Directive C-23
Example C-37 DUPC Directive C-24
Example C-38 DUPF Directive. C-25
Example C-39 END Directive. C-27
Example C-40 ENDBUF Directive C-27

Example C-41 ENDIF Directive C-27
Example C-42 ENDM Directive C-28
Example C-43 EQU Directive C-29
Example C-44 EXITM Directive C-30
Example C-45 FAIL Directive C-30
Example C-46 FORCE Directive C-31
Example C-47 GLOBAL Directive C-31
Example C-48 GSET Directive C-32
Example C-49 HIMEM Directive C-32
Example C-50 IDENT Directive C-33
Example C-51 IF Directive C-34
Example C-52 INCLUDE Directive C-34
Example C-53 LIST Directive C-35
Example C-54 LOCAL Directive C-36
Example C-55 LOMEM Directive C-36
Example C-56 LSTCOL Directive C-37
Example C-57 MACLIB Directive C-38
Example C-58 MACRO Directive C-39
Example C-59 MODE Directive C-39
Example C-60 MSG Directive C-40
Example C-61 NOLIST Directive C-40
Example C-62 OPT Directive C-48

Example C-63 ORG Directive C-50
Example C-64 PAGE Directive C-52
Example C-65 PMACRO Directive. C-52
Example C-66 PRCTL Directive. C-53
Example C-67 RADIX Directive C-54
Example C-68 RDIRECT Directive C-54
Example C-69 SCSJMP Directive C-55
Example C-70 SCSREG Directive C-56
Example C-71 SECTION Directive C-58
Example C-72 SET Directive C-58
Example C-73 STITLE Directive C-59
Example C-74 SYMOBJ. C-59
Example C-75 TABS Directive C-60
Example C-76 TITLE Directive. C-60
Example C-77 UNDEF Directive C-60
Example C-78 WARN Directive C-61
Example C-79 XDEF Directive. C-61
Example C-80 XREF Directive. C-62
Example C-81 .BREAK Statement. C-64
Example C-82 .CONTINUE Statement C-65
Example C-83 .FOR Statement C-66
Example C-84 .IF Statement C-66

Example C-85 .LOOP Statement C-67
Example C-86 .REPEAT Statement C-67
Example C-87 .WHILE Statement. C-68
Example C-88 Condition Code Expression. C-69
Example D-1 Program Description D-4
Example D-2 Included Files D-5
Example D-3 Constant Definitions D-6
Example D-4 Interrupt Buffers. D-6
Example D-5 Sample Program Listing D-7

SECTION 1

QUICK START GUIDE

1.1 OVERVIEW 1-3
1.2 EQUIPMENT 1-3
1.3 INSTALLATION PROCEDURE 1-5

1.1 OVERVIEW

This document supports the DSP56603 Evaluation Module (DSP56603EVM) 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 (this section) is a Quick Start Guide that provides a summary description of the evaluation module contents, additional requirements, and quick installation and test information.
- Section 2 provides a simple programming example.
- Section 3 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.

1.2 EQUIPMENT

The following section gives a brief summary of the equipment required to use the DSP56603EVM, some of which is included with the module, and some of which must be supplied by the user.

1.2.1 What You Get with the DSP56603EVM

The following material is provided with the DSP56603EVM:

- DSP56603 Evaluation Module board
- Supporting documentation, including *DSP56603EVM Kit Contents List (DSP56603EVMCL/D)* , which provides a comprehensive list of all kit contents

- CD-ROM containing Motorola DSP Technical Documentation, including the DSP56600 Family Manual, DSP56603 Technical Data sheet, and DSP56603 User's Manual
- CD-ROM containing the Motorola Development Tools software and on-line User's Manual
- 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 Windows95)
  - 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 plug to DB9 receptacle)
- 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 DSP56603EVM board
2. Connecting the board to the PC and power
3. Installing the software
4. Testing the installation

1.3.1 Preparing the DSP56603EVM

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, a jumper connects pins 2 and 3. These jumpers perform the following functions:

- J4 controls the interface between the DSP56603 JTAG / OnCE port and DSP56002 Synchronous Serial Interface (SSI).
- J7 controls the interface between the audio codec and the DSP56603 Synchronous Serial Interface (SSI0).

- J9 controls the interface between the external SRAM and the Memory Chip Select ($\overline{\text{MCS}}$) pin.

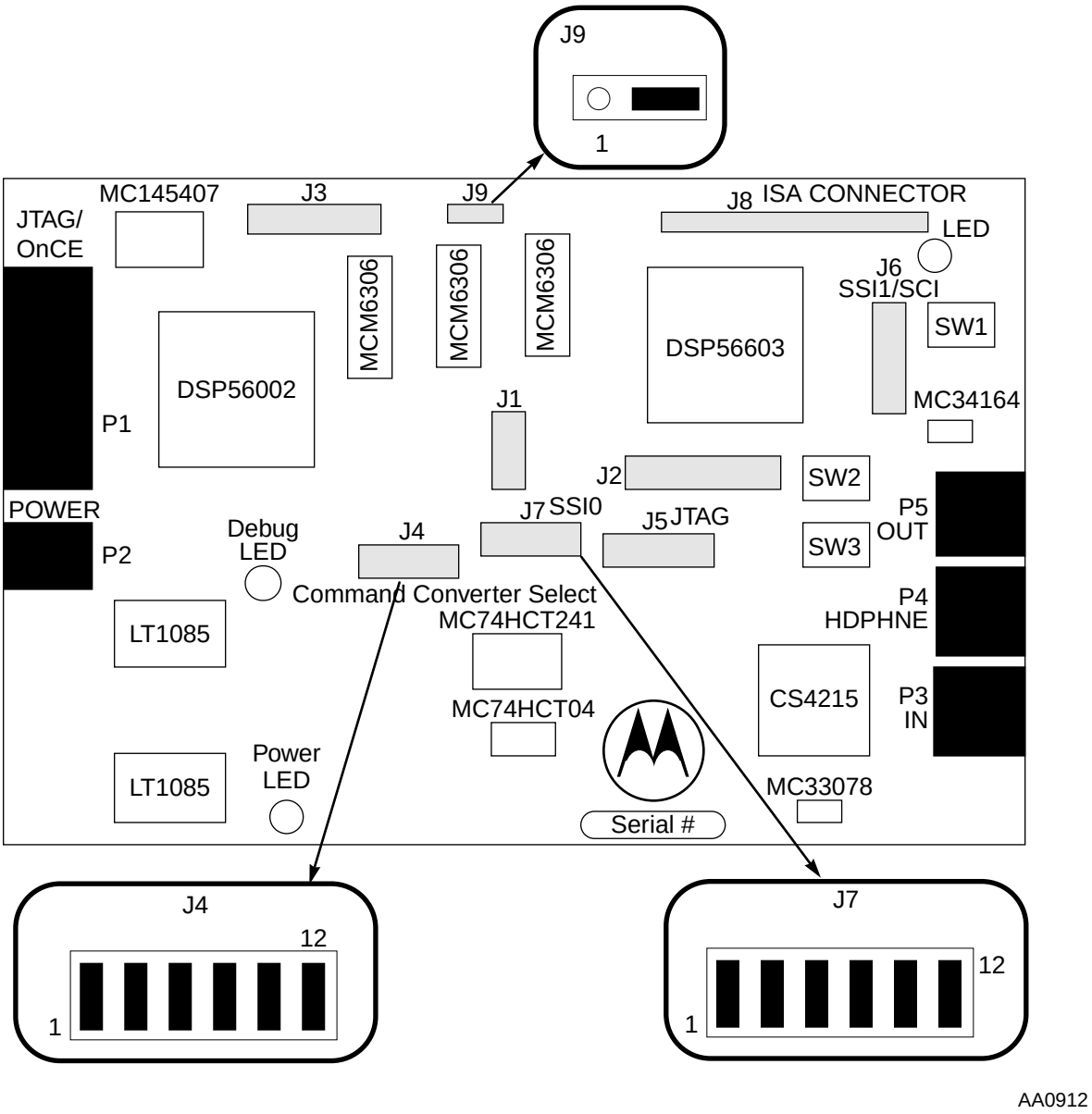


Figure 1-1 DSP56603EVM Component Layout

Ensure that the jumpers on J4, J7, and J9 are connected as shown in **Figure 1-1**. Next, install the four plastic standoffs in the holes in the corners of the DSP56603EVM board.

1.3.2 Connecting the DSP56603EVM to the PC and Power

Figure 1-2 shows the interconnection diagram for connecting the PC and the external power supply to the DSP56603EVM board. Use the following steps to complete cable connections:

CAUTION

Turn off all devices or disconnect power to them before attaching or disconnecting 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 DSP56603EVM 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 DSP56603EVM board.
5. Apply power to the power supply. The green Power LED lights when power is correctly applied.

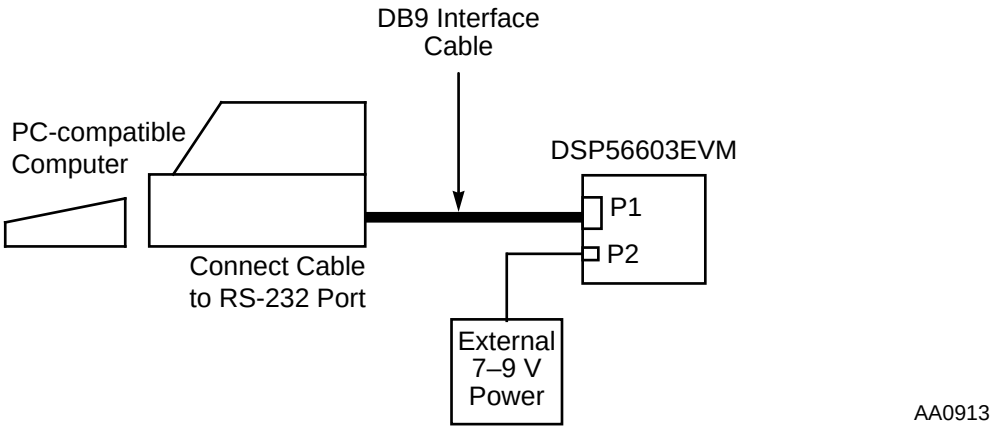


Figure 1-2 Connecting the DSP56603EVM Cables

1.3.3 Installing the Software

The DSP56603EVM software includes the following:

- Motorola diskette containing:
  - Assembler
  - Linker
  - Test 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. Enter ***a:install a:*** on the command line. The program installs the software in the default destination ***c: EVM5660x*** .

   If your diskette drive is designated other than a:, specify the appropriate source drive.
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.

   If your diskette drive is designated other than a:, specify the appropriate drive.

   The install program creates a program group called ***EVM5660x*** and a program icon called ***EVM5660x*** within Windows. This step completes the software installation.
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.

1.3.4 Testing the DSP56603EVM

The following sections describe the DSP56603EVM self-test and analysis programs. The self-test allows the user to determine if the DSP56603EVM is properly connected and ready for operation. The analysis program allows the user to examine the DSP56603EVM operation in more detail to diagnose problems that may occur.

1.3.4.1 DSP56603EVM Self-Test

This section describes how to run the DSP56603EVM self-test. The self-test contains two parts, as follows:

- The first part of the self-test examines the external memory of the DSP56603EVM by writing to, and reading from, the external SRAM. It also examines the analog circuitry of the DSP56603EVM by analyzing the response of the analog circuitry to a sequence of tones. This part of the self-test is run after start-up by pressing the switch labeled SW2 to invoke external Interrupt Request A ($\overline{\text{IRQA}}$). (Refer to **Figure 1-1** on page 1-6 for the location of SW2.) Do not connect the headphones for the first part of the self-test.
- The second part of the self-test analyzes the audio circuitry of the DSP56603EVM by moving an audio signal through the codec. This part of the self-test is run by pressing the switch labeled SW3 to invoke external Interrupt Request D ($\overline{\text{IRQD}}$). (Refer to **Figure 1-1** on page 1-6 for the location of SW3.) Connect the headphones for the second part of the self-test.

The following steps describe how to run the self-test code for the DSP56603EVM.

WARNING

Do not wear the headphones during the first part of the self-test.

1. Install the cable with the 1/8-inch stereo plugs between the line input jack labeled P3/IN on the DSP56603EVM and the line output jack labeled P5/OUT on the DSP56603EVM. Refer to **Figure 1-1** on page 1-6 for the location of the line input and output jacks.
2. Start Windows.
3. Invoke the Debugger by double-clicking on the icon labeled ***EVM5660x*** in the ***EVM5660x*** program group.

4. Click on the command window and enter ***force r*** to reset the DSP56603 and enter the Debug mode.
5. Enter ***load fltr_tst*** to load the self-test file into the DSP56603.
6. Enter ***go*** to begin 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 DSP56603EVM passed the test. If the DSP56603EVM fails the test, the LED flashes. To repeat the test, press SW2 to invoke $\overline{\text{IRQA}}$.
7. Now run the second part of the self-test code to test the analog circuitry of the DSP56603EVM. Using the cable with the 1/8-inch stereo plugs, connect the phone output of the audio source to the line input jack labeled P3/IN on the DSP56603EVM. Also connect a pair of headphones to the headphone jack labeled P4/HDPHNE on the DSP56603EVM. 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.

Note: The headphones can safely be connected to the DSP56603EVM while it is powered on.

8. Press the switch labeled SW3 on the DSP56603EVM to invoke $\overline{\text{IRQD}}$. 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 step 1 and press SW2. To repeat the second part of the test, perform step 7 and press SW3. You can switch between the parts at any time.

WARNING

Do not wear the headphones during the first part of the self-test.

If both parts of the self-test complete correctly, the DSP56603EVM 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 DSP56603EVM continues to fail either test, run the DSP56603EVM analysis program described in the following section to try to determine the cause of the problem.

