TMS320C54x DSP Reference Set

Volume 4: Applications Guide

This document contains preliminary data current as of publication date and is subject to change without notice.

Literature Number: SPRU173
Manufacturing Part Number: D425009-9761 revision *
October 1996

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Preface

Read This First

About This Manual

The purpose of this book is to present integrated solutions for typical TMS320C54x design issues. It combines a description of specific programming topics with code examples. The text discusses basic programming topics for the ’54x digital signal processor (DSP). The code examples were created and tested in real time using the ’54x evaluation module (EVM) as a platform. You may use these examples in developing your applications.

How to Use This Manual

The book is divided into two parts: topic information, provided in Chapters 1–9, and complete code examples, provided in Chapter 10.

- Topic information, Chapters 1–9: These chapters give you a framework of knowledge for programming the ’54x. Before creating code, beginners may want to read these chapters entirely to understand why these issues must be addressed in certain ways. Advanced users may want to read only the topics relevant to specific code applications.
- Complete code examples, Chapter 10: These examples elaborate on the code provided in Chapters 1–9. This code has been tested and can be run as is.

Notational Conventions

This document uses the following conventions.

- Program listings and program examples are shown in a special font.

  Here is a sample program listing:

```
STL     A, *AR1+          ; Int_RAM(I)=0
RSBX    INTM              ; Globally enable interrupts
B       MAIN_PG           ; Return to foreground program
```

- Throughout this book, the notation ’54x refers to the TMS320C54x and the TMS320VC54x. The notations ’541, ’542, etc., refer to the TMS320C541, TMS320C542, etc. The notation ’LC548 refers to the TMS320LC548.

Related Documentation from Texas Instruments

The following books describe the ’54x and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number.

TMS320C54x DSP Reference Set (literature number SPRU210) is composed of four volumes of information, each with its own literature number for individual ordering.

TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals (literature number SPRU131) describes the TMS320C54x 16-bit, fixed-point, general-purpose digital signal processors. Covered are its architecture, internal register structure, data and program addressing, the instruction pipeline, DMA, and on-chip peripherals. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set (literature number SPRU172) describes the TMS320C54x digital signal processor mnemonic instructions individually. Also includes a summary of instruction set classes and cycles.

TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set (literature number SPRU179) describes the TMS320C54x digital signal processor algebraic instructions individually. Also includes a summary of instruction set classes and cycles.

TMS320C54x DSP Reference Set, Volume 4: Applications Guide (literature number SPRU173) describes software and hardware applications for the TMS320C54x digital signal processor. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

TMS320C54x, TMS320LC54x, TMS320VC54x Fixed-Point Digital Signal Processors (literature number SPRS039) data sheet contains the electrical and timing specifications for these devices, as well as signal descriptions and pinouts for all of the available packages.

TMS320C54x Assembly Language Tools User’s Guide (literature number SPRU102) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the ’C54x generation of devices.

TMS320C5xx C Source Debugger User's Guide (literature number SPRU099) tells you how to invoke the 'C54x emulator, EVM, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C54x Code Generation Tools Getting Started Guide (literature number SPRU147) describes how to install the TMS320C54x assembly language tools and the C compiler for the 'C54x devices. The installation for MS-DOS™, OS/2™, SunOS™, Solaris™, and HP-UX™ 9.0x systems is covered.

TMS320C54x Evaluation Module Technical Reference (literature number SPRU135) describes the 'C54x EVM, its features, design details and external interfaces.

TMS320C54x Optimizing C Compiler User's Guide (literature number SPRU103) describes the 'C54x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C54x generation of devices.

TMS320C54x Simulator Getting Started (literature number SPRU137) describes how to install the TMS320C54x simulator and the C source debugger for the 'C54x. The installation for MS-DOS™, PC-DOS™, SunOS™, Solaris™, and HP-UX™ systems is covered.

TMS320 Third-Party Support Reference Guide (literature number SPRU052) alphabetically lists over 100 third parties that provide various products that serve the family of '320 digital signal processors. A myriad of products and applications are offered—software and hardware development tools, speech recognition, image processing, noise cancellation, modems, etc.

TMS320 DSP Development Support Reference Guide (literature number SPRU011) describes the TMS320 family of digital signal processors and the tools that support these devices. Included are code-generation tools (compilers, assemblers, linkers, etc.) and system integration and debug tools (simulators, emulators, evaluation modules, etc.). Also covered are available documentation, seminars, the university program, and factory repair and exchange.

Trademarks

Borland is a trademark of Borland International, Inc.

HP-UX is a trademark of Hewlett-Packard Company.

MS-DOS is a registered trademark of Microsoft Corporation.

OS/2, PC/AT, and PC-DOS are trademarks of International Business Machines Corporation.

PAL® is a registered trademark of Advanced Micro Devices, Inc.

Solaris and SunOS are trademarks of Sun Microsystems, Inc.

SPARC is a trademark of SPARC International, Inc., but licensed exclusively to Sun Microsystems, Inc.

Windows is a registered trademark of Microsoft Corporation.

320 Hotline Online, TI, XDS510, and XDS510WS are trademarks of Texas Instruments Incorporated.

If You Need Assistance. . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.html

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/mirrors/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32	Email: epic@ti.com
Deutsch +49 8161 80 33 11 or	+33 1 30 70 11 68		
English	+33 1 30 70 11 65		
Francais	+33 1 30 70 11 64		
Italiano	+33 1 30 70 11 67		
EPIC Modem BBS	+33 1 30 70 11 99		
European Factory Repair	+33 4 93 22 25 40		
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10	

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: comments@books.sc.ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Figures

Tables

Examples

Equations

Chapter 1

Introduction

The TMS320C54x is a fixed-point digital signal processor (DSP) in the TMS320 family. It provides many options for the design of telecommunication and wireless applications. It executes 50 million instructions per second (MIPS), providing high performance, low power consumption, and cost effectiveness.

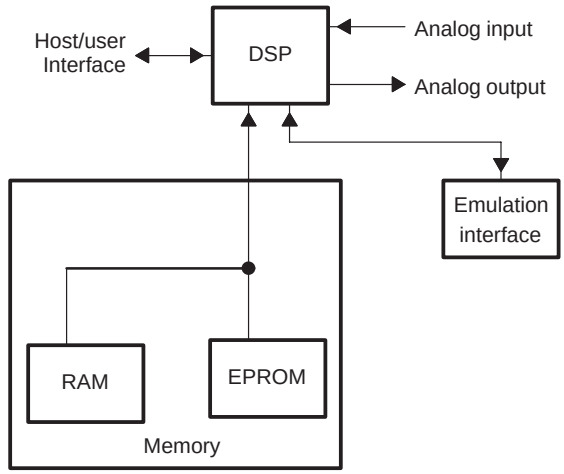
The code examples in this book were tested in real time using a ’54x Evaluation Module (EVM) platform. This chapter introduces you to DSP and EVM architecture and describes the ’54x memory interface mechanism and the EVM’s interface with memory devices. Since the ’54x EVM contains a ’541 DSP, the chapter discusses the ’541 specifically.

Topic **Page**

1.1 DSP Overview

The block diagram in Figure 1–1 represents a typical DSP system. It uses an analog interface, where an input signal is digitized and processed by the DSP and has an output terminal. The RAM and EPROM blocks make up the system's memory. These blocks sometimes replace with DSP on-chip memory. For a stand-alone system, an EPROM bootloads the code during system power-up. The emulation interface can access the '54x high-level language debuggers, factory-installed tests, and field diagnostics. The host can download program files or data through the emulation port. The host interface provides buffering, host I/O decode, and access control.

Figure 1–1. Typical DSP System



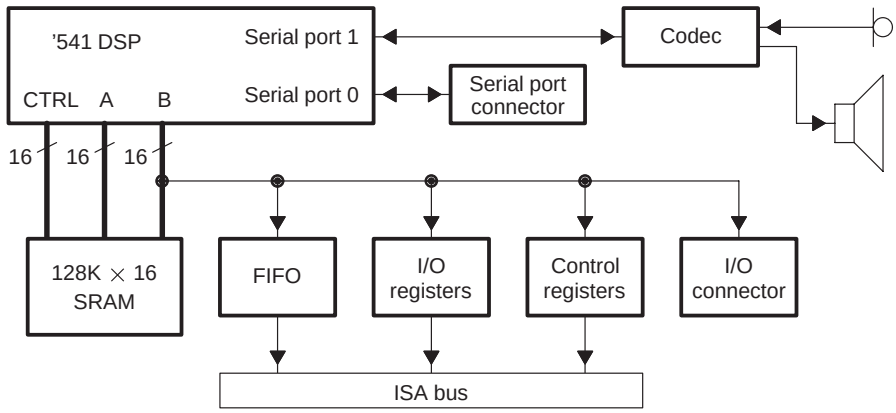
1.2 '54x Evaluation Module (EVM) Overview

The '54x EVM is a PC/AT™ plug-in half-card that consists of a '541 DSP, 128K × 16 words of fast SRAM, and a TLC320AC01 analog interface chip (AIC). It also includes a programmer interface and a mouse-driven, window-oriented C source debugger. It provides a platform for using real-time device evaluation, benchmarking, and system debugging to design embedded systems.