1.3.4.2 DSP56603EVM Analysis Program

This section describes how to run the DSP56603EVM 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 DSP56603EVM is failing the tests.

The following steps describe how to run the analysis program for the DSP56603EVM.

WARNING

Do not wear the headphones during this test.

1. Run the program ***tst603.exe***. This can be done from the Microsoft Windows Program Manager by pulling down the File menu, choosing Run, entering ***c:\EVM5660x\tst603.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 labeled P3/IN on the DSP56603EVM and the line output jack labeled P5/OUT on the DSP56603EVM. 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 goes 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 DSP56603EVM serial number. The serial number can be found at the bottom of the DSP56603EVM 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 outputs a set of diagnostics to the screen, similar to those shown in **Figure 1-3** and **Figure 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 DSP56603 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, labeled dB below MAX.

-- DSP56603EVM Performance Analysis Ver.1.00 --
EVM Serial No.: 1234 Mon Nov 04 17:45:47 1996
DSP56603 Chip Revision No. 001603_

	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	-15.00		
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 ***

AA0914

Figure 1-3 DSP56603EVM Test Sample Output—Fail

-- DSP56603EVM Performance Analysis Ver.1.00 --
EVM Serial No.: 1234 Mon Nov 04 17:45:47 1996
DSP56603 Chip Revision No. 001603

	LEFT	RIGHT	——-——--dB below MAX——-----		
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	-15.00
Pass: 6	Address:	X:0000	Y:0000		
	Expected:	000000	000000		
	Received:	000000	000000		

-------- PASS --------

AA0915

Figure 1-4 DSP56603EVM 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 DSP56603EVM 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 DSP56603EVM to pass the analog circuitry test. If either channel's dB levels do not satisfy the acceptable responses, an asterisk is displayed 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 DSP56603EVM passes the analog circuitry test, no asterisks are shown 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 DSP56603EVM passed the test. If all the values in the Address, Expected, and Received fields are zero, as in **Figure 1-4**, the DSP56603EVM passed the external memory test. If the DSP56603EVM fails the external memory test, these fields 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 DSP56603EVM passed or failed the test. If the last line says PASS, as in **Figure 1-4**, then the DSP56603EVM 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 DSP56603EVM continues to fail the test, email the DSP help line at dsphelp@dsp.sps.mot.com.

Now the test is complete and the DSP56603EVM is ready for operation.

SECTION 2

EXAMPLE TEST PROGRAM

2.1 OVERVIEW 2-3
2.2 WRITING THE PROGRAM 2-4
2.3 ASSEMBLING THE PROGRAM 2-8
2.4 MOTOROLA DSP LINKER 2-16
2.5 INTRODUCTION TO THE DEBUGGER SOFTWARE 2-23
2.6 RUNNING THE PROGRAM 2-24

2.1 OVERVIEW

This section contains an example that illustrates how to develop a very simple program for the DSP56603. 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 2-1 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.

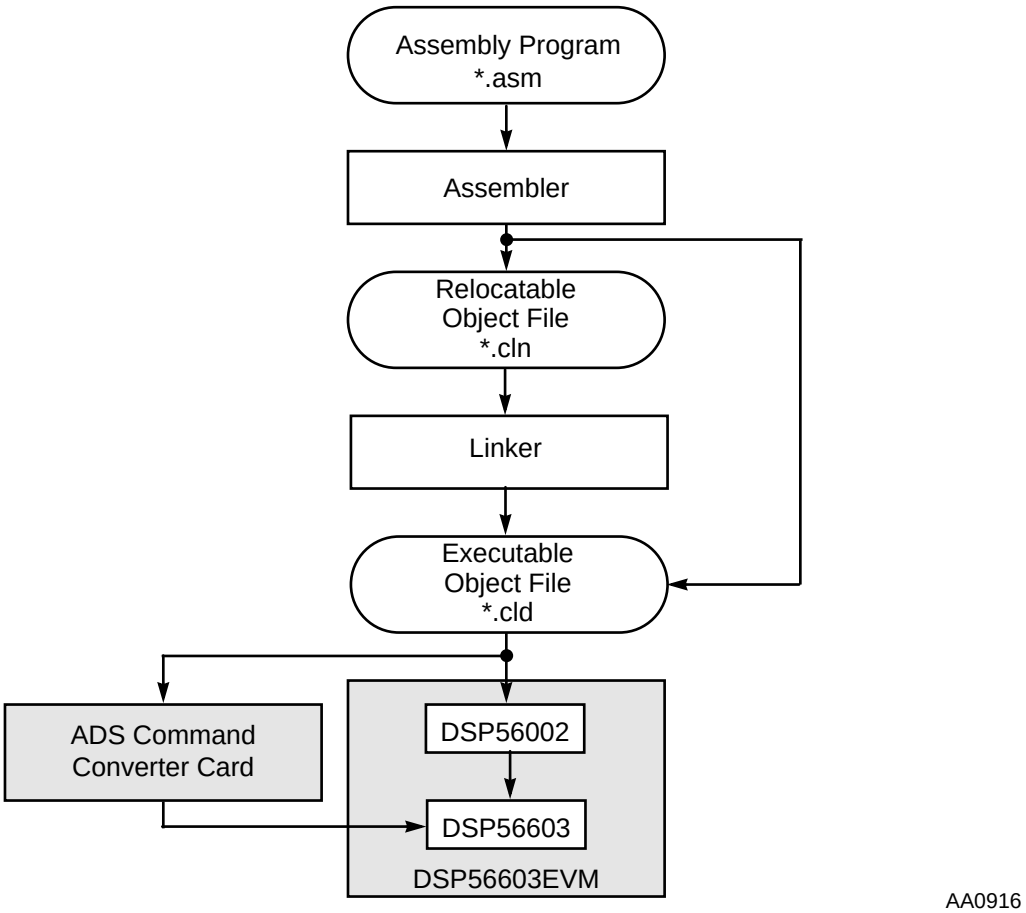


Figure 2-1 Development Process Flow

The following sections give basic information regarding the assembly program, the Assembler, the Linker, and the object files. Detailed information about these subjects can

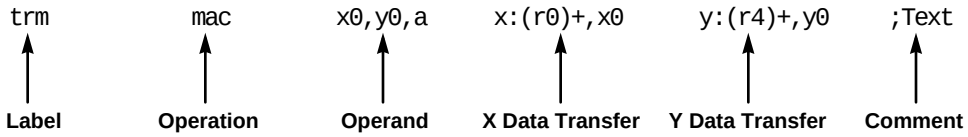
be found in the Assembler and Linker manuals provided with the Motorola DSP CLAS software package available through your Motorola sales office or distributor. The documentation is also available through the Motorola DSP internet URL *http://www.motorola-dsp.com.*

2.2 WRITING THE PROGRAM

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

2.2.1 Source Statement Format

Programs written in assembly language consist of a sequence of source statements. Each source statement can include as many as 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. **Example 2-1** shows all six possible fields.



Example 2-1 Sample Source Statement

2.2.1.1 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.

2.2.1.2 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 can be one of three types:

- **Opcode**—mnemonics that correspond directly to DSP machine instructions
- **Directive**—special operation codes known to the Assembler that control the assembly process
- **Macro call—**invocation of a previously defined macro that is to be inserted in place of the macro call

2.2.1.3 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.

2.2.1.4 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 *DSP56600 Family Manual (DSP56600FM/AD)* for a complete discussion of addressing modes that are applicable to data transfer specifications.

2.2.1.5 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.

2.2.2 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 DSP56603 has a special instruction (MAC) that multiplies two values and adds the result to the contents of an accumulator. This program is provided as example.asm on the DSP56603EVM diskette and is placed in the **EVM5660x** directory by the installation procedure.

Example 2-2 Simple DSP56603 Code Example

```
;************************************************************
;A SIMPLE PROGRAM: CALCULATING THE SUM OF PRODUCTS
;************************************************************
    nolist
    include 'ioequ.asm'
    list
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      $4756,$7383,$9267,$8989,$0912,$f250
        dc      $9871,$3A87,$9872,$34b8,$7346,$2337
        dc      $f767,$4234,$3247,$f400
;************************************************************
;Y MEMORY
;************************************************************
        org     y:YBASE                 ;instructs the assembler that we
                                        ;are referring to Y memory starting
                                        ;at location YBASE
list2   dc      $f987,$8000,$fedc,$4873,$9575,$3698
        dc      $2479,$8a34,$7345,$3447,$9384,$304f
        dc      $1234,$6577,$5671,$6756
```

Example 2-2 Simple DSP56603 Code Example (Continued)

```
;*********************************************************
;PROGRAM
;*********************************************************
        org     p:0                     ;put following program in program
                                        ;memory starting at location 0
        jmp     START                   ;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

START
        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
        movep   #$0003,x:M_PCTL0        ;set up PLL for MF of 4
        movep   #$0004,x:M_PCTL1        ;set up PLL for MF of 4
        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
        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
;*********************************************************
```

2.3 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.

2.3.1 Assembler Command Format

The Motorola DSP Assembler is included with the DSP56603EVM 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 DSP56603. The general format of the command line to invoke the Assembler is:

asm56600 [options] <filenames>

where ***asm56600*** 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.

2.3.2 Assembler Options

-A

The **-A** 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>

The **-B** 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 does not generate an object file. If no <objfil> is specified, the Assembler uses 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: ***asm56600 -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>

The **-D** option replaces all occurrences of <symbol> with <string> in the source files to be assembled.

Example: ***asm56600 -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 is overwritten.

Example: ***asm56600 -EWerrors prog.asm***

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

–F<argfil>

The **–F** 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: ***asm56600 -Fopts.cmd***

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

-G

The **-G** 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: ***asm56600 -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: ***asm56600 -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>

The **-L** 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 uses 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: ***asm56600 -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 is named using the basename of the first source file, in this case filter, and the listing file is called filter.lst.

-M<pathname>

The **-M** 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: ***asm56600 -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

The **-V** option causes the Assembler to report assembly progress to the standard error output stream.

-Z

The **-Z** 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: ***asm56600 -A -B -L -G filename.asm***

2.3.3 Assembler Directives

In addition to the DSP56603 instruction set, assembly programs can contain mnemonic directives that specify auxiliary actions to be performed by the Assembler. These are the Assembler directives. These directives are not always translated into machine language. This section describes these Assembler directives.

2.3.3.1 Assembler Significant Characters

Assembler significant characters are one- and two-character sequences that are significant to the Assembler. They include the following:

; —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

2.3.3.2 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

2.3.3.3 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

2.3.3.4 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

2.3.3.5 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

2.3.3.6 Object File Control Directives

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

2.3.3.7 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

2.3.3.8 Structured Programming Directives

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

2.3.4 Assembling the Example Program

The Assembler is a MS-DOS based program, thus to use the Assembler you need to exit Windows or open a MS-DOS Prompt Window. To assemble the example program, type ***asm56600 -a -b -l -g example.asm*** in the ***EVM5660x*** directory created by the installation process. This creates two additional files: example.cld and example.lst. The example.cld file is the absolute object file of the program, and this is what is downloaded into the

DSP56603. The example.lst file is the listing file and gives full details of where the program and data is placed in the DSP56603 memory.

2.4 MOTOROLA DSP LINKER

Though not needed for our simple example, the Motorola DSP Linker is also included with the DSP56603EVM. The Motorola DSP Linker is a program that processes relocatable object files produced by the Motorola DSP Assembler, generating an absolute executable file that can be downloaded to the DSP56603. 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.

2.4.1 Linker Options

-A

The **-A** 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>

The **-B** 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 uses 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 is 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 is overwritten.

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

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

-F<argfil>

The **-F** 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 that 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>

The **-M** 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 uses 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 does not generate a map file. The **-M** option should be specified only once. If the file named in the **-M** option already exists, it is 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 is named using the basename of the first input file, in this case filter. The map file is 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 can 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 can be I for Internal memory, E for External memory, R for ROM, A for Port A, or 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 can 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 can 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>

The **-R** 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 is made to open the file in the current directory. If no filename is specified, the Linker uses 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 does 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>

The **-U** 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 can be resolved by code within the library proj.lib.

-V

The **-V** 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. These link time options are described in **Table 2-1**. All options can be preceded by NO to reverse their meaning. The **-X<opt>** sequence can be repeated for as many options as desired.

Table 2-1 Link Time Options

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
Note: * Default setting	

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 aborts on warnings as well as errors.

-Z

The **-Z** 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 contains no symbol or line number information.

2.4.2 Linker Directives

Similar to the Assembler directives, the Linker includes mnemonic directives that specify auxiliary actions to be performed by the Linker. The Linker directives are:

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

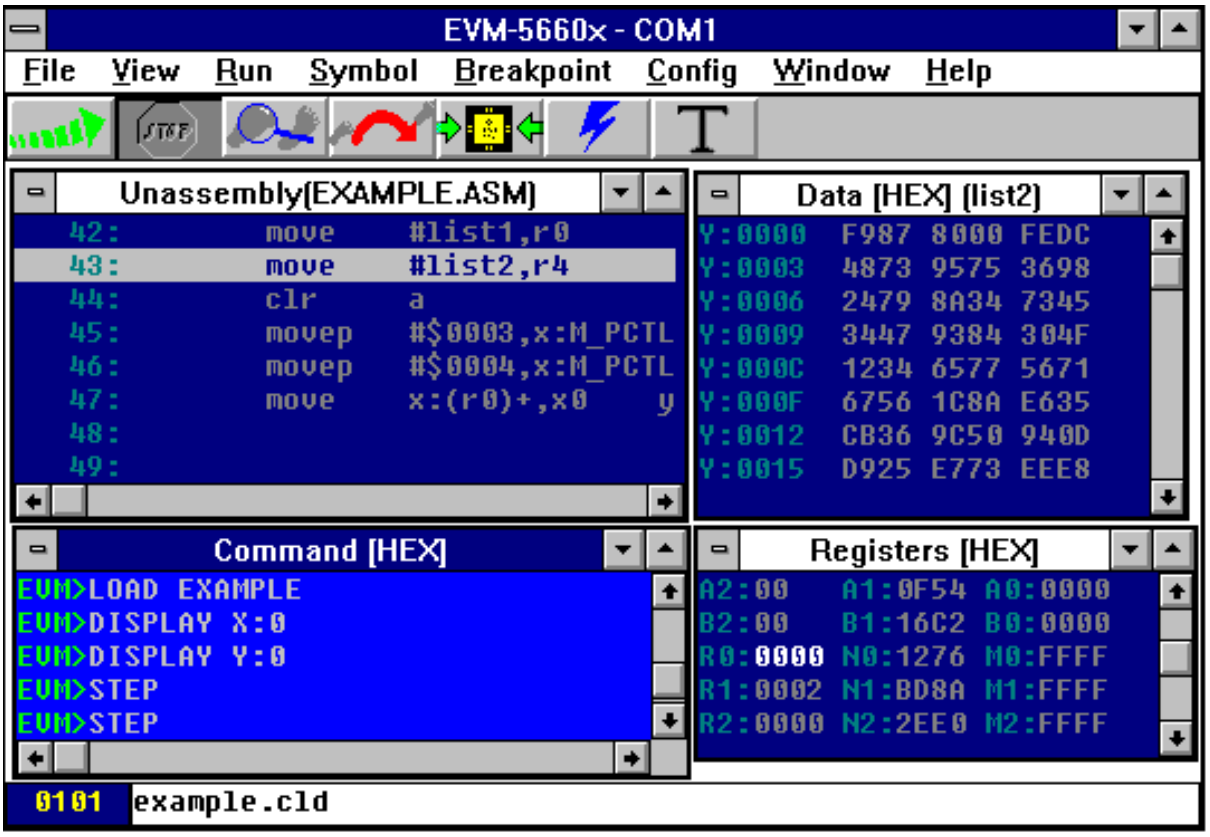
SYMBOL —Set symbol value

2.5 INTRODUCTION TO THE DEBUGGER SOFTWARE

This section provides 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 DSP56603. The Domain Technologies Debugger is included with the DSP56603EVM on the Domain Technologies 3-1/2 inch diskette and can be installed by following the instructions in **Installing the Software** on page 1-8. To invoke the Debugger, double-click on the icon labelled ***EVM5660x*** in the ***EVM5660x*** program group that was created when the Debugger was installed.

As shown in **Figure 2-2**, the Debugger display provides a 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 are placed in the command window. As shown in the self-test from **Testing the DSP56603EVM** on page 1-9, the command window is where commands are entered. The data window is used to display DSP56603 data. The unassembly window is used to display the DSP56603 programs. The next instruction to be executed is highlighted. The registers window shows the contents of the DSP56603 internal registers.

When the command window is selected, the tool-bar at the top of the screen contains buttons for the most often used commands. Other buttons are displayed when other windows are selected, and the function of these buttons can be found in the Debug-56K Manual.



AA0917

Figure 2-2 Example Debugger Display Window

2.6 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 36 is highlighted in the unassembly window, as this is the first instruction to be executed. However, before executing the program, check that the values expected to be in data memory are actually there. To do this, type ***display x:0*** and ***display y:0***. The data is displayed in the data window.

To step through the program, enter ***step*** in the command window prompt. As a shortcut, click on the step button or type the start of the command and press the space bar, and the debugger completes the remainder of the command. To repeat the last command, press return. As you step through the code, the registers in the registers window are changed by the instructions. After each cycle, any register that has been changed is 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 9F20 0274.

Stepping through a program like this is helpful for short programs, but impractical for large complex programs. The way to debug large programs is to set breakpoints, which are user-defined points at which execution of the code is halted, 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, enter ***break p:$107*** in the command window. The line before the loop brightens in the unassembly window, indicating the breakpoint has been set. To point the DSP56603 back to the start point of the program, type ***change pc 0***. This sets the program counter to point to the reset vector. To start the program, enter ***go*** or click on the Go button. The DSP56603 stops when it reaches the breakpoint, and you can step through the remainder of the code.

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

SECTION 3

DSP56603EVM TECHNICAL SUMMARY

3.1 DSP56603EVM DESCRIPTION AND FEATURES 3-3
3.2 DSP56603 DESCRIPTION 3-3
3.3 MEMORY. 3-4
3.4 AUDIO CODEC 3-6
3.5 COMMAND CONVERTER 3-9

3.1 DSP56603EVM DESCRIPTION AND FEATURES

An overview description of the DSP56603EVM is provided in the *DSP56603EVM Product Information (DSP56603EVMP/D)* included with this kit. The main features of the DSP56603EVM include the following:

- DSP56603 16-bit Digital Signal Processor
- Memory
- 16-bit CD-quality audio codec
- Command Converter

3.2 DSP56603 DESCRIPTION

A full description of the DSP56603, including functionality and user information is provided in the following documents included as a part of this kit:

- *DSP56603 Technical Data (DSP56603/D):* Provides features list and specifications including signal descriptions, DC power requirements, AC timing requirements, and available packaging
- *DSP56603 User's Manual (DSP56603UM/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
- *DSP56600 Family Manual (DSP56600FM/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 *DSP56603 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*.

Figure 3-1 shows a functional block diagram of the DSP56603EVM.

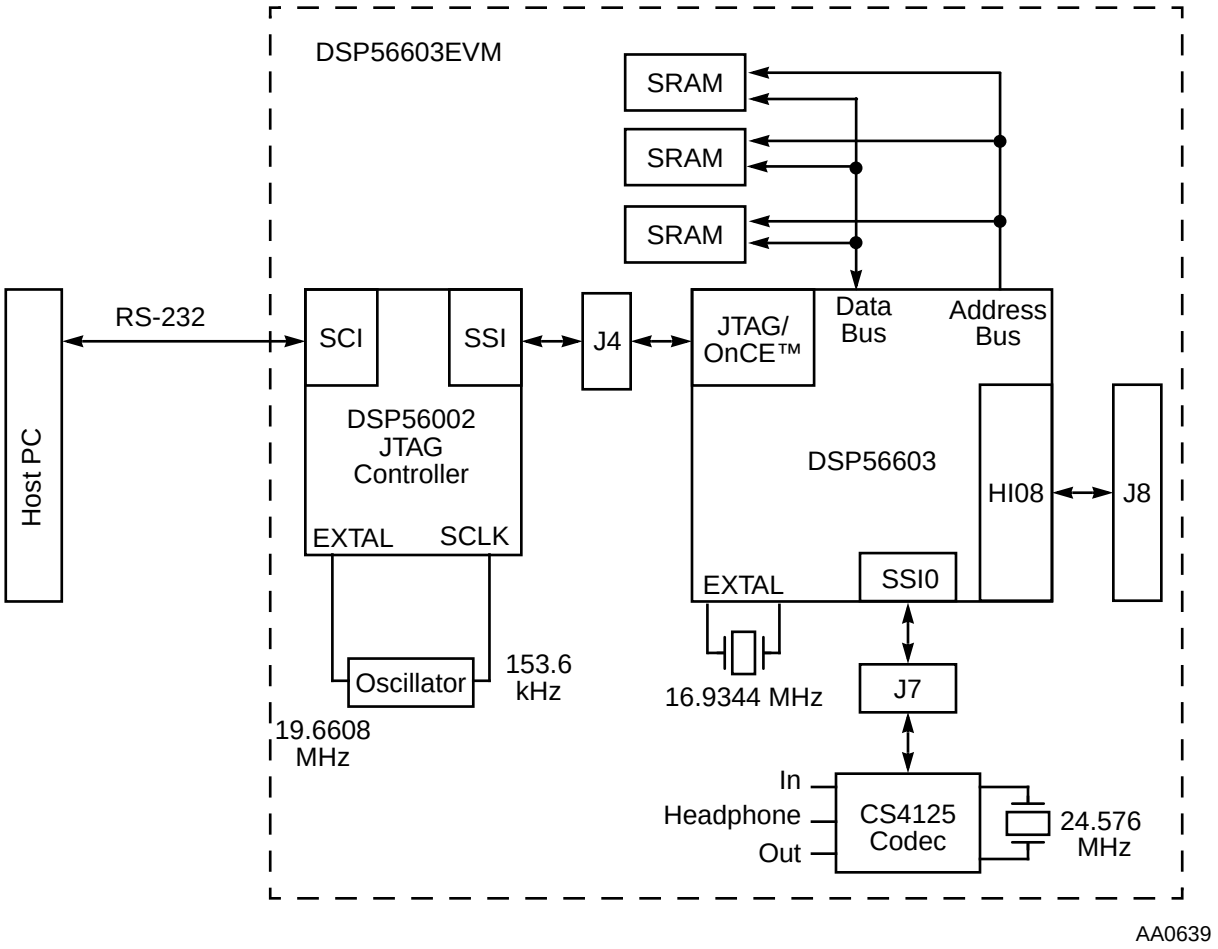
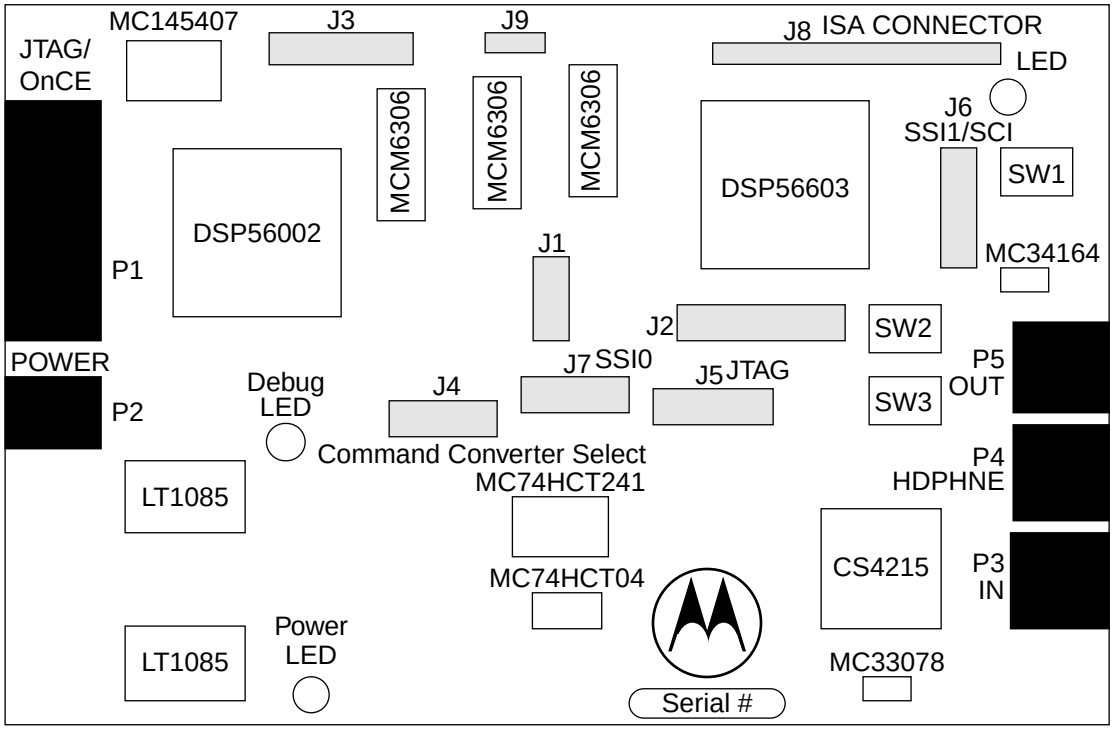


Figure 3-1 DSP56603EVM Functional Block Diagram

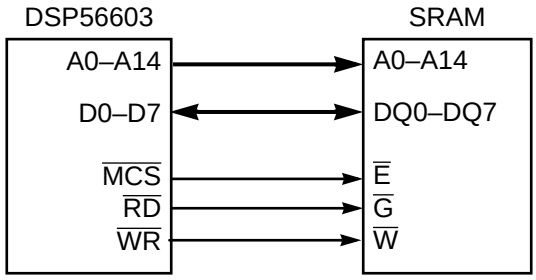
3.3 MEMORY

The DSP56603EVM 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. Refer to **Figure 3-2** for the location of the SRAM on the DSP56603EVM. The basic connection for the SRAM is shown in **Figure 3-3**.



AA0918

Figure 3-2 DSP56603EVM Component Layout



AA0919

Figure 3-3 SRAM Connections to the DSP56603

The data input/output pins DQ0–DQ7 for the first bank of SRAM are connected to the DSP56603 D0–D7 pins. Similarly, pins D8–D15 and D16–D23 of the DSP56603 are connected to the second and third banks of SRAM. The SRAM Write ($\overline{W}$) and Output ($\overline{G}$) enable lines are connected to the DSP56603 Write ($\overline{WR}$) and Read ($\overline{RD}$) lines, respectively. The SRAM chip Enable ($\overline{E}$) is generated by the DSP56603 Memory Chip Select ($\overline{MCS}$) pin.

The SRAM Address input pins A0–A13 are directly connected to the respective Port A address pins of the DSP. The SRAM address pin A14 is connected to the A14 Port A

address pin of the DSP56603 through the jumper on J9. For the DSP56603EVM, the jumper on J9 *must* connect pins 2 and 3, as shown in **Figure 3-4**, so that the SRAM address line A14 is connected to the DSP56603 address line A14. Alternate configurations for J9 will not work with the DSP56603EVM.