The EVM has all common software debugging features, including breakpoint, modify, load, and watch. User-defined windows can be set up to view device parameters. The '54x EVM comes with a full assembler/linker package and an expansion port for interfacing to peripherals. Other hosts, systems, and target boards can communicate with the EVM using the serial port connector, provided on the board. The EVM allows you to prototype application software and hardware. The example code provided in chapter 10 uses '54x EVM as the hardware platform along with a host interface and demonstrates the various applications using signal processing techniques.

Figure 1–2 shows the configuration of the '54x EVM. It interfaces to 128K words of zero wait-state static RAM. The EVM includes 64K words of zero wait-state program memory and 64K words of zero wait-state data memory. An external I/O interface supports 16 parallel I/O ports and a serial port. The host-target communication system provides a simple means to pass data between the target and host during real-time operation. The two channels, A and B, are single, 16-bit bidirectional registers, mapped into two I/O port locations. Channel B has a 64-word deep FIFO buffer. The analog interface circuit (AIC) interfaces to the '541 serial port. The codec is a TLC320AC01 AIC that provides a 14-bit A/D and D/A interface with 16 bits of dynamic range, and sampling rates up to 43.2 kHz. Two RCA connectors provide analog input and output for 'AC01.

Figure 1–2. Block Diagram of a '54x EVM



1.3 Memory Interface

The ’54x has several types of memory interfaces. Each interface contains a 16-bit address bus and 16-bit data bus signal lines. These transfer information and control interface operation. All of the interfaces are independent of one another, and different operations can be performed simultaneously on each interface.

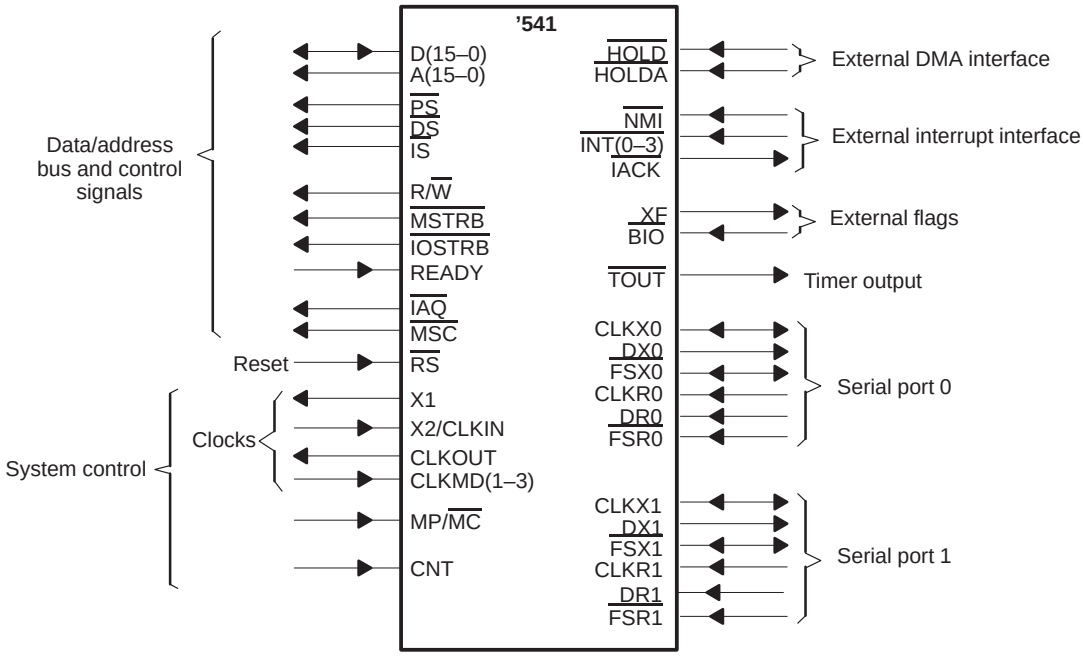
The external direct memory access (DMA) interface lets external devices cause the processor to give up the bus and control signals for DMA. The $\overline{\text{MSTRB}}$ signal is activated for program and memory accesses, and the $\overline{\text{IOSTRB}}$ signal is used for transactions with I/O ports (PORTR and PORTW instructions). The R/$\overline{\text{W}}$ signal controls the direction of accesses. The external ready input signal (READY) and the software-programmable wait-state generators allow the processor to interface with memory and I/O devices of varying speeds.

Two signals, $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$, allow an external device to take control of the processor’s buses. The processor acknowledges receiving a $\overline{\text{HOLD}}$ signal from an external device by bringing $\overline{\text{HOLDA}}$ low. The $\overline{\text{RS}}$ signal initializes the internal ’541 logic and executes the system-initialization software. The $\overline{\text{PS}}$, $\overline{\text{DS}}$, and $\overline{\text{I/O}}$ select signals are used to select any external program, data, or I/O access. The connection of X1 and X2/CLKIN determines the clock source that drives the clock generator. The clock mode that operates the clock generator is determined by the CLKMD(1–3) signals.

Figure 1–3 shows the external interfaces on the ’541. The device contains two independent bidirectional serial ports: serial port 0 and serial port 1. The analog interface circuit interfaces directly to the serial ports of the ’541. The external flag, $\overline{\text{BIO}}$, is used to monitor the status of the peripheral devices. The XF pin signals external devices via software. The ’541 has four external, maskable user interrupts that external devices can use to interrupt the processor and one external, nonmaskable interrupt ($\overline{\text{NMI}}$). These provide a convenient means to perform periodic I/O or other functions.

The ’541 can be interfaced with EPROMs and static RAMs. The speed, cost, and power limitations imposed by an application determine the selection of memory device. If speed and maximum throughput are important, the ’541 can run with no wait states. In this case, memory accesses are performed in a single machine cycle. Slower memories can be accessed by introducing an appropriate number of wait states or slowing down the system clock.