Figure 3-4 Configuration for J9

3.4 AUDIO CODEC

The DSP56603EVM 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 3-2** on page 3-5 for the location of the codec on the DSP56603EVM and to **Figure 3-1** on page 3-4 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 DSP56603. 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 SSI 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 DSP56603 SSI0 through the shorting jumpers on J7 shown in **Figure 3-2** on page 3-5. By removing these jumpers, the user has full access to the SSI0 pins of the DSP56603. The following sections describe the connections for the analog and digital sections of the codec.

3.4.1 Codec Analog Input/Output

The DSP56603EVM contains 1/8-inch stereo jacks for stereo input, output, and headphones. **Figure 3-5** shows the analog circuitry of the codec.

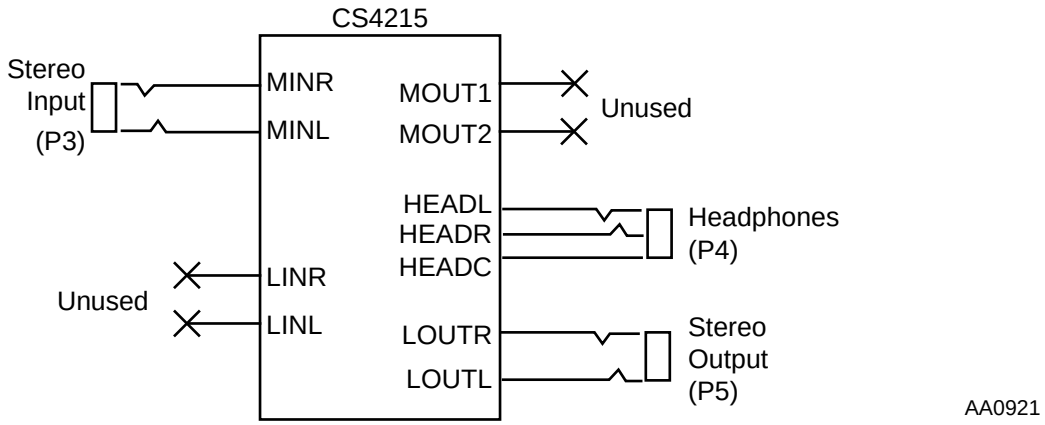


Figure 3-5 Codec Analog Input/Output Diagram

The stereo jack labelled P3/IN on the DSP56603EVM 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 can be disabled using the control mode of the codec.

The analog outputs of the codec are routed through 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 DSP56603EVM, which permits direct connection of stereo headphones to the DSP56603EVM. 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 DSP56603EVM. 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.

3.4.2 Codec Digital Interface

Figure 3-6 shows the digital interface to the codec.

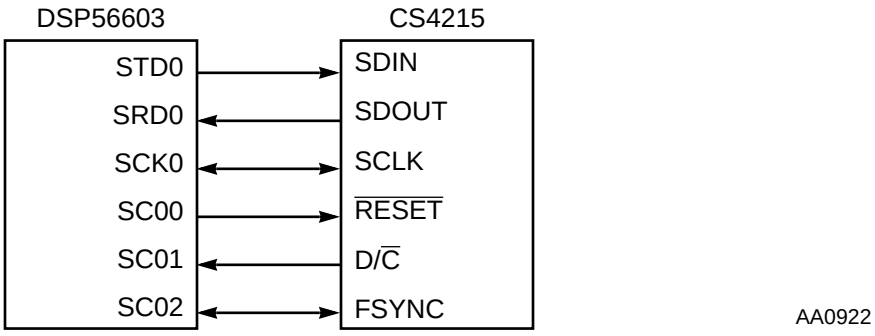


Figure 3-6 Codec Digital Interface Connections

The serial interface of the codec transfers digital audio data and control data into and out of the device. The codec communicates with the DSP56603 through the SSI0, which consists of independent transmitter and receiver sections.

On the DSP56603 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 DSP56603 bidirectional Serial Clock (SCK0) pin provides the serial bit rate clock for the SSI0 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 DSP56603 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 can be an input to the codec, or it can be an output from the codec.

3.4.3 Codec Clock

Figure 3-7 shows the clock generation diagram.

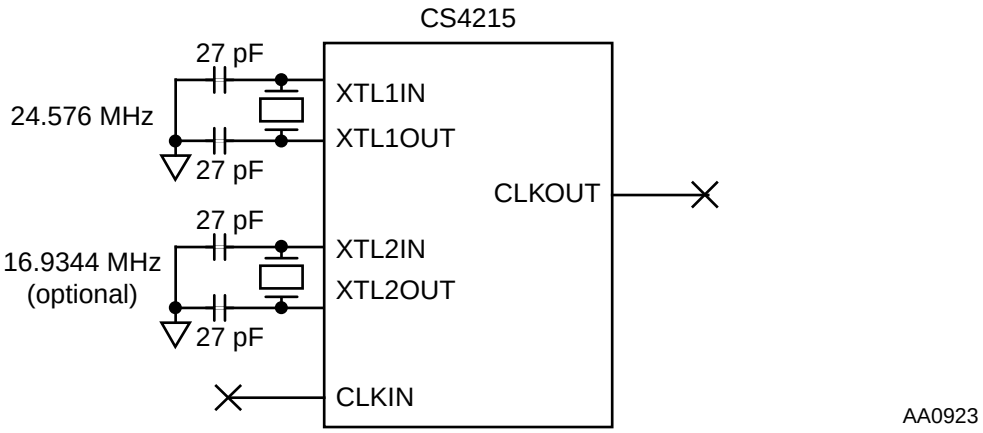


Figure 3-7 Codec Clock Generation Diagram

Two external crystals can 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 can be used. Refer to the CS4215 data sheet for information regarding selection of the correct clock source and divide ratios.

The codec on the DSP56603EVM 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.

3.5 COMMAND CONVERTER

The DSP56603EVM 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 can include read data, write data, reset OnCE, reset DSP56603 (the HA2 pin of the DSP56002 is then used to reset the DSP56603), 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 DSP56603 OnCE/JTAG port. The DSP56603 can then continue to receive data or transmit data back to the DSP56002. The DSP56002 sends a reply to the host PC to give status information.

The set of replies can include acknowledge good, acknowledge bad, in Debug mode, out of Debug mode, or data read. When the DP56603 is in the Debug state, the red Debug LED (D6) is lit.

The DSP56002 is connected to the DSP56603 OnCE port through the shorting jumpers on J4. By removing the jumpers, the user has full access to the OnCE/JTAG pins of the DSP56603. Refer to **Figure 3-2** on page 3-5 for the location of J4 on the DSP56603EVM and to **Figure 3-1** on page 3-4 for a functional diagram. **Figure 3-8** shows the RS-232 serial interface diagram.

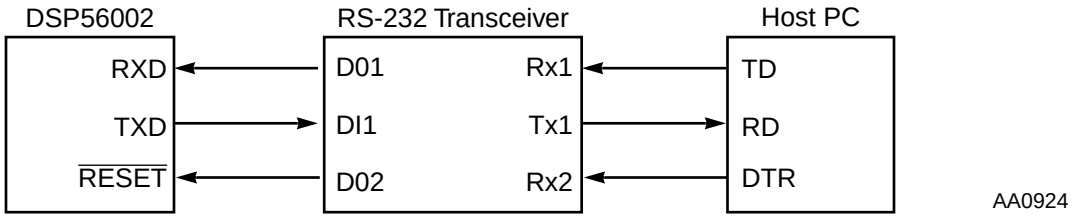


Figure 3-8 RS-232 Serial Interface

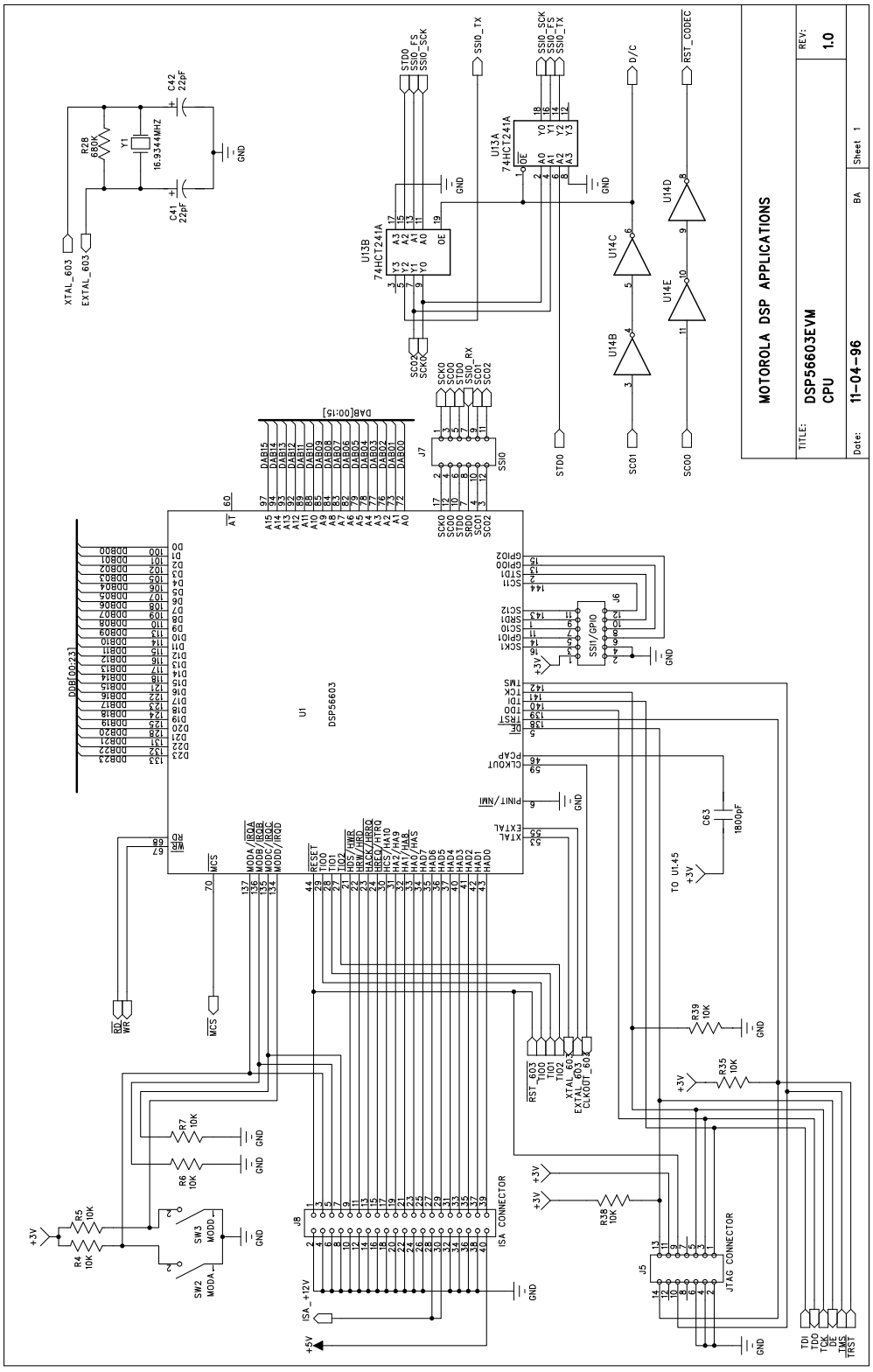
Motorola's 5 Volt-Only Driver/Receiver MC145407 is used to transmit the signals between the host PC and the DSP56002. Serial data is transmitted from the host PC Transmitted Data (TD) signal and received on the DSP56002 Receive Data (RXD) pin. Serial data is similarly transmitted from the DSP56002 Transmit 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.