Figure 1–3. External Interfaces on the ’541



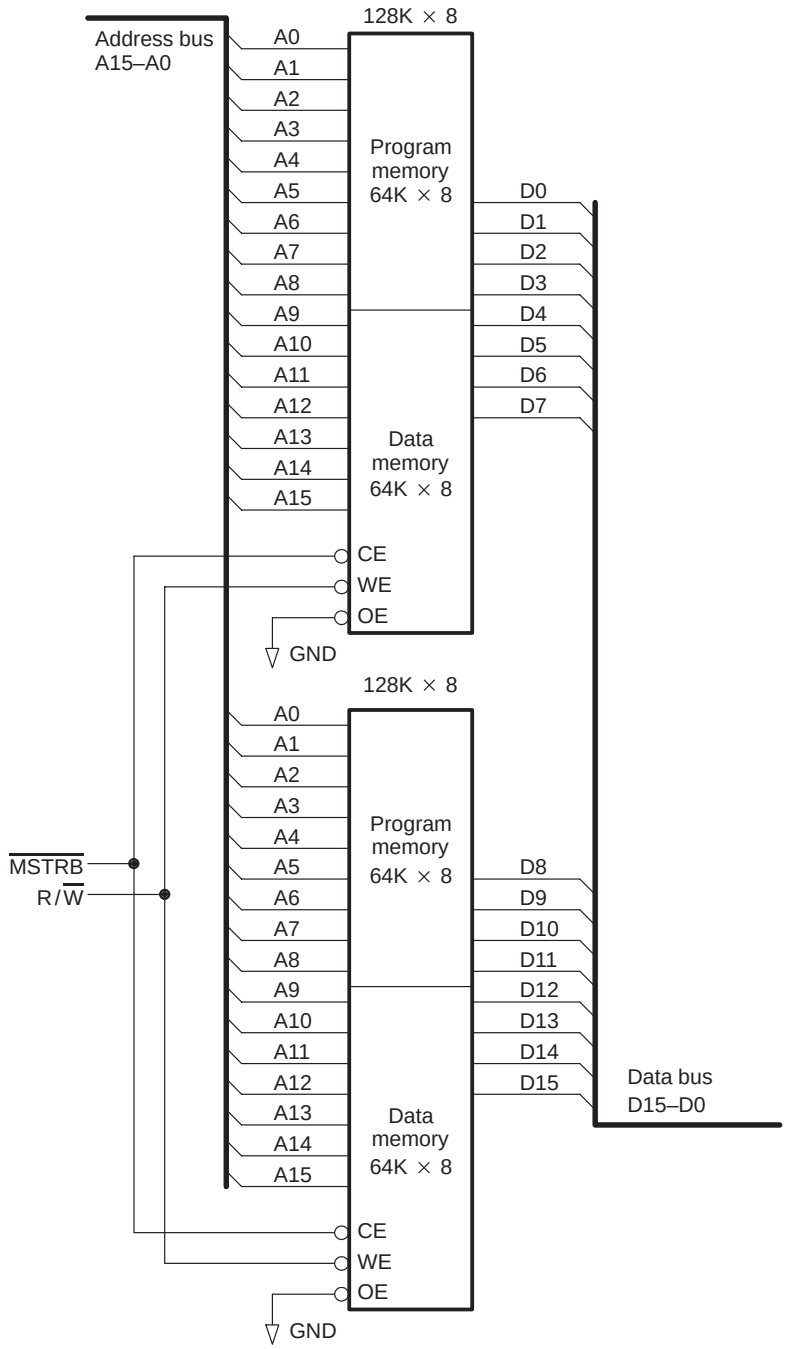
1.4 '54x EVM External Memory Interface

The '54x EVM includes 64K words of zero wait-state program memory and 64K words of zero wait-state data memory, providing a total of 128K words of external memory. The $\overline{PS}$ line controls accesses to data and program memory. Each time the DSP accesses external program memory, the $\overline{PS}$ line goes low, driving the A16 address line low and forcing access to memory locations 00000h – 0FFFFh. Data memory accesses have no effect on the $\overline{PS}$ line. During data memory accesses, the $\overline{PS}$ line remains high. This means that the EVM accesses memory locations from 100000h – 1FFFFh. These accesses are tabulated in the truth table below.

Access	$\overline{MSTRB}$	$\overline{PS}$	$\overline{DS}$
Program	0	0	1
Data	0	1	0
No access	1	1	1

Figure 1–4 shows a diagram of program/data memory interfaces for the EVM.

Figure 1–4. '54x EVM Interface to External SRAM for Program and Data Memory



Chapter 2

System Start-Up

To successfully power up a system, you must have a clear understanding of reset signals, clock modes and sources, memory interfaces with and without wait states, and bank switching. Common methods to start up a program include using on-chip ROM, external 16-bit EPROM, and bootloading from an 8-bit EPROM. This chapter examines these start-up methods and parameters and gives examples of initialization software.

Topic **Page**

2.1 On-Chip ROM/External 16-Bit EPROM

The ’54x program memory space can reside either on- or off-chip. On-chip memory can include RAM, ROM, and external SRAM; off-chip memory can include 16-bit EPROM. There are two common methods to execute a program: running from memory (which uses on-chip ROM or 16-bit external EPROM) or using the bootloader either serially or in parallel from a 16- or an 8-bit EPROM.

The system can run using a 16-bit external EPROM or on-chip ROM. With on-chip ROM, internal memory is enabled. The EVM uses 128K words of external memory, which includes 64K words each of program and data memory. If a program resides in the external memory space, the program space includes part of the 64K of program memory. The EVM has external SRAM that can emulate either on-chip ROM or external EPROM. Executing the code from on-chip ROM requires no wait states, whereas executing from external EPROM requires some wait states, depending upon the speed of the DSP and the speed of the EPROM. In either case, the program starts executing from the reset vector, FF80h.

Program memory in on-chip ROM is configured internally, since the processor is in microcomputer mode when the MP/$\overline{\text{MC}}$ pin is low. For 16-bit EPROM, the program space is external. The processor is configured in microprocessor mode when MP/$\overline{\text{MC}}$ is high.

2.2 Processor Initialization

At power-up, the state of the ’54x processor is partially defined, since all the bits in both status control registers are established. Thus, the processor is in a predefined condition at reset. Some of the conditions in these registers can be changed, depending upon the system configuration and the application. The reset signal puts the processor in a known state. In a typical DSP system, the processor must be initialized for proper execution of an application.

Starting up the system from on-chip ROM or 16-bit EPROM is done by resetting the processor. In both cases, the processor is reset by applying a low level to the $\overline{RS}$ pin. Reset causes the processor to fetch the reset vector. Upon reset, the IPTR bits of the PMST register are all set to 1, mapping the vectors to page 511 in program memory space. This means that the reset vector always resides at program memory location 0FF80h. This location usually contains a branch instruction to direct program execution to the system initialization routine.

After a reset, the processor initializes the following internal functions:

- Stack pointer (SP)
- Memory-mapped peripheral control registers (SWWSR and BSCR)
- Status registers (ST0 and ST1)
- Control register (PMST)

Some of the reset bits in the status and control registers can be changed during initialization. The ’54x has a software stack and the stack pointer must be initialized. The data page pointer, which is initialized to page 0 during reset, must also be configured during initialization. The predefined and the remaining bits in ST0, ST1, and PMST are initialized so that the processor starts executing the code from a defined state.

Software wait-state generators interface external memory devices to the processor. Programmable bank switching determines the bank size. The software wait-state register (SWWSR) and the bank-switching control register (BCSR) are initialized.

Example 2–1 initializes the vector table. Example 2–2, on page 2-6, initializes the ’54x, where the processor branches to main_start upon reset.

Example 2–1. Vector Table Initialization — vectors.asm

```
; TEXAS INSTRUMENTS INCORPORATED
          .mmregs
          .include        ”init_54x.inc”
          .include        ”main.inc”
          .ref            SYSTEM_STACK
          .ref            main_start
          .ref            receive_int1
          .ref            host_command_int1
;-------------------------------------------------------------------------------
;  Functional Description
;           This function initializes the vector table of 541 device
;-------------------------------------------------------------------------------
          .sect           ”vectors”
reset: BD main_start                      ; RESET vector
          STM             #SYSTEM_STACK,SP
nmi:      RETE
          NOP
          NOP
          NOP                             ;NMI~
; software interrupts
sint17                    .space  4*16
sint18                    .space  4*16
sint19                    .space  4*16
sint20                    .space  4*16
sint21                    .space  4*16
sint22                    .space  4*16
sint23                    .space  4*16
sint24                    .space  4*16
sint25                    .space  4*16
sint26                    .space  4*16
sint2                     .space  4*16
sint28                    .space  4*16
sint29                    .space  4*16
sint30                    .space  4*16
int0:     RETE
          NOP
          NOP                             ; INT0
          NOP
int1:     BD     host_command_int1        ; Host interrupt
          PSHM   ST0
          PSHM   ST1                      ; INT1
int2:     RETE
          NOP
          NOP
          NOP
tint:     RETE
          NOP
          NOP                             ; TIMER
          NOP
```

Example 2–1. Vector Table Initialization — vectors.asm (Continued)

```
rint0:    RETE                              ; Serial Port Receive
          NOP                               ; Interrupt 0
          NOP
          NOP
xint0:    RETE                              ; Serial Port Transmit
                                            ; Interrupt 0
          NOP
          NOP
          NOP
rint1:    BD     receive_int1               ; Serial Port Receive
          PSHM   ST0                        ; Interrupt 1
          PSHM   ST1
xint1:    RETE                              ; Serial Port Transmit
          NOP                               ; Interrupt 1
          NOP
          NOP
int3:     RETE
          NOP
          NOP                               ; INT3
          NOP
          .end

*-------------------------------------------------------------------------------
```

Example 2–2. Processor Initialization — init_54x.asm

```
; TEXAS INSTRUMENTS INCORPORATED
    .mmregs
    .include  ”init_54x.inc”
    .include  ”main.inc”
    .ref      d_frame_flag
    .ref      RCV_INT1_DP
    .ref      aic_init,serial_init,init_54,init_bffr_ptr_var
    .ref      task_handler,evm_handshake,fifo_host_transfer
STACK         .usect    ”stack”,K_STACK_SIZE
SYSTEM_STACK  .set      K_STACK_SIZE+STACK
    .def      main_start
    .def      SYSTEM_STACK
;
;-------------------------------------------------------------------------------
;  Functional Description
;  This is the main function that calls other functions.
;-------------------------------------------------------------------------------
          .sect  ”main_prg”
*******************************************************************************
* The code initializes the 541 device, handshake between Target (DSP)
* and the host (PC). Zeros all buffers, variables and init. pointers
* Initializes serial port, programs AC01 registers for selecting sampling
* rate, gains etc.
 ******************************************************************************
main_start:
    CALL      init_54                ; initialize ST0,ST1 PMST and
                                     ;other registers
    .if       K_HOST_FLAG = 1
    CALL      evm_handshake          ; EVM host handshake
    .endif
    CALL      init_bffr_ptr_var      ; init tables,vars,bffrs,ptr
    CALL      serial_init            ; initialize serial_port 1
    CALLD     aic_init               ; Configures AC01
    LD        #0,DP
    NOP
*******************************************************************************
* After enabling interrupts from the above, the real processing starts here
* After collecting 256 samples from AC01 a flag(d_frame_flag is set).
* Handles the task initiated by the user and transfers the data to the
* host. Keeps the sequence for ever !!!!!
*******************************************************************************
start_loop
       LD     #RCV_INT1_DP,DP        ; restore the DP
loop:
    BITF      d_frame_flag,1         ; if 256 samples are received
    BC        loop,NTC               ; if not just loop back
    CALL      task_handler           ; handles task scheduling
    CALL      fifo_host_transfer     ; EVM HOST interface
    B         loop
    .end
```

Example 2–2. Processor Initialization — init_54x.asm (Continued)

```
* Filename: Main.inc
* Includes all the constants that are used in the entire application
K_0              .set   0      ; constant
K_FIR_INDEX      .set   1      ; index count
K_FIR_BFFR       .set   16     ; FIR buffer size
K_neg1           .set   –1h    ; index count
K_BIQUAD         .set   2      ; there are 2 bi-quad sections
K_IIR_SIZE       .set   10     ; each bi-quad has 5 coeffs
K_STACK_SIZE     .set   200    ; stack size
K_FRAME_SIZE     .set   256    ; PING/PONG buffer size
K_FRAME_FLAG     .set   1      ; set after 256 collected
H_FILT_SIZE      .set   128    ; H(z) filter size
ADPT_FILT_SIZE   .set   128    ; W(z) filter size
K_mu             .set   0h     ; initial step constant
K_HOST_FLAG      .set   1      ; Enable EVM_HOST interface
K_DEFAULT_AC01   .set   1h     ; default AC01 init
*  This include file sets the FFT size for the ’54x Real FFT code
*  Note that the Real FFT size (i.e. the number of points in the
*  original real input sequence) is 2N; whereas the FFT size is
*  the number of complex points formed by packing the real inputs,
*  which is N.  For example, for a 256-pt Real FFT, K_FFT_SIZE
*  should be set to 128 and K_LOGN should be set to 7.
K_FFT_SIZE       .set   128    ; # of complex points (=N)
K_LOGN           .set   7      ; # of stages (=logN/log2)
K_ZERO_BK        .set   0
K_TWID_TBL_SIZE  .set   128    ; Twiddle table size
K_DATA_IDX_1     .set   2      ; Data index for Stage 1
K_DATA_IDX_2     .set   4      ; Data index for Stage 2
K_DATA_IDX_3     .set   8      ; Data index for Stage 3
K_FLY_COUNT_3    .set   4      ; Butterfly counter for Stage 3
K_TWID_IDX_3     .set   32     ; Twiddle index for Stage 3
*******************************************************************************
*       FILENAME: INIT54x.INC
*  This include file contains all the initial values of ST0, ST1, PMST, SWWSR, BSCR registers
;ST0 Register Organization
*      ---------------------------------------------------------------
*      |15       13 |  12 |  11 |  10 |  9  |  8         0|
*      ---------------------------------------------------------------
*      |     ARP     |  TC |  C  |  OVA|  OVB|      DP      |
*      ---------------------------------------------------------------
*******************************************************************************
K_ARP            .set   000b<<13     ; ARP can be addressed from 000b –111b
                                     ; reset value
K_TC             .set   1b<<12       ; TC = 1 at reset
K_C              .set   1b<<11       ; C = 1 at reset
K_OVA            .set   1b<<10       ; OVA = 0 at reset, Set OVA
K_OVB            .set   1b<< 9       ; OVB = 0 at reset, Set OVB
K_DP             .set   000000000b<<0; DP is cleared to 0 at reset
K_ST0            .set   K_ARP|K_TC|K_C|K_OVA|K_OVB|K_DP
*****************************************************************
```

Example 2–2. Processor Initialization — init_54x.asm (Continued)

```
;ST1 Register Organization
*  ------------------------------------------------------------------
*  | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 |  6 |  5 | 4        0 |
*  ------------------------------------------------------------------
*  |BRAF|CPL | XF | HM |INTM| 0  |OVM|SXM|C16|FRCT|CMPT|     ASM    |
*  ------------------------------------------------------------------
*********************************************************************************
K_BRAF        .set   0b << 15  ; BRAF = 0 at reset
K_CPL         .set   0b << 14  ; CPL = 0 at reset
K_XF          .set   1b << 13  ; XF = 1 at reset
K_HM          .set   0b << 12  ; HM = 0 at reset
K_INTM        .set   1b << 11  ; INTM
K_ST1_RESR    .set   0b << 10  ; reserved
K_OVM         .set   1b << 9   ; OVM = 0 at reset
K_SXM         .set   1b << 8   ; SXM = 1 at reset
K_C16         .set   0b << 07  ; C16 = 0 at reset
K_FRCT        .set   1b << 06  ; FRCT = 0 at reset,
                               ;  Set FRCT
K_CMPT        .set   0b << 05  ; CMPT = 0 at reset
K_ASM         .set   00000b << 00  ; ASM = 0 at reset
K_ST1_HIGH    .set   K_BRAF|K_CPL|K_XF|K_HM|K_INTM|K_ST1_RESR|K_OVM|K_SXMK_ST1_LOW
.set   K_C16|K_FRCT|K_CMPT|K_ASM
K_ST1         .set   K_ST1_HIGH|K_ST1_LOW
*********************************************************************************
*
*PMST Register Organization
*      ---------------------------------------------------------
*      |15       7  |  6  |  5  |  4  |  3  |  2  |  1      0 |
*      ---------------------------------------------------------
*      |     IPTR    |MP/MC |  OVLY | AVIS | DROM |CLKOFF| Reserved |
*      ---------------------------------------------------------
*********************************************************************************
K_IPTR        .set   111111111b << 07    ; 111111111b at reset
K_MP_MC       .set   1b << 06            ; 1 at reset
K_OVLY        .set   0b << 05            ; OVLY = 0 at reset
K_AVIS        .set   0b << 04            ; AVIS = 0 at reset
K_DROM        .set   0b << 03            ; DROM = 0 at reset
K_CLKOFF      .set   0b << 02            ; CLKOFF = 0 at reset
K_PMST_RESR   .set   00b << 0            ; reserved
                                         ; for 548  bit 0 = SMUL
                                         ;saturation on multiply
                                         ; bit 1 = SST = saturation on store
K_PMST.set    K_IPTR|K_MP_MC|K_OVLY|K_AVIS|K_DROM|K_CLKOFF|K_PMST_RESR
*********************************************************************************
```