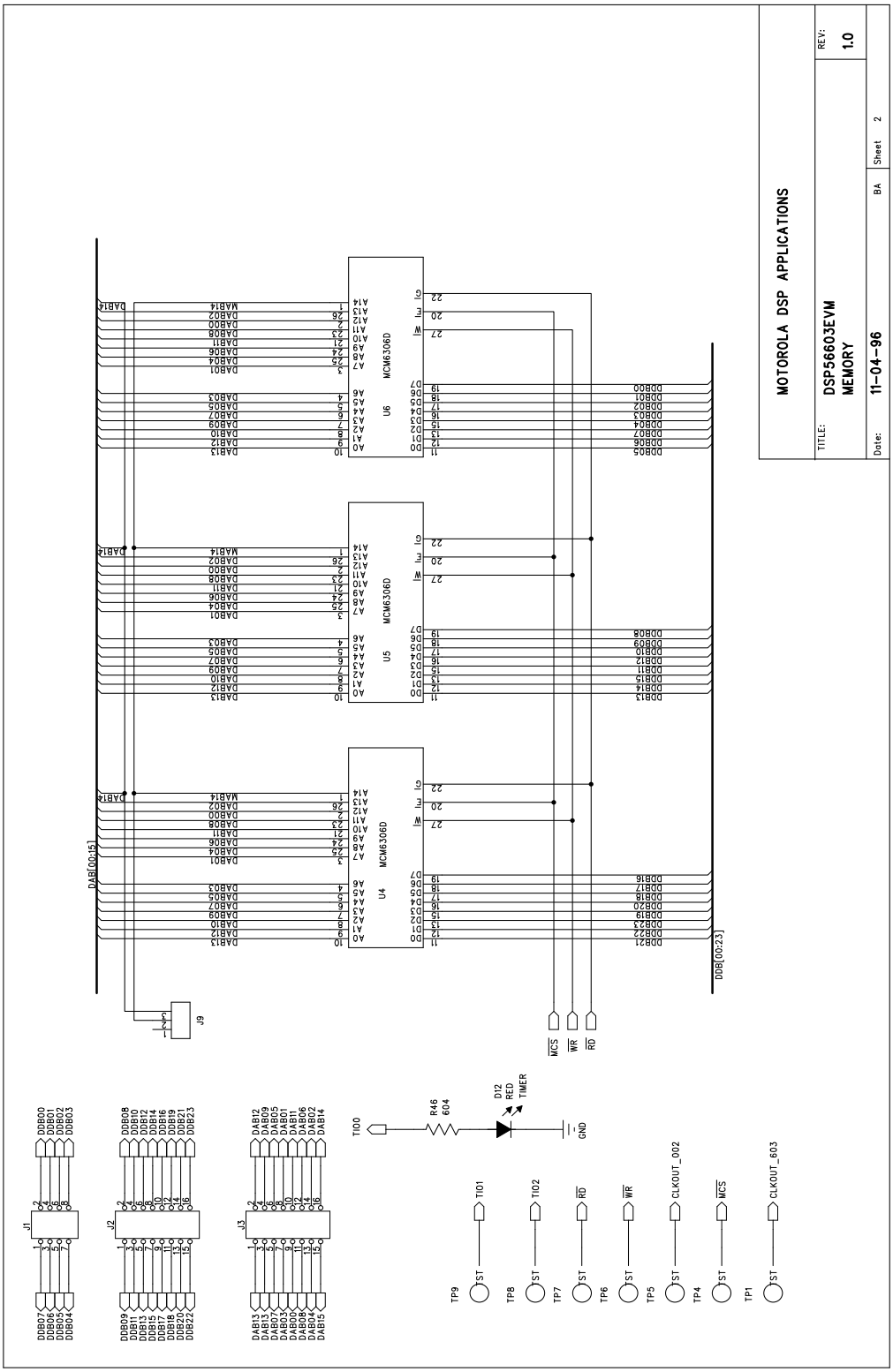
As an option, the DSP56603EVM 14-pin JTAG connector at J5 allows the user to connect an ADS command converter card directly to the DSP56603EVM if the DSP56002 command converter software is not used (J4 jumpers removed). Pin 8 has been removed from J5. The JTAG cable from the ADS command converter is similarly keyed so that the cable cannot be connected to the DSP56603EVM incorrectly.

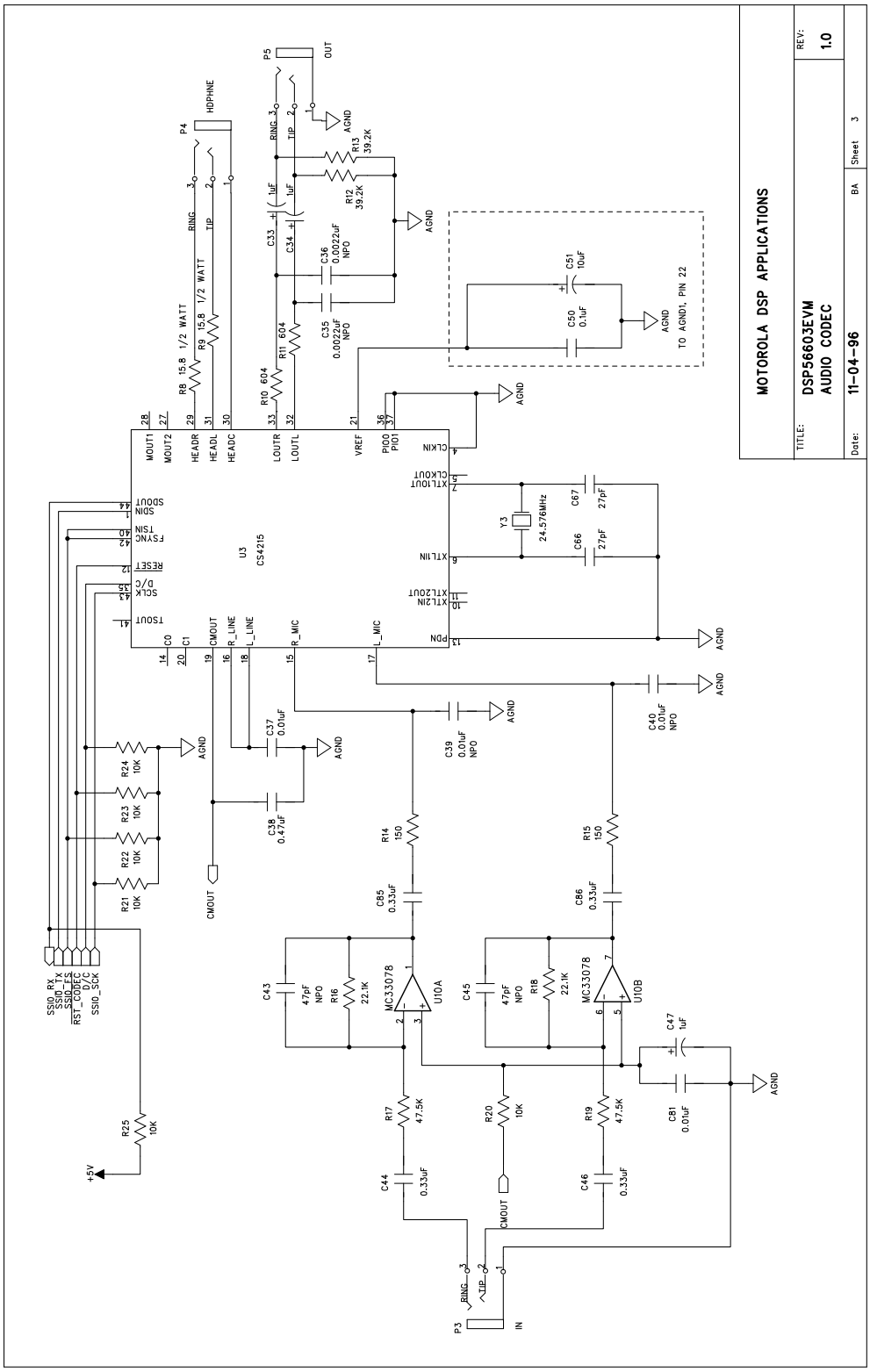
APPENDIX A

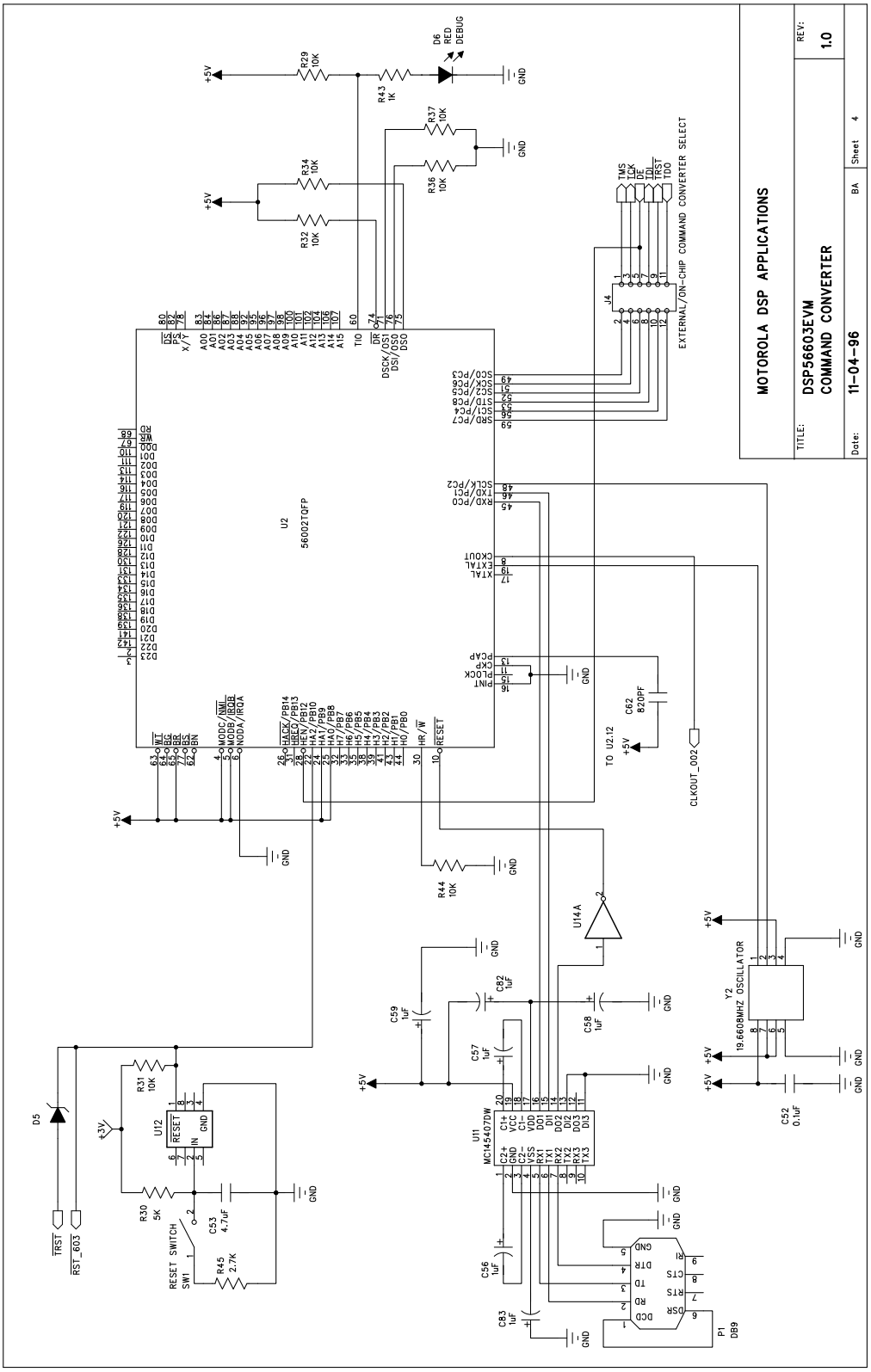
DSP56603EVM SCHEMATICS

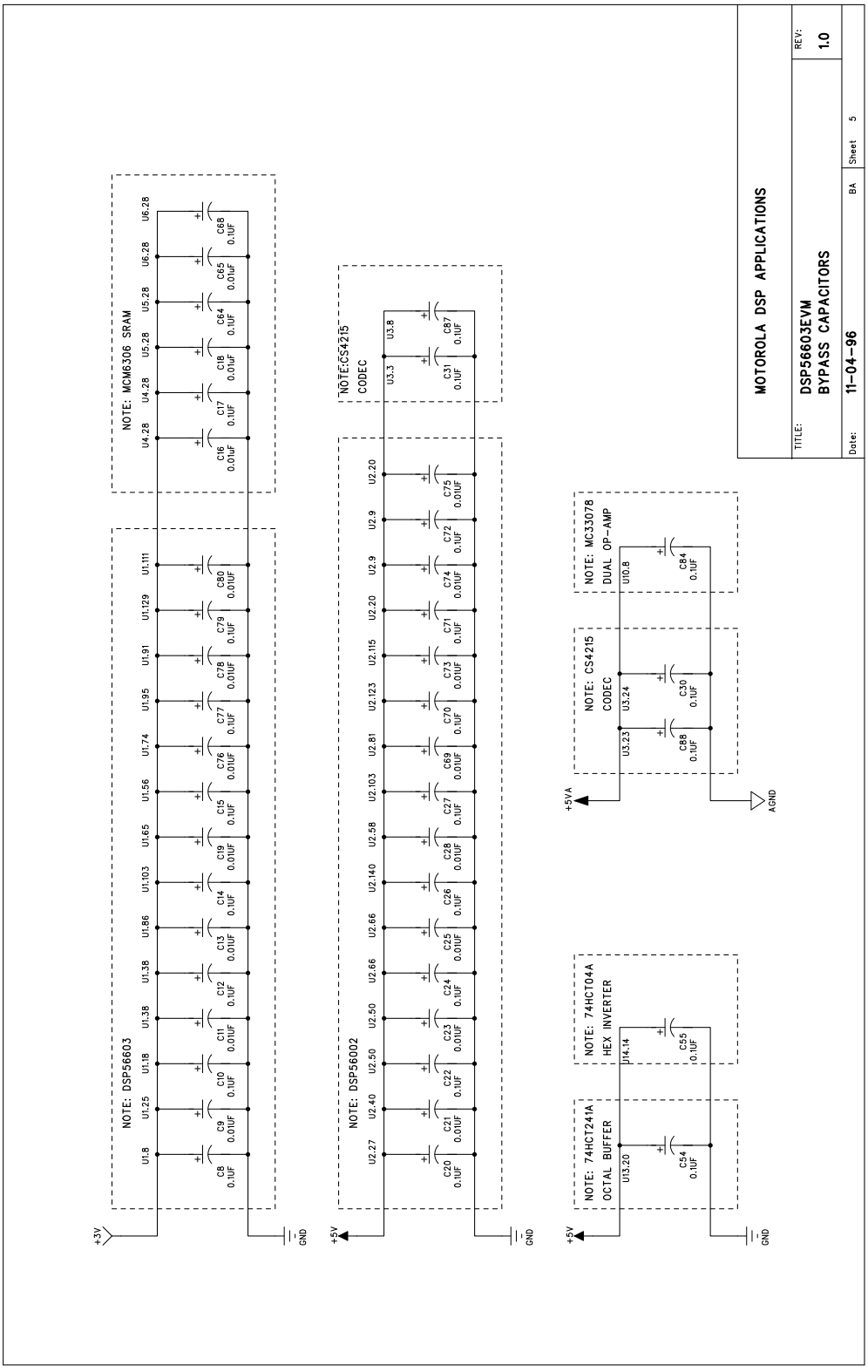
A.1 CPU SCHEMATIC A-3
A.2 MEMORY SCHEMATIC A-4
A.3 AUDIO CODEC SCHEMATIC A-5
A.4 COMMAND CONVERTER SCHEMATIC A-6
A.5 BYPASS CAPACITORS SCHEMATIC A-7
A.6 POWER SUPPLY SCHEMATIC A-8