Example 2–2. Processor Initialization — init_54x.asm (Continued)

```
*SWWSR Register Organization
*      ---------------------------------------------------------
*      |      15     |14    12|11     9|8     6| 5    3 |  2     0  |
*      ---------------------------------------------------------
*      |  Reserved  |   I/O  |   Data   |  Data |  Program|  Program |
*      ---------------------------------------------------------
*******************************************************************************
K_SWWSR_IO   .set   2000h     ; set the I/O space
*******************************************************************************
*Bank Switching Control Register (BSCR)Organization
*      ------------------------------------------
*      |15       12 |   11   | 10     2   |  1   |   0 |
*      ------------------------------------------
*      |   BNKCMP   | PS–DS | Reserved |  BH  | EXIO |
*      ------------------------------------------
*******************************************************************************
K_BNKCMP      .set   0000b << 12      ; bank size = 64K
K_PS_DS       .set   0b << 11
K_BSCR_RESR   .set   000000000b <<2   ; reserved space
K_BH          .set   0b << 1          ; BH = 0 at reset
K_EXIO        .set   0b << 0          ; EXIO = 0 at reset
K_BSCR        .set   K_BNKCMP|K_PS_DS|K_BSCR_RESR|K_BH|K_EXIO
; TEXAS INSTRUMENTS INCORPORATED
   .mmregs
   .include  ”init_54x.inc”
   .def      init_54
;-------------------------------------------------------------------------------
;  Functional Description
;  Initializes the processor from a reset state
;-------------------------------------------------------------------------------
   .sect     ”main_prg”
init_54:     ; Init.the s/w wait state reg.for 2 wait states for I/O operations
   STM    #K_SWWSR_IO, SWWSR      ; 2 wait states for I/Ooperations
                                  ; wait states for Bank Switch
   STM    #K_BSCR, BSCR           ; 0 wait states for BANK SWITCH
; initialize the status and control registers
   STM    #K_ST0, ST0
   STM    #K_ST1, ST1
   RETD
   STM    #K_PMST,PMST
   .end
```

Chapter 3

Analog I/O

Most DSP systems transfer data through peripherals. These peripherals include parallel and serial ports. This chapter describes how the serial ports are initialized and how the TLC320AC01 ('AC01) analog interface circuit (AIC) interfaces to the '54x serial port. This chapter also describes the various issues involved such as stack, context switching, interrupt priorities, and different addressing modes for collecting the samples during the interrupt processing.

Topic **Page**

3.1 Synchronous Serial Port Devices

Several ’54x devices implement a variety of types of flexible serial port interfaces. These serial port interfaces provide full duplex, bidirectional, communication with serial devices such as codecs, serial analog to digital (A/D) converters, and other serial systems. The serial port interface signals are directly compatible with many industry-standard codecs and other serial devices. The serial port may also be used for interprocessor communication in multiprocessing applications. When the serial ports are in reset, the device can be configured to shut off the serial port clocks. This allows the device to run in a low-power mode of operation.

Three signals are necessary to connect the ’54x to the serial port, as shown in Figure 3–1. On the transmitting device, the transmit data signal (DX) sends the data, the transmit frame synchronization signal (FSX) initiates the transfer at the beginning of the packet, and the transmit clock signal (CLKX) clocks the bit transfer.

The corresponding pins on the receiving device are the received serial data signal (DR), the receive frame synchronization signal (FSR) and, the receive clock signal (CLKR), respectively. At reset, CLKX, CLKR, DR, FSX, and FSR become inputs and DX is set for high impedance. Figure 3–1 shows the ’54x interface to an ’AC01 that uses the serial port to transfer the data to and from the DSP. The SCLK signal clocks the serial data into and out of the device.

Figure 3–1. Interfacing a TLC320AC01C AIC to the ’54x

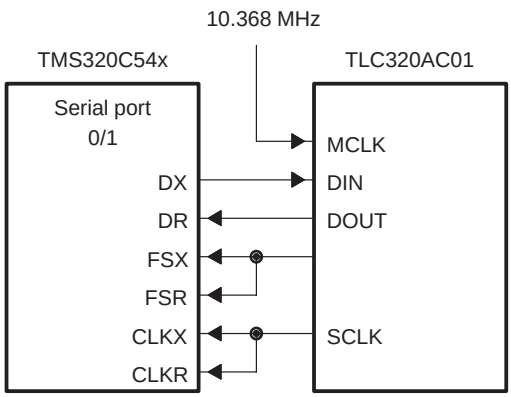
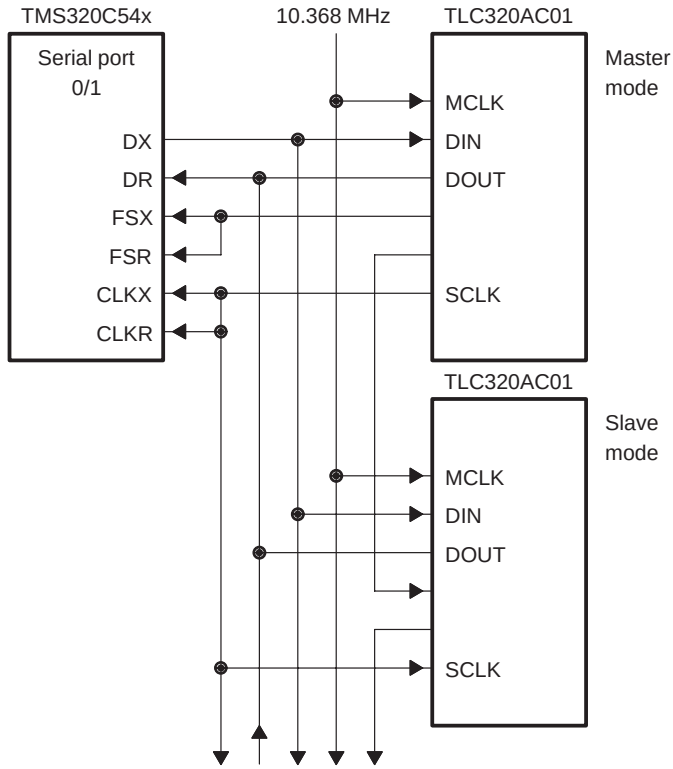


Figure 3–2 shows the master and slave-to-DSP interface. In the master-slave mode, the master ’AC01, generates its own shift clock and frame sync signals and generates all delayed frame-sync signals to support slave devices. The slave receives the shift clock and frame-sync signals from the master device.

Figure 3–2. Master- and Slave-to-’54x Interfaces



The serial port operates through three memory-mapped registers: the serial port control register (SPC), the data transmit register (DXR), and the data receive register (DRR). The configuration of the serial port is changed by writing to the SPC register. The system configuration is as follows:

- Word length: The serial port must be programmed to use 16-bit data, since communications between the ’AC01 AIC and the host DSP use 16 bits. This means clearing the format bit, FO, to a logic 0.
- Frame synchronization: The system operates in burst mode with frame synchronization signals generated externally by the ’AC01. This means the frame synchronization mode bit, FSM, must be set to a logic 1. A frame sync pulse is required on the frame sync transmit and receive pins (FSX and FSR, respectively) for the transmission and reception of every word.

It also means that the FSX pin has to be configured as an input pin by clearing the transmit mode bit, TXM, to a logic 0.

- Clock: The clock is generated externally by the ’AC01 and is supplied to the DSP on the CLKX pin. This means clearing the clock mode bit, MCM, to a logic 0.

To reset the serial port, two writes to the SPC register must occur. The first must write 0s to the transmit reset (XRST) and receive reset (RRST) bits and the desired configuration to all the other bits. The second must write 1s to the XRST and RRST bits and write the desired configuration to all the other bits. The SPC register is written as 0000 0000 1100 1000 to pull the serial port out of reset. This is shown in Example 3–1 on page 3-7.

3.2 TLC320AC01 Analog Interface Circuit

The ’AC01 is an audio-band processor. It integrates a band-pass switched-capacitor antialiasing input filter, 14-bit A/D and D/A converters, a low-pass switched-capacitor output reconstruction filter, sinx/x compensation, and a serial port for data and control transfers. The A/D and D/A channels operate synchronously, and data is transferred in 2s-complement format. It has nine internal registers that can be programmed for an application.

There are three basic modes of operation:

1) Stand-alone analog interface mode: The ’AC01 generates the shift clock and frame synchronization signals for data transfers; it is the only AIC used.

2) Master-slave mode: One ’AC01 serves as the master, generating the master shift clock and frame synchronization signals; the other ’AC01s serve as slaves.

3) Linear codec mode: The shift clock and frame synchronization signals are externally generated by the host.

Software control of the AIC allows you to program the sampling frequency, cutoff frequencies for the low- and high-pass filters, and analog input and output gains at any time. No programming is needed when the default values of the ’AC01 are satisfactory.

Data transfers between the DSP and the ’AC01 are categorized as primary and secondary serial communications. During primary communication, the 14 most significant bits (MSBs), bits 2 through 15, specify sample data. If the two least significant bits (LSBs), bits 1 and 0, are 11, a subsequent 16 bits of control information is received by the ’AC01 as part of the secondary serial communication interval. This control information programs one of the nine internal registers. During the secondary communication interval, the bits transmitted to the ’AC01 are defined as follows:

- Bits 15 and 14, which control phase shifting in certain applications, are usually 0s.
- Bit 13 decides whether the data is written to or read from a register.
- Bits 12 through 8 define the address of the register to be programmed.
- Bits 7 through 0 contain the data to be stored in a register for write operations.