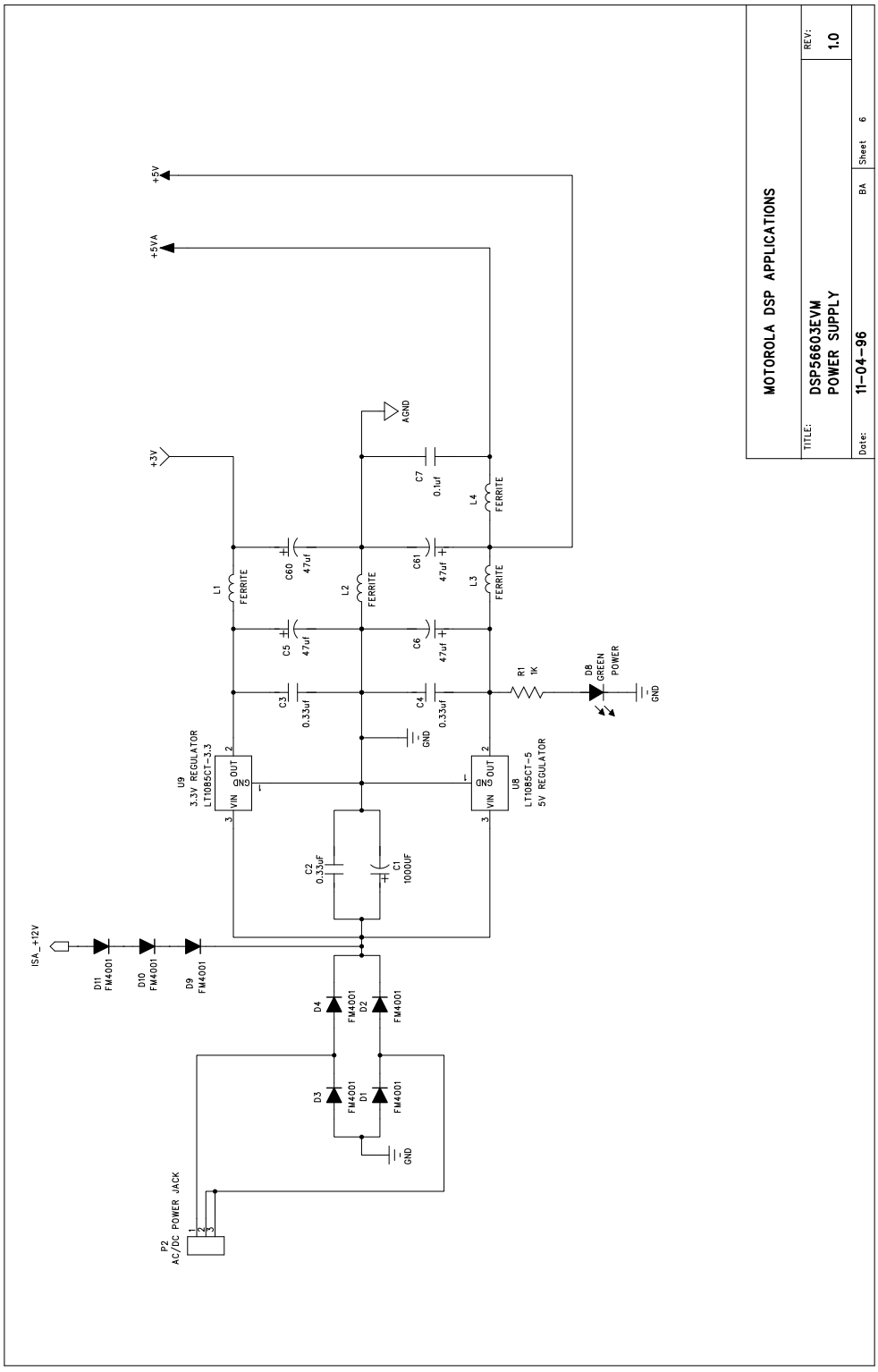

MOTOROLA DSP APPLICATIONS
TITLE:
DSP56603EVM
CPU
REV:
1.0
Date:
11-04-96
BA
Sheet 1
U1
DSP56603
U13A
74HCT241A
U13B
74HCT241A
U14B
U14C
U14D
U14E
J5
JTAG CONNECTOR
J6
SSI1/GPIO
J7
SSI0
J8
ISA CONNECTOR
SW2
MODA
SW3
MODD
Y1
16.9344MHZ
R28
680K
C41
22pF
C42
22pF
C63
1800pF
R39
10K
R35
10K
R38
10K
R7
10K
R6
10K
R5
10K
R4
10K
XTAL_603
EXTAL_603
DAB[00:15]
DDB[00:23]
TO U1.45
+3V
+5V
GND
RST_CODEC
D/C
SSI0_TX
SSI0_SCK
SSI0_FS
STD0
SCO0
SCO1
SCO2
SCK0
SCK1
SCK2
SRD0
SRD1
SCI0
SCI1
SCI2
GPIO0
GPIO1
GPIO2
TDI
TDO
TCK
DE
TMS
TRST
RST_603
CLKOUT_603
ISA_+12V
RD
WR
MCS
MODA/IRQA
MODB/IRQB
MODC/IRQC
MODD/IRQD
RESET
TIO0
TIO1
TIO2
HDS/HWR
HRW/HRD
HACK/HRRQ
HREQ/HTRQ
HCS/HA10
HA2/HA9
HA1/HA8
HA0/HA5
HAD7
HAD6
HAD5
HAD4
HAD3
HAD2
HAD1
HAD0
PINIT/NMI
EXTAL
XTAL
CLKOUT
PCAP
TDI
TDO
TCK
DE
TMS
TRST


MOTOROLA DSP APPLICATIONS
TITLE: DSP56603EVM MEMORY
REV: 1.0
Date: 11-04-96
BA
Sheet 2
U6 MCM6306D
U5 MCM6306D
U4 MCM6306D
J9
J1
J2
J3
R46 604
D12 RED TIMER
GND
TIO0
TIO1
TIO2
RD
WR
CLKOUT_002
MCS
CLKOUT_603
TP9
TP8
TP7
TP6
TP5
TP4
TP1
DAB[00:15]
DDB[00:23]


MOTOROLA DSP APPLICATIONS
TITLE: DSP56603EVM AUDIO CODEC
REV: 1.0
Date: 11-04-96
BA
Sheet 3
U3 CS4215
U10A MC33078
U10B MC33078
P3 IN
P4 HDPHNE
P5 OUT
Y3 24.576MHz
TO AGND1, PIN 22
AGND
+5V
CMOUT
SSIO_RX
SSIO_TX
SSIO_FS
RST_CODEC
D/C
SSIO_SCK


MOTOROLA DSP APPLICATIONS
TITLE: DSP56603EVM
COMMAND CONVERTER
REV: 1.0
Date: 11-04-96
BA
Sheet 4
EXTERNAL/ON-CHIP COMMAND CONVERTER SELECT
U2
560021QFP
U11
MC145407DW
U12
RESET
U14A
Y2
19.6608MHZ OSCILLATOR
P1
DB9
RESET SWITCH
SW1
TO U2.12
CLKOUT_002
TRST
RST_603
+5V
+3V
GND


MOTOROLA DSP APPLICATIONS
TITLE: DSP56603EVM
BYPASS CAPACITORS
REV: 1.0
Date: 11-04-96
BA
Sheet 5
NOTE: MCM6306 SRAM
NOTE: DSP56603
NOTE: DSP56002
NOTE:CS4215
CODEC
NOTE: MC33078
DUAL OP-AMP
NOTE: 74HCT04A
HEX INVERTER
NOTE: 74HCT241A
OCTAL BUFFER
+3V
+5V
+5VA
GND
AGND


MOTOROLA DSP APPLICATIONS
TITLE: DSP56603EVM
POWER SUPPLY
REV: 1.0
Date: 11-04-96
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 POWER
U9 3.3V REGULATOR LT1085CT-3.3
U8 LT1085CT-5 5V REGULATOR
VIN
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

DSP56603EVM PARTS LIST

B.1 PARTS LISTING

The following table contains information on the parts and devices on the DSP56603EVM.

Table B-1 DSP56603EVM Parts List

Designator	Manufacturer	Part Number	Description
U1	Motorola	DSP56603	DSP
U2	Motorola	DSP56002	DSP (OnCE)
U3	Crystal Semiconductor	CS4215KL	Audio Codec
U4 U5 U6	Motorola	MCM6306DJ15	SRAM
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
Y1	Ecliptek	EC2-169-16.9344MHZ	16.9344 MHz Crystal
Y2	ECS	OECS-196.6-3-C3X1A	19.6608 MHz Crystal

Table B-1 DSP56603EVM Parts List (Continued)

Designator	Manufacturer	Part Number	Description
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
C7 C8 C10 C12 C14 C15 C17 C20 C22 C24 C26 C27 C30 C31 C50 C52 C54 C55 C64 C68 C70 C71 C72 C77 C79 C84 C87 C88	Murata	GRM40-X7R104K025BL	0.1 µF Capacitor

Table B-1 DSP56603EVM Parts List (Continued)

Designator	Manufacturer	Part Number	Description
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
R4 R5 R6 R7 R20 R21 R22 R23 R24 R25 R29 R31 R32 R34 R35 R36 R37 R38 R39 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

Table B-1 DSP56603EVM Parts List (Continued)

Designator	Manufacturer	Part Number	Description
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 C-3
C.2 ASSEMBLER SIGNIFICANT CHARACTERS C-3
C.3 ASSEMBLER DIRECTIVES C-13
C.4 STRUCTURED CONTROL STATEMENTS C-62

C.1 INTRODUCTION

This appendix supplements information in **Section 2** 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 are 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 is aligned with the label field in the listing file. Otherwise, the comment is shifted right and aligned with the comment field in the listing file.

Example C-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 C-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 is concatenated to the previous line of the source statement, and the result is 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 C-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 C-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 is:

```
            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, is replaced by an ASCII string representing the value of <symbol>. This operator can be used with the backslash (\) operator. The value of <symbol> must be an integer (not floating point).

Example C-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 contains the following SET statements and macro call,

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

the resulting expansion as it would appear in the listing file is:

Example C-5 Example of Use of Return Value Character (Continued)

```
        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, is replaced by an ASCII string representing the hexadecimal value of <symbol>. This operator can be used in association with the backslash (\) operator. The value of <symbol> must be an integer (not floating point).

Example C-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 is:

```
HEXA        NOP
```

C.2.6 ^ Macro Local Label Override

When used as a unary expression operator in a macro expansion, the circumflex (^) causes 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 are not 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 C-7 Example of Local Label Override Character

Consider the following macro definition:

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

If this macro is called as follows,

```
_LOCAL
        LOAD    _LOCAL
```

the Assembler ordinarily issues 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 C-8 Example of a Macro String Delimiter Character

Consider the following macro definition:

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

If this macro were called as follows,

```
CSTR    ABCD
```

The resulting macro expression is:

```
        DC      'ABCD'
```

C.2.7.2 Quoted String DEFINE Expansion

A sequence of characters that matches a symbol created with a **DEFINE** directive is not 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 are expanded within the string. In all other respects, the usage of double quotes is equivalent to that of single quotes.

Example C-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 is:

```
            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 C-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 C-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 C-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> can be any legal string combination, including another substring. An error is issued if either <offset> or <length> exceed the length of <string>.

Example C-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 always picks 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 picks 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 C-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 picks the long absolute form of addressing by default:

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

Example C-14 Example of an I/O Short Addressing Mode Force Operator (Continued)

Because the Long Absolute Addressing mode causes the instruction to be two words long instead of one word for the I/O Short Absolute Addressing mode, it is desirable to force the I/O Short Absolute Addressing mode as follows:

```
            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 always picks 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 picks 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 C-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 picks the long absolute form of addressing by default:

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

Because the Long Absolute Addressing mode causes the instruction to be two words long instead of one word for the Short Absolute Addressing mode, it is desirable to force the Short Absolute Addressing mode as follows:

```
            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 always picks 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 C-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 picks 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 follows:

```
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 C-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 always picks the shortest form of immediate addressing consistent with the instruction. If the immediate data is a forward or external reference, then the Assembler picks 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 C-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 the first pass one. Therefore, the Assembler uses the long immediate addressing form for the **MOVE** instruction.

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

```
            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 follows:

```
            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 always picks 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 C-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. Therefore, the Assembler uses 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

The following subsections define the Assembler directives and their use.

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 whose the length in words is equal to the value of <expression>. The buffer type can be either **M**odulo or **R**everse-carry. If the runtime location counter is not 0, 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 is 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> can have any memory space attribute, but must be an absolute integer greater than 0 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 2, a warning is issued. A label is not allowed with this directive.

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

Example C-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 0 is assumed. If the runtime location counter is not 0, 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 occurs 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 0. Also, if the first expression is not a power of 2, a warning is generated. Both expressions can have any memory space attribute.

<label>, if present, is assigned the value of the runtime location counter after a valid base address has been established. Only one word of object code is shown on the listing, regardless of how large the first expression is. However, the runtime location counter is incremented by the number of words generated.

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

Example C-21 BSB 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 0 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 0, an error is generated. Both expressions can have any memory space attribute.

<label>, if present, is assigned the value of the runtime location counter at the start of the directive processing. Only one word of object code is shown on the listing, regardless of how large the first expression is. However, the runtime location counter is incremented by the number of words generated.

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

Example C-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 0 is assumed. If the runtime location counter is not 0, 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 occurs if the first expression contains symbols that are not yet defined (forward references), has a value of less than or equal to 0, or falls outside the range $2 \leq$ <expression> $\leq m$, where m is the maximum address of the target DSP. Both expressions can have any memory space attribute.

<label>, if present, is assigned the value of the runtime location counter after a valid base address has been established. Only one word of object code is shown on the listing, regardless of how large the first expression is. However, the runtime location counter is incremented by the number of words generated.

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

Example C-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. However, **BUFFER** directives cannot be nested, and certain types of directives such as **MODE**, **ORG**, **SECTION**, and other buffer allocation directives cannot be used. The <expression> represents the buffer size. If less data is allocated than the size of the buffer, the remaining buffer locations are 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 whose length in words is equal to the value of <expression>. The buffer type can be either **M**odulo or **R**everse-carry. If the runtime location counter is not 0, 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 is 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> can have any memory space attribute, but must be an absolute integer greater than 0 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 2, a warning is issued.

Note: A label is not allowed with this directive.

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

Example C-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> is put in the object file as a comment.

Note: A label is not allowed with this directive.

Note: See also **IDENT.**

Example C-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 is considered the last line of the comment. The comment text can include any printable characters and the comment text is reproduced in the source listing as it appears in the source file.

Note: A label is not allowed with this directive.

Example C-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> can be a numeric constant, a single or multiple character string constant, a symbol, or an expression. The **DC** directive can 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 is filled with 0s. If the **DC** directive is used in L memory, the arguments are evaluated and stored as long word quantities. Otherwise, an error occurs if the evaluated argument value is too large to represent in a single DSP word.

<label>, if present, is 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 C-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 has its remaining characters left-aligned and the rest of the word is 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 C-28 Multiple Character String Definition

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

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

Example C-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> can be a byte integer constant, a single or multiple character string constant, a symbol, or a byte expression. The **DCB** directive can 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 is zero-filled.

<label>, if present, is 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 C-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 has its remaining characters left-aligned and the rest of the word is 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 C-28**.)

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

Example C-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 are used on all following source lines. All succeeding lines are searched for an occurrence of <symbol>, which is 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 results if a new definition of a previously defined symbol is attempted. The Assembler output listing shows lines after the **DEFINE** directive has been applied. Therefore, redefined symbols are replaced by their substitution strings (unless the **NODXL** option in effect; see the **OPT** directive).

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

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

Note: A label is not allowed with this directive.

Note: See also **UNDEF.**

Example C-31 DEFINE Directive

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

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

then the source line:

```
        DS        ARRAYSIZ
```

is 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 whose length 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 0 and cannot contain any forward references (symbols that have not yet been defined). <label>, if present, is assigned the value of the runtime location counter at the start of the directive processing.

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

Example C-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 whose length in words is equal to the value of <expression>. If the runtime location counter is not 0, 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 is 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 0 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, is assigned the value of the runtime location counter after a valid base address has been established.

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

Example C-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 whose length in words is equal to the value of <expression>. If the runtime location counter is not 0, 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 is issued if there is insufficient memory remaining to establish a valid base address. Next, the runtime location counter is incremented 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 0 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 2, a warning is generated. <label>, if present, is assigned the value of the runtime location counter after a valid base address has been established.

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

Example C-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 are 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 is not 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 can be nested to any level.

<label>, if present, is 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 C-35 DUP Directive

The sequence of source input statements,

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

generates 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
```

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

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

are shown only 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 is 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, is 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 C-36 DUPA Directive

If the input source file contains the following statements:

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

then the assembled source listing shows:

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

Note that the lines

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

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

Example C-36 DUPA Directive (Continued)

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

are shown only 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 are 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, is 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 C-37 DUPC Directive

If input source file contains the following statements:

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

then the assembled source listing shows:

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

Note that the lines

Example C-37 DUPC Directive (Continued)

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

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

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

are shown only 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 is 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 can be used within the body of instructions.

<label>, if present, is 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 C-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 shows:

Example C-38 DUPF Directive (Continued)

```
        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
```

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

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

are shown only 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> can 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 C-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 remains 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 C-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 C-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 C-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.**

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 occurs 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> can be relative or absolute, but cannot include a symbol that is not yet defined (no forward references are allowed).

Note: See also **SET.**

Example C-43 EQU Directive

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

This assigns 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 is assigned to the symbol COMPUTE.

C.3.24 EXITM Exit Macro

```
EXITM
```

The **EXITM** directive causes 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 C-44 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 causes an error message to be output by the Assembler. The total error count is 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 C-45 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. If a relocatable operand value forced short is determined to be too large for the instruction word, an error occurs 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 C-46 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 is generated.

Note: A label is not allowed with this directive.

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

Example C-47 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 C-48 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 C-49 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 is 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 is also passed on to the object module.

Note: See also **COBJ.**

Example C-50 IDENT Directive

If the following line is included in the source file:

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

then the object module identification record includes 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 are included as part of the source file being assembled only if the <expression> has a nonzero result. If the <expression> has a value of 0, then the source file is 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 are assembled, and the statements between the **ELSE** and **ENDIF** directives are skipped. Alternatively, if <expression> has a value of 0, then the statements between the **IF** and **ELSE** directives are skipped, and the statements between the **ELSE** and **ENDIF** directives are 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 always refers to the nearest previous **IF** directive, as does the **ENDIF** directive.

Note: A label is not allowed with this directive.

Note: See also **ENDIF.**

Example C-51 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 C-52 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
```

The **LIST** directive prints the listing from this point on. The **LIST** directive is not printed, but the subsequent source lines are 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.

In the following sequence, the listing is not disabled until another **NOLIST** directive is issued.

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

Note: A label is not allowed with this directive.

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

Example C-53 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 is generated.

Note: A label is not allowed with this directive.

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

Example C-54 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 C-55 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>]]]]]]]

The **LSTCOL** directive 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. Widths can 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 can 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 C-56 LSTCOL Directive

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

C.3.37 MACLIB Macro Library

MACLIB <pathname>

The **MACLIB** 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> is searched for a file of the unknown name (with the .ASM extension added). If such a file is found, the current source line is saved, and the file is 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 results 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 can be given, in which case the Assembler searches each directory in the order in which it is encountered.

Note: A label is not allowed with this directive.

Note: See also **INCLUDE.**

Example C-57 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 used to call the macro. If the macro is named the same as an existing Assembler directive or mnemonic, a warning is 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 replaces 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 can be nested, but the nested macro is not defined until the primary macro is expanded.

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

Example C-58 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]>
```

The **MODE** directive causes the Assembler to change to the designated operational mode. The **MODE** directive can 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 is 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 C-59 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 causes a message to be output by the Assembler. The error and warning counts are not 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 C-60 MSG Directive

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

C.3.41 NOLIST Stop Assembly Listing

```
        NOLIST
```

The **NOLIST** directive halts printing of the listing from this point on (including the **NOLIST** directive). Subsequent source lines are not printed.

The **NOLIST** directive actually decrements a counter that is checked for a positive value and is symmetrical with respect to the **LIST** directive. In the following sequence, the listing is not disabled until another **NOLIST** directive is issued.

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

Note: A label is not allowed with this directive.

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

Example C-61 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 can also 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:

- Listing format control
- Reporting options
- Message control
- Symbol options
- 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

These miscellaneous options control internal Assembler operations :

- **CC**—Enable cycle counts
- **CK**—Enable use of checksums
- **CM**—Preserve comment lines within macros
- **CONST**—Make **EQU** symbols assembly time constants
- **CONTCK**—Continue using checksums
- **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 checks that only valid add or subtract operations are performed on address terms.
- **CC**—Enable cycle counts and clear total cycle count. Cycle counts are 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 using checksums for 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—**Maintain **EQU** symbols as assembly time constants and do not send them 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 is shown on the output listing.
- **CONTCK—**Re-enable using checksums for 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 are folded underneath the source line and aligned with the opcode field. Lines that start with the comment character are 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 are 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 stops the Assembler from producing a source listing.
- **INTR**—(default, reset in Absolute mode) Perform interrupt location checks. Certain DSP instructions are not permitted to 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 also looks 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 checksums for 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 are 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 are 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. Thus, on a line containing only a label and opcode, the comment begins 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 causes 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 because of a 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 C-62 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>—This parameter specifies which memory space (X, Y, L, P, or E) is to 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 is extended to two words; otherwise, it is truncated. If the memory space is **E**, then depending on the memory space qualifier, any generated words are split into bytes, one byte per word, or a 16/8-bit combination.
- <rlc>—This parameter specifies which runtime counter H, L, or default (if neither H or L is specified) is associated with the <rms> is to be used as the runtime location counter.

- <rmp>—This parameter 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>—This parameter indicates the 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>—This parameter indicates the 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 is used.
- <lms>—This parameter specifies which memory space (X, Y, L, P, or E) is to 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 is extended to two words; otherwise, it is truncated. If the memory space is E, then depending on the memory space qualifier, any generated words are split into bytes, one byte per word, or a 16/8-bit combination.
- <llc>—This parameter specifies which load counter, H, L, or default (if neither H or L is specified) is associated with the <lms> is to be used as the load location counter.
- <lmp>—This parameter 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>—This parameter indicates the non-negative absolute integer expression representing the counter number to be used as the load location counter. This parameter must be enclosed in parentheses, and should not exceed the value 65,535.
- <exp2>—This parameter indicates the 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 is 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 assumes 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 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, cannot refer to another overlay block.

Note: See also **MODE.**

Example C-63 ORG Directive

```
ORG  P:$1000
```

This usage of the **ORG** directive 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:
```

This usage of the **ORG** directive 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 is not initialized, and its last value is used. Subsequent generated code is 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:
```

This usage of the **ORG** directive i ndicates that code is to be generated for an overlay. The runtime memory space is P, and the default counter is used as the runtime location counter. It is 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) is used as the load location counter. The counter value and mode are whatever it was the last time it was referenced.

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

Example C-63 ORG Directive (Continued)

This usage of the **ORG** directive 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 is not initialized, and its last value is used. The load memory space is set to E, and the qualifier 8 indicates a bytewise RAM configuration. Instructions and data are generated 8 bits per output word with byte-oriented load addresses. The default load counter is used and there is no explicit load origin.

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

This usage of the **ORG** directive indicates that code is to 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 is not initialized, and the last previous value of counter 5 is 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) is used as the load location counter. The default load counter is 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 advances the listing to the top of the next page. In this case, the **PAGE** directive is not output.
2. The **PAGE** directive with arguments can be used to specify the printed format of the output listing. Arguments can 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 does not cause a page eject and is 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 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 C-64 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 is 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 C-65 PMACRO Directive

```
PMACRO    MAC1,MAC2
```

This statement causes 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>
```

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

The **PRCTL** directive can appear anywhere in the source file and the control string is 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 is 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 C-66 PRCTL Directive

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

C.3.47 RADIX Change Input Radix for Constants

```
RADIX       <expression>
```

The **RADIX** directive 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 (`). 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 C-67 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 is assumed to be a macro. Macro definitions that have the same name as Assembler directives or mnemonics 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 that has the same name as an existing directive or opcode, it does 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 results if the **RDIRECT** directive is encountered in a section.

Note: A label is not allowed with this directive.

Example C-68 RDIRECT Directive

```
            RDIRECT   PAGE,MOVE
```

This causes 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 **Structured Control Statements** on page C-62). 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 causes 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 C-69 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 C-70 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 can 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 can 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 can 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 that are split into parts or among files are logically recombined so that each section can be relocated as a unit.

Sections can 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 can 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 C-71 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 can not include a symbol that is not yet defined (no forward references are allowed).

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

Example C-72 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 is printed on the top of all succeeding pages until another **STITLE** directive is encountered. The subtitle is initially blank. The **STITLE** directive is not printed in the source listing. An **STITLE** directive with no string argument causes the current subtitle to be blank.

Note: A label is not allowed with this directive.

Note: See also **TITLE.**

Example C-73 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 C-74 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 C-75 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 is printed on the top of all succeeding pages until another **TITLE** directive is encountered. The title is initially blank. The **TITLE** directive is not printed in the source listing. A **TITLE** directive with no string argument causes the current title to be blank.

Note: A label is not allowed with this directive.

Note: See also **STITLE.**

Example C-76 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> no longer represents 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 C-77 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 causes a warning message to be output by the Assembler. The total warning count is 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 C-78 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 is generated.

Note: A label is not allowed with this directive.

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

Example C-79 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 is generated, and all references within the current section to such a symbol are 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 C-80 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 can 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. The leading period distinguishes these keywords from other directives and mnemonics. Structured control

directives can 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 the following subsections. Syntactic variables used in the formats are defined as follows:

- **<expression>—**This variable describes a simple or compound expression
- **<stmtlist>—**This variable describes 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>—**This variable describes 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>—**This variable describes the initial value of the **.FOR** loop counter. The effective address can be any mode, and can represent an arbitrary Assembler expression.

- **<op3>—**This variable describes the terminating value of the **.FOR** loop counter. The effective address can be any mode, and can represent an arbitrary Assembler expression.
- **<op4>—**This variable describes 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 can be any mode, and can represent an arbitrary Assembler expression.
- **<cnt>—**This variable describes the terminating value in a **.LOOP** statement. This can be any arbitrary Assembler expression.

All structured control statements can 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 that is illegal in those contexts.

Example C-81 .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 that is illegal in those contexts.

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

Example C-82 .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>. U se a user-defined operand, <op1>, to serve as a loop counter. **.FOR-TO** allows counting upward, while **.FOR-DOWNTO** allows counting downward. The programmer can 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 C-83 .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 C-84 .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 generate a NOP instruction just before the loop address.

Example C-85 .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 C-86 .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 C-87 .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 are 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 are referred to in the following discussion as “expressions.”