The bits received from the ’AC01 during the secondary communication interval are 0s for bits 15 through 8, and the value of the register data for bits 7 through 0.

The three programs that follow show the use of the ’AC01 in different circumstances: Example 3–1 assumes that the user is satisfied with the default configuration of the ’AC01 and proceeds to receive and send data, Example 3–2 on page 3-12 shows how to program the ’AC01 for a particular configuration before data transfer, and Example 3–3 on page 3-17 demonstrates serial communications while in a master-slave configuration. All three programs consist of a main routine which, apart from initializing the serial port and the interrupt, also resets the ’AC01.

Example 3–1 initiates a reset of the ’AC01, initializes the serial port, and sets up interrupts. To reset the ’AC01 properly, it must be held in reset for at least one MCLK cycle. With the ’AC01 operating at 10.368 MHz, one MCLK cycle equals 96.45 ns. In all the examples, the total time spent between initiating a reset and pulling the ’AC01 out of reset is 12 cycles. For a ’54x DSP operating at 40 MHz, one cycle is 25 ns. This means that the ’AC01 spends 300 ns in reset. This is roughly three times the specification’s recommended value and is safe, since it allows a proper reset.

If a 50-MHz DSP is used, one cycle equals 20 ns; the same code keeps the ’AC01 in reset for 240 ns. This is also safe, since it is approximately 2.5 times the specification’s recommended value.

The ’AC01 can operate at a minimum clock frequency of 5.184 MHz. This implies that it must be held in reset for at least one MCLK cycle, or 192.9 ns. Assuming a processor speed of 50 MHz, the ’AC01 spends 12 cycles multiplied by 20 ns = 240 ns in reset.

A safe margin holds the ’AC01 in reset for at least twice the specification’s recommended time. To allow the necessary time in reset, use a dedicated timer function or insert useful instructions to fill the time slot. After the ’AC01 is reset, the main program sits in a loop, waiting for a serial port interrupt.

Example 3–1 initializes the serial port and uses the default initialization of the ’AC01, which is in the master mode and transfers the data upon a receive interrupt. It is not necessary to initiate a secondary communication interval to program the ’AC01. The serial port is configured between the cycles that the AIC is in reset.

Example 3–1. Default Initialization of ’AC01

```
;TEXAS INSTRUMENTS INCORPORATED
   .mmregs
   .include     ”interrpt.inc”
   .include     init_ser.inc”
AIC_VAR_DP      .usect ”aic_vars”,0
aic_in_rst      .usect ”aic_vars”,1
aic_out_of_rst  .usect ”aic_vars”,1
   .def         serial_init
;-------------------------------------------------------------------------------
;  Functional Description
;  This routine initializes the serial port 1 of 541. The serial port is put
;  in reset by writing 0’s to RRST and XRST bits and pulled out of reset by
;  writing 1’s to both RRST and XRST bits. This routine also puts the AC01
;  in reset and after 12 cycles the AC01 is pulled out of reset. The serial
;  port initialization is done during the 12 cycle latency of the AC01 init.
;
;-------------------------------------------------------------------------------
   .sect        ”ser_cnfg”
serial_init:
   LD           #AIC_VAR_DP,DP             ; initialize DP for aic_reset
   ST           #K_0, aic_in_rst           ; bit 15 = 0 of TCR resets AIC
   PORTW        aic_in_rst,K_TRGCR_ADDR    ;write to address 14h (TCR)
*******************************************************************************
*We need at least 12 cycles to pull the AIC out of reset.
*******************************************************************************
   STM          #K_SERIAL_RST, SPC1        ;reset the serial port with
                                           ;0000 0000 0000 1000
   STM          #K_SERIAL_OUT_RST, SPC1    ;bring ser.port out of reset
                                           ;0000 0000 1100 1000
   RSBX         INTM
   LD           #0,DP
   ORM          #(K_RINT1|K_INT1),IMR      ; Enable RINT1,INT1
                                           ; 0000 0000 0100 0010
   LD           #AIC_VAR_DP,DP             ; restore DP
   STM          #(K_RINT1),IFR             ; clear RINT1
   STM          #K_0,DXR1                  ; 0000 0000 0100 0000
; Pull the AC01 out of reset – the AC01 requires that it be held in reset for
; 1 MCLK, which is equivalent to 96.45ns (based on an MCLK of 10.368MHz)
   ST           #K_8000, aic_out_of_rst    ; bit 15 = 1 brings AIC from reset
   RETD
   PORTW        aic_out_of_rst, K_TRGCR_ADDR  ;AIC out of reset
   .end
* This include file includes the SPC1 register configuration
*******************************************************************************
```

Example 3–1. Default Initialization of ’AC01 (Continued)

```
* FILENAME: “INIT_SER.INC”
; SPC Register Organization
*  ------------------------------------------------------
*  | 15  | 14  |   13   |   12    | 11   |  10  | 9  |  8  |
*  |-----|-----|--------|---------|------|------|----|-----|
*  |FREE | SOFT| RSRFULL| XSREMPTY| XRDY | RRDY | IN1| IN0 |
*  ------------------------------------------------------
*  ---------------------------------------------
*  | 7   | 6   |   5 | 4  |  3  |  2 | 1   |  0  |
*  |-----|-----|-----|----|-----|----|-----|-----|
*  |RRST | XRST| TXM | MCM| FSM | FO | DLB | RES |
*  ---------------------------------------------
* This include file includes the SPC1 register configuration
*************************************************************************
```

;Bit	Name	Function
;0	Reserved	Always read as 0
;1	DLB	Digital loop back : 0 –> Disabled, 1 _. Enabled
;2	FO	Format bit: 0 – > data transferred as 8 bit bytes, 1 –> 16 bit words
;3	FSM	Frame sync pulse: 0 –> serial port in continuous mode, 1 –> FSM is required
;4	MCM	Clock mode bit: 0 –> CLKX obtained from CLKX pin 1–> CLKX obtained from CLKX
;5	TXM	Transmit mode bit: 0 –> Frame sync pulses generated externally and supplied on FSX pin, 1–> Internally generated frame sync pulses out on FSX pin
;6	XRST	Transmit reset bit: 0 –> reset the serial port, 1–> bring serial port out of reset
;7	RRST	Receive reset bit: 0 –> reset the serial port, 1–> bring serial port out of reset
;8	IN0	Read–only bit reflecting the state of the CLKR pin
;9	IN1	Read–only bit reflecting the state of the CLKX pin
;10	RRDY	Transition from 0 to 1 indicates data is ready to be read
;11	XRDY	Transition from 0 to 1 indicates data is ready to be sent
;12	XSREMPTY	Transmit shift register empty (Read–only) 0 –> transistor has experienced underflow, 1–> has not experienced underflow
;13	RSRFUL	Receive shift register full flag (Read–only): 0 –> Receiver has experienced overrun, 1–> receiver has not experienced overrun
;14	SOFT	Soft bit – 0 –> immediate stop, 1–> stop after word completion
;15	FREE	Free run bit: 0 –> behavior depends on SOFT bit, 1–> free run regardless of SOFT bit

Example 3–1.Default Initialization of ’AC01 (Continued)

```
; The system has the following configuration:
;  Uses 16-bit data => FO = 0
;  Operates in burst mode => FSM = 1
;  CLKX is derived from CLKX pin => MCM = 0
;  Frame sync pulses are generated externally by the AIC => TXM = 0
; Therefore, to reset the serial port, the SPC field would have
;  0000 0000 0000 1000
; To pull the serial port out of reset, the SPC field would have
;  0000 0000 1100 1000
K_0              .set   00000000b << 8          ; bits 15-8 to 0 at reset
K_RRST           .set   0b << 7                 ; First write to SPC1 is 0
                                                ; second write is 1
K_XRST           .set   0b << 6                 ; First write to SPC1 is 0
                                                ; second write is 1
K_TXM            .set   0b << 5
K_MCM            .set   0b << 4
K_FSM            .set   1b << 3                 ; Frame Sync mode
K_ZERO           .set   000b << 0
K_SERIAL_RST     .set   K_0|K_RRST|K_XRST|K_TXM|K_MCM|K_FSM|K_ZERO
                                                ; first write to SPC1 register
K_RRST1          .set   1b << 7                 ; second write to SPC1
K_XRST1          .set   1b << 6                 ; second write to SPC1
K_SERIAL_OUT_RST .set   K_0|K_RRST1|K_XRST1|K_TXM|K_MCM|K_FSM|K_ZERO
K_TRGCR_ADDR     .set    14h                    ; Target/Status I/O address
K_0              .set    0h
K_8000           .set    8000h                  ; set bit 15 to pull AIC out
                                                ; of reset
* FILENAME: INTERRUPT.INC
; 541 Interrupt Mask Register (IMR) Organization
*  ---------------------------------------------------------------------
*  | 15     9 |  8  |  7   |   6  |  5   |  4   | 3   | 2   | 1   |  0  |
*  |----------|-----|------|------|------|------|-----|-----|-----|-----|
*  | Reserved | INT3| XINT1| RINT1| XINT0| RINT0| TINT| INT2| INT1| INT0|
*  ---------------------------------------------------------------------
* This file includes the IMR and IFR configuration
*******************************************************************************
K_IMR_RESR    .set      0000000b << 9          ; reserved space
K_INT3        .set      1b << 8                ; disable INT3
K_XINT1       .set      1b << 7                ; disable transmit interrupt 1
K_RINT1       .set      1b << 6                ; enable receive interrupt 1
K_XINT0       .set      1b << 5                ; disable transmit interrupt 0
K_RINT0       .set      1b << 4                ; disable receive interrupt
K_TINT        .set      1b << 3                ; disable timer interrupt
K_INT2        .set      1b << 2                ; disable INT2
K_INT1        .set      1b << 1                ; disable INT1
K_INT0        .set      1b << 1                ; enable INT0
```

Example 3–2 on page 3-12 demonstrates initiation of a secondary communication interval for programming the ’AC01. The configuration of the ’AC01 can be changed via its nine internal registers. To understand this, consider registers 1 and 2 in the ’AC01.

The data content of register 1, also referred to as the register A, determines the divisions of the master clock frequency that produce the internal filter clock, FCLK. The default value of this register is 18 (decimal). The filter clock frequency is calculated using the equation:

FCLK frequency = master clock frequency / (register A contents $\times$ 2)

This means that for an MCLK of 10.368 MHz, the default value of the filter clock frequency is

FCLK = 288 kHz

This FCLK determines the –3 dB corner frequency of the low-pass filter which is given by:

$$f_{LP} = FCLK/40$$

The default value of f_{LP} is equal to 7.2 kHz.

The data content of register 2 of the ’AC01, also referred to as register B, determines the division of FCLK to generate the conversion clock, and is given by the equation:

Conversion frequency = FCLK/(register B contents)

The default value of register B is equal to 18 (decimal). Hence, the default value of the conversion (sampling) frequency is equal to:

288 kHz /18 = 16 kHz

This register also determines the –3 dB corner frequency of the high-pass filter, which is given by:

$$f_{HP} = \text{sampling frequency}/200$$

Hence, the default value of f_{HP} is equal to 80 Hz.

For a system that processes speech signals, the following parameters are desirable:

Sampling frequency f_S = 8.0 kHz

Low-pass filter corner frequency, f_{LP} = 3.6 kHz

Assume that the ’AC01 uses a master clock frequency, MCLK, of 10.368 MHz. The ’AC01’s parameters are:

$$f_{LP} = FCLK/40 \text{ gives } FCLK = 144 \text{ kHz}$$

$$FCLK = MCLK/(\text{register A} \times 2) \text{ gives register A} = 36 \text{ (decimal)}$$

$$f_S = FCLK/B \text{ gives register B} = 18 \text{ (decimal)}$$

This also means that with the specified sampling frequency, the –3 dB corner frequency of the high-pass filter changes to 8.0 kHz / 200 = 48 Hz.

After the ’AC01 has been reset in the main program, it makes a call to the routine wrt_cnfg, which programs the new values of registers A and B as calculated above. To prevent the occurrence of an interrupt while programming the ’AC01, wrt_cnfg disables all interrupts until the end of programming. After each word has been sent to the serial port, the code waits for the data to be copied from the data transmit register to the transmit shift register before it sends the next data. After the ’AC01 has been programmed, the main routine waits for an interrupt. The service routines that transfer data between the memory buffers and the serial port transmit and receive registers remain the same.

Example 3–2. Initiation of a Secondary Communication Interval for Programming the ’AC01

```
;TEXAS INSTRUMENTS INCORPORATED
   .mmregs
   .include     ”interrpt.inc”
   .include     ”init_ser.inc”
AIC_VAR_DP      .usect ”aic_vars”,0
aic_in_rst      .usect ”aic_vars”,1
aic_out_of_rst  .usect ”aic_vars”,1
   .def   serial_init
;-------------------------------------------------------------------------------
; Functional Description
;  This routine initializes the serial port 1 of 541. The serial port is put
;  in reset by writing 0’s to RRST and XRST bits and pulled out of reset by
;  writing 1’s to both RRST and XRST bits. This routine also puts the AC01
;  in reset and after 12 cycles the AC01 is pulled out of reset. The serial
;  port initialization is done during the 12 cycle latency of the AC01 init.
; ;-----------------------------------------------------------------------
   .sect  ”ser_cnfg”
serial_init:
   LD     #AIC_VAR_DP,DP               ; initialize DP for aic_reset
   ST     #K_0, aic_in_rst             ; bit 15 = 0 of TCR resets AIC
   PORTW  aic_in_rst,K_TRGCR_ADDR      ;write to address 14h (TCR)
*******************************************************************************
*We need at least 12 cycles to pull the AIC out of reset.
*******************************************************************************
   STM    #K_SERIAL_RST, SPC1          ;reset the serial port with
                                       ;0000 0000 0000 1000
   STM    #K_SERIAL_OUT_RST, SPC1      ;bring ser.port out of reset
                                       ;0000 0000 1100 1000
   RSBX   INTM
   LD     #0,DP
   ORM    #(K_RINT1|K_INT1),IMR        ; Enable RINT1,INT1
                                       ; 0000 0000 0100 0010
   LD     #AIC_VAR_DP,DP               ; restore DP
   STM    #(K_RINT1),IFR               ; clear RINT1
   STM    #K_0,DXR1                    ; 0000 0000 0100 0000
; Pull the AC01 out of reset – the AC01 requires that it be held in reset for
; 1 MCLK, which is equivalent to 96.45ns (based on an MCLK of 10.368MHz)
   ST     #K_8000, aic_out_of_rst      ; bit 15 = 1 brings AIC from reset
   RETD
   PORTW  aic_out_of_rst, K_TRGCR_ADDR  ;AIC out of reset
   .end
; TEXAS INSTRUMENTS INCORPORATED
   .mmregs
   .include     ”interrpt.inc”
   .ref   wrt_cnfg                     ; initializes AC01
   .def   aic_init
;-------------------------------------------------------------------------------
;  Functional Description
;  This routine disables IMR and clears any pending interrupts before
;  initializing AC01. The wrt_cnfg function configures the AC01
;-------------------------------------------------------------------------------
   .sect  ”aic_cnfg”
```