A structured control statement expression can 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 (see **Condition Code Expressions** on page C-69). 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) can 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 C-88 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 can 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 **Condition Code Expressions** on page C-69), 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 can 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.

Values in the <reg1> and <reg2> data registers are not saved before expression evaluation. This means that any user data in those registers is 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: 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. 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 can 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 can 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 that 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 can 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 only moves the operands into appropriate registers to execute the compare. The SCSREG directive can be used to reassign structured control statement registers. The MEX Assembler option (see the OPT directive) can 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 D-3
D.2 PROGRAMMING THE CODEC. D-3
D.3 ECHO.ASM PROGRAM DESCRIPTION D-3

D.1 INTRODUCTION

OK, you have a new toy and you can't wait to start developing application. But first you have 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 DSP56603EVM 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 DSP56603EVM 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 version is part of 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 DSP56600 Assembler program, select the stand-alone version of the routine, and generate an executable module. The command line, which can either be typed in or executed from a batch file, is:

```
asm56600 -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 lines 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 D-1 Program Description

```
page    132,60
;*************************************************************************
;       ECHO.ASM  Ver.2.1
;       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
;       History:
;               11 Oct 1996:  TRM/BEA - ver.2.1 :Modified to run on DSP56603EVM
;
;*************************************************************************
```

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 D-2 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 operatonal 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 four lines of the program construct the constants that define the feature selections which are made during initialization of the codec.

Example D-3 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     $0000
CTRL_WD_78      equ     $0000

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

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 two 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:0000. These are the buffers used by the interrupt handler.

Example D-4 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 (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 D-5 Sample Program Listing

```
        org P:$1000
START
main
        movep   #$0004,x:M_PCTL0        ;PLL = 5 x 16.9344 / 2 Mhz = 42.336MHz
        movep   #$0041,x:M_PCTL1
        movep   #$0001,x:M_BCR          ;1 wait state SRAM
        ENDIF
;------------------------------


        org     p:
echo
        movec   #0,SP                   ;clear stack pointer
        move    #0,SC                   ;clear stack counter
        ori     #3,mr                   ;disable interrupts
        movep   #$0004,x:M_PCTL0        ;PLL = 5 x 16.9344 / 2 Mhz = 42.336MHz
        movep   #$0041,x:M_PCTL1
        movep   #$0011,x:M_TCSR0        ;timer enabled
                                        ;output/timer pulse

        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 D-5 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)
       include         'ada_init.asm'   ;load the code init routines
       ENDIF
```

INDEX

Symbols

C-11
#< C-11
#> C-12
% C-6
* C-8
++ C-9
; C-3
;; C-4
< C-10
<< C-9
> C-10
? C-5
@ C-8
\ C-4
^ C-6, C-7

A

A/D converter 3-6
Address Attribute Pin, AA0 3-5
Address Pins, A0–A17) 3-5
Addressing
    I/O short C-9
    immediate C-11
    long C-10
    long immediate C-12
    short C-10
    short immediate C-11
Analog Circuitry Test 1-9
Analog Input/Output 3-7
assembler 2-3, 2-15
    control 2-12
    data definition/storage allocation 2-12, 2-13
    directives 2-11
    listing control and options 2-14
    macros and conditional assembly 2-14
    mode C-39
    object file control 2-14
    option C-41
    options 2-8
    significant characters 2-11
    structured programming 2-14
    symbol definition 2-12, 2-13
    warning C-48
assembling the example program 2-15
assembling the program 2-8
assembly programming 2-3
audio codec 3-3, 3-6
    clock 3-9
    crystals 3-9
Audio Codec Common Headphone Return Pin, HEADC 3-7
Audio Codec Data/Control Select Pin, D/C 3-8
Audio Codec Digital Interface 3-8
Audio Codec Frame Sync Pin, FSYNC 3-8
Audio Codec Left Headphone Output Pin, HEADL 3-7
Audio Codec Left Line Output Pin, LOUTL 3-7
Audio Codec Left Microphone Input Pin, MINL 3-7
Audio Codec Reset Pin, RESET 3-8
Audio Codec Right Headphone Output Pin, HEADR 3-7
Audio Codec Right Line Output Pin, LOUTR 3-7
Audio Codec Right Microphone Input Pin, MINR 3-7
Audio Codec Serial Data Input Pin, SDIN 3-8
Audio Codec Serial Data Output Pin, SDOUT 3-8
Audio Codec Serial Port Clock Pin, SCLK 3-8
audio interface cable 1-4, 1-9
audio source 1-4

B

Buffer
    address C-13
    end C-27

C

Checksum C-44
code example 2-6
codec 3-6
    digital interface 3-8
    digital interface connections 3-8
    programming D-1
Command Converter 3-3, 3-9
command format
    assembler 2-8
Comment C-17
    delimiter C-3
    object file C-16
    unreported C-4

comment field 2-4
Conditional assembly C-33, C-44
Constant
    define C-17
    storage C-14
Crystal Semiconductor CS4215 3-6
CS4215 3-6
Cycle count C-43, C-44

D

D / A converter 3-6
Data Pins, D(0:23) 3-5
data transfer fields 2-5
DC Offset 1-11
Debugger 2-3, 2-23
    running the 2-24
Debugger software 2-23
development process flow 2-3
Directive C-13
    .BREAK C-64
    .CONTINUE C-65
    .FOR C-65
    .IF C-66
    .LOOP C-67
    .REPEAT C-67
    .WHILE C-68
    BADDR C-13
    BSB C-13
    BSC C-14
    BSM C-15
    BUFFER C-15
    COBJ C-16
    COMMENT C-17
    DC C-17
    DEFINE C-19, C-44
    DS C-20
    DSM C-20
    DSR C-21
    DUP C-22
    DUPA C-23
    DUPC C-24
    DUPF C-25
    END C-26
    ENDBUF C-27
    ENDIF C-27
    ENDM C-28
    ENDSEC C-28
    EQU C-29
    EXITM C-29
    FAIL C-30
    FORCE C-30
    GLOBAL C-31
    GSET C-31
    HIMEM C-32
    IDENT C-32
    IF C-33
    in loop C-44
    INCLUDE C-34
    LIST C-35
    LOCAL C-35
    LOMEM C-36
    LSTCOL C-36
    MACLIB C-37
    MACRO C-38
    MODE C-39
    MSG C-39
    NOLIST C-40
    OPT C-41
    ORG C-48
    PAGE C-51
    PMACRO C-52
    PRCTL C-53
    RADIX C-53
    RDIRECT C-54
    SCSJMP C-55
    SCSREG C-55
    SECTION C-56
    SET C-58
    STITLE C-59
    SYMOBJ C-59
    TABS C-59
    TITLE C-60
    UNDEF C-60
    WARN C-61
    XDEF C-61
    XREF C-62
Domain Technologies Debugger 1-3, 1-8, 2-23
DSP development tools 2-3
DSP linker 2-15, 2-16
DSP56002 3-9
DSP56002 Receive Data Pin, RXD 3-10
DSP56002 Transmit Data Pin, TXD 3-10
DSP56300 Family Manual 3-3
DSP56303 2-3
    Chip Errata 3-3
    Product Specification 1-3
    Product Specification, Revision 1.02 3-3

Technical Data 1-3, 3-3
User's Manual 3-3
DSP56303 code example 2-6
DSP56303EVM
additional requirements 1-4
Component Layout 3-5
component layout 1-6
connecting to the PC 1-7
contents 1-3
description 3-3
features 3-3
functional block diagram 3-4
installation procedure 1-4
interconnection diagram 1-7
jumper settings 1-6
memory 3-4
power connection 1-7
preparation for installation 1-5
Product Information 1-3
software installation 1-8
SRAM 3-4
testing the installation 1-9
User's Manual 1-3

E

ECHO.ASM file D-3
ESD warning 1-5
ESSI0 3-6
example program 2-5
assembling 2-15
Expression
address C-43
compound C-70
condition code C-69
formatting C-71
operand comparison C-69
radix C-53
simple C-68
External Memory Test 1-9

F

field
comment 2-5
data transfer 2-5
label 2-4
operand 2-5
operation 2-4
X data transfer 2-4
Y data transfer 2-4
File
include C-34
listing C-45
format
assembler command 2-8
source statement 2-4
Function C-8

H

headphones 1-4, 1-9
Host Address Pin, HA2 3-9
host PC 3-9
Host PC Data Terminal Ready Pin, DTR 3-10
Host PC Receive Data Pin, RD 3-10
host PC requirements 1-4
Host PC Transmit Data Pin, TD 3-10

I

Include file C-34
installing software 1-8

J

J4 3-10
J5 3-10
J7 3-6
JTAG 3-9
jumper settings 1-6

K

kit contents 1-3

L

Label
local C-45, C-48
label field 2-4
Line continuation C-4
linker 2-3, 2-15, 2-16
directives 2-22
options 2-16
Listing file C-45
format C-36, C-44, C-45, C-47, C-51, C-59
sub-title C-59
title C-60
Location counter C-8, C-48

M

Macro
- argument
  - concatenation operator C-4
  - local label override operator C-6, C-7
  - return hex value operator C-6
  - return value operator C-5
- call C-45
- comment C-44
- definition C-38, C-45
- directive C-38
- end C-28
- exit C-29
- expansion C-45
- library C-37, C-45
- purge C-52

MC145407 3-10
MC33078 3-7
MC74HCT241A 3-6
Memory
- limit C-32, C-36
- spaces C-45, C-48
- utilization C-45

Motorola
- software 1-8
  - DSP linker 2-15, 2-16

N

Noise Level 1-11

O

Object file 2-3
- comment C-16
- identification C-32
- symbol C-47, C-59

OnCE commands 3-9
OnCE/JTAG conversion 3-9
operand field 2-5
operation field 2-4
Option
- AE C-42, C-43
- assembler operation C-43
- CC C-43
- CEX C-41, C-43
- CK C-43, C-44
- CL C-41, C-44
- CM C-43, C-44
- CONST C-43, C-44
- CONTC C-44
- CONTCK C-43, C-44
- CRE C-41, C-44
- DEX C-42, C-44
- DLD C-43, C-44
- DXL C-41, C-44
- FC C-41, C-44
- FF C-41, C-44
- FM C-41, C-44
- GL C-43, C-44
- GS C-43, C-44
- HDR C-41, C-45
- IC C-42, C-45
- IL C-42, C-45
- INTR C-43, C-45
- LB C-43, C-45
- LDB C-43, C-45
- listing format C-41
- LOC C-42, C-45
- MC C-42, C-45
- MD C-42, C-45
- message C-42
- MEX C-42, C-45
- MI C-43, C-45
- MSW C-42, C-45
- MU C-42, C-45
- NL C-42, C-45
- NOAE C-45
- NOCC C-45
- NOCEX C-45
- NOCK C-46
- NOCL C-46
- NOCM C-46
- NODEX C-46
- NODLD C-46
- NODXL C-46
- NOFC C-46
- NOFF C-46
- NOFM C-46
- NOGS C-46
- NOHDR C-46
- NOINTR C-46
- NOMC C-46
- NOMD C-46
- NOMEX C-46
- NOMI C-46
- NOMSW C-46
- NONL C-46

NONS C-46
NOPP C-46
NOPS C-46
NORC C-46
NORP C-46
NOSCL C-46
NOU C-47
NOUR C-47
NOW C-47
NS C-42, C-47
PP C-41, C-47
PS C-43, C-47
PSM C-43
RC C-41, C-47
reporting C-41
RP C-43, C-47
RSV C-43
S C-42, C-47
SCL C-42, C-47
SCO C-42, C-47
SI C-43
SO C-42, C-47
SVO C-43
symbol C-42
U C-42, C-47
UR C-42, C-48
W C-42, C-48
WEX C-48
XLL C-42, C-48
XR C-42, C-48

P

PC 3-9
PC requirements 1-4
power supply, external 1-4, 1-7
program
  assembling the 2-8
  example 2-5
  writing the 2-4
Program counter C-8, C-48
programming
  assembly 2-3
  development 2-3
  example 2-3

Q

Quick Start Guide 1-1

R

Read Enable Pin, RD 3-5
Reset, DSP56002 3-10
RS-232 cable connection 1-7
RS-232 interface 3-9
RS-232 interface cable 1-4
RS-232 serial interface 3-9
running the Debugger program 2-24

S

SCI, DSP56002 3-9
Section C-56
  end C-28
  global C-31, C-44, C-56
  local C-35, C-57
  nested C-47
  static C-44, C-57
self-test 1-9
Serial Clock Pin, SCK0 3-8
Serial Control Pin 0, SC00 3-8
Serial Control Pin 1, SC01 3-8
Serial Control Pin 2, SC02 3-8
serial interface 3-9
Serial Receive Data Pin, SRD0 3-8
Serial Transmit Data Pin, STD0 3-8
software
  Domain Technologies 1-8
  Motorola 1-8
software installation 1-8
Source file
  end C-26
source statement format 2-4
SRAM 3-4
SRAM Address Pins, A(0:14) 3-5
SRAM Chip Enable Pin, E 3-5
SRAM Data Pins, DQ(0:7) 3-5
SRAM Output Enable Pin, G 3-5
SRAM Write Enable Pin, W 3-5
Stereo Headphones 3-7
Stereo Input 3-7
Stereo Output 3-7
String
  concatenation C-9
  packed C-47
Symbol
  case C-45
  cross-reference C-44
  equate C-29, C-44
  global C-44

listing C-47
set C-31, C-58
undefined C-48

T

test
analysis 1-9, 1-10
self 1-9
tutorial, codec programming D-1

W

Warning C-48
Windows 1-8
Write Enable Pin, WR 3-5

X

X data transfer field 2-4

Y

Y data transfer field 2-4