Example 3–2. Initiation of a Secondary Communication Interval for Programming the 'AC01(Continued)

```
aic_init:
   CALLD  wrt_cnfg  ; initialize AC01
   ANDM   #(~K_RINT1|K_INT1),IMR ; disable receive_int1,INT1
   ORM    #(K_RINT1|K_INT1),IMR  ; enable the RINT1, INT1
   RETD
   STM    #(K_RINT1),IFR   ; service any pending interrupt
   .end
*************************************************************************
* This file includes the AC01 registers initialization
* All registers have 2 control bits that initiates serial communication
* There are 2 communication modes – primary and secondary communications
* During primary communication the control bits D00 and D01 are 11 to request
* for a secondary communication. In the secondary serial communications the
* control bits D15 and D14 perform same control function as primary.
* The R/W~ bit at reset is set to 0 placing the device in write mode.
*************************************************************************
K_NOP_ADDR       .set   0 << 8
K_REG_0          .set   K_NOP_ADDR
K_A_ADDR         .set   1 << 8              ; REG 1 address
K_A_REG          .set   36
K_REG_1          .set   K_A_ADDR|K_A_REG    ; FCLK = 144KHz => A =24h
K_B_ADDR         .set   2 << 8              ; REG 2 address
K_B_REG          .set   18
K_REG_2          .set   K_B_ADDR|K_B_REG    ; Sampling rate = 8KHz
K_AA_ADDR        .set   3 << 8              ; Register 3 address
K_AA_REG         .set   0
K_REG_3          .set   K_AA_ADDR|K_AA_REG  ; ; no shift
K_GAIN_ADDR      .set   4 << 8              ; Register 4 address
K_MONITOR_GAIN   .set   00b << 4            ; Monitor output gain = squelch
K_ANLG_IN_GAIN   .set   01b << 2            ; Analog input gain = 0dB
K_ANLG_OUT_GAIN  .set   01b << 0            ; Analog output gain = 0dB
K_REG_4          .set   K_GAIN_ADDR|K_MONITOR_GAIN|K_ANLG_IN_GAIN|K_ANLG_OUT_GAIN
K_ANLG_CNF_ADDR  .set   5 << 8              ; Register 5 address
K_ANLG_RESRV     .set   0 << 3              ; Must be set to 0K_HGH_FILTER
                 .set   0 << 2              ; High pass filter is enabled
K_ENBL_IN        .set   01b << 0            ; Enables IN+ and IN–
K_REG_5          .set   K_ANLG_CNF_ADDR|K_ANLG_RESRV|K_HGH_FILTER|K_ENBL_IN
K_DGTL_CNF_ADDR  .set   6 << 8              ; Register 6 address
K_ADC_DAC        .set   0 << 5              ; ADC and DAC is inactive
K_FSD_OUT        .set   0 << 4              ; Enabled FSD output
K_16_BIT_COMM    .set   0 << 3              ; Normal 16–bit mode
K_SECND_COMM     .set   0 << 2              ; Normal secondary communication
K_SOFT_RESET     .set   0 << 1              ; Inactive reset
K_POWER_DWN      .set   0 << 0              ; Power down external
K_REG_HIGH_6     .set   K_DGTL_CNF_ADDR|K_ADC_DAC|K_FSD_OUT|K_16_BIT_COMM
K_REG_LOW_6      .set   K_SECND_COMM|K_SOFT_RESET|K_POWER_DWN
K_REG_6          .set   K_REG_HIGH_6|K_REG_LOW_6
K_FRME_SYN_ADDR  .set   7 << 8              ; Register 7 address
K_FRME_SYN       .set   0 << 8              ;
K_REG_7          .set   K_FRME_SYN_ADDR|K_FRME_SYN
K_FRME_NUM_ADDR  .set   8 << 8              ; Register 8 address
K_FRME_NUM       .set   0 << 8              ;
```

Example 3–2. Initiation of a Secondary Communication Interval for Programming the ’AC01(Continued)

```
K_REG_8          .set   K_FRME_NUM_ADDR|K_FRME_NUM
; primary word with D01 and D00 bits set to 11 will cause a
; secondary communications interval to start when the frame
; sync goes low next
K_SCND_CONTRL.set   11b << 0  ; Secondary comm.bits
AIC_REG_START_LIST  .sect   ”aic_reg”   ; includes the aic table
    .word  AIC_REG_END_LIST–AIC_REG_START_LIST–1
    .word  K_REG_1
    .word  K_REG_2
    .word  K_REG_3
    .word  K_REG_4
    .word  K_REG_5
    .word  K_REG_6
    .word  K_REG_7
    .word  K_REG_8
AIC_REG_END_LIST
K_XRDY    .set        0800h   ; XRDY bit in SPC1
; TEXAS INSTRUMENTS INCORPORATED
    .mmregs
    .include  ”interrpt.inc”
    .ref   wrt_cnfg  ; initializes AC01
    .def   aic_init
;-------------------------------------------------------------------------------
;  Functional Description
;  This routine disables IMR and clears any pending interrupts before
;  initializing AC01. The wrt_cnfg function configures the AC01
;-------------------------------------------------------------------------------
    .sect  ”aic_cnfg”
aic_init:
    CALLD  wrt_cnfg               ; initialize AC01
    ANDM   #(~K_RINT1),IMR        ; disable receive_int1
    ORM    #(K_RINT1|K_INT1),IMR  ; enable the RINT1, INT1
    RETD
    STM    #(K_RINT1),IFR         ; service any pending interrupt
    .end
; TEXAS INSTRUMENTS INCORPORATED
    .mmregs
    .include  ”aic_cfg.inc”
aic_reg_tble .usect  ”aic_vars”,10
    .def   wrt_cnfg
;
;  Functional Description
;         Writes new configuration data into the AC01. Assuming a system
;         which processes speech signals and *  requires the following parameters
;         Low pass filter cut–off frequency = 3.6 kHz
;         Sampling rate = 8000 Hz
;         Assume the Master clock MCLK = 10.368 MHz
;         This example demonstrates how to program these parameters –
;         the registers affected are:
;         Register A which determines the division of the MCLK frequency
;         to generate the internal filter clock FCLK.
;         It also determines the –3 dB corner frequency of the low–pass filter
```

Example 3–2. Initiation of a Secondary Communication Interval for Programming the 'AC01(Continued)

```
;         Register B  which determines the division of FCLK to generate
;         the sampling (conversion) frequency
;         It also determines the –3dB corner frequency of the high–pass filter
;-------------------------------------------------------------------------------
    .asg   AR1,AIC_REG_P
    .sect  "aic_cnfg"
wrt_cnfg:
    STM    #aic_reg_tble,AIC_REG_P          ; init AR1
    RPT    #AIC_REG_END_LIST–AIC_REG_START_LIST
    MVPD   #AIC_REG_START_LIST,*AIC_REG_P+  ; move the table
    STM    #aic_reg_tble,AIC_REG_P          ; init AR1
    STM    #K_REG_0,DXR1                    ; primary data word –
                                            ; a jump start!
wait_xrdy
    BITF   SPC1,K_XRDY                      ;test XRDY bit in SPC1
    BC     wait_xrdy,NTC                    ;loop if not set
    STM    #K_SCND_CONTRL,DXR1              ;send primary word with
                                            ; D01–D00  = 11 to
                                            ;signify secondary communication
    LD     *AIC_REG_P+,A
    STLM   A,BRC                            ; gives the # of registers to be
    NOP
                                            ; initialized
    RPTB   aic_cfg_complte–1
wait_xrdy1
    BITF   SPC1,K_XRDY                      ;test XRDY bit in SPC1
    BC     wait_xrdy1,NTC                   ;loop if not set
    LD     *AIC_REG_P+,A                    ; Read the register contents
    STLM   A, DXR1
wait_xrdy2
    BITF   SPC1,K_XRDY                      ;test XRDY bit in SPC1
    BC     wait_xrdy2,NTC                   ;loop if not set
    STM    #K_SCND_CONTRL,DXR1              ; set to read the next register
aic_cfg_complte                             ; contents
wait_xrdy3
    BITF   SPC1,K_XRDY                      ;test XRDY bit in SPC1
    BC     wait_xrdy3,NTC                   ;loop if not set
    RET
```

Example 3–3 on page 3-17 uses two ’AC01s in master-slave configuration. After the ’AC01s have been reset, the Ac01s make a call to the routine wrt_cnfg. This routine programs the low- and high-pass filter cutoff frequencies, the number of slave devices attached, and sets up the frame-sync delay register. During the programming mode, the master and its slaves are programmed with the same control information. The last register programmed is frame-sync delay, which determines the lag between the frame-sync and delayed frame-sync signals. The minimum value for the frame-sync delay register is 18 (decimal). This is because every data transfer requires 16 clock periods, and the output data, DOUT, is placed in the high-impedance state on the 17th rising edge of the clock.

Example 3–3 uses a minimum value of 18, which means that the time delay between the frame sync and the delayed frame sync is equal to 18 multiplied by the period of SCLK. With SCLK = 2.592 MHz based on an MCLK of 10.368 MHz, the time delay is 6.94μs. The hardware schematic indicates that the delayed frame-sync signal from the master is the frame-sync signal for the slave. The master, therefore, generates a frame-sync signal for itself and a delayed frame-sync signal for its slave. It then synchronizes the host serial port for data transfers to itself and its slaves. All interrupts on the host are disabled during the time the ’AC01s are programmed. This disallows the host from receiving or sending any data to or from the ’AC01s.

The output data (DOUT) from the ’AC01 operating in the master mode is available at the falling edge of the master frame-sync signal. In the primary communication interval, this data represents the 14-bit A/D conversion result. The two least significant bits (LSBs), D01 and D00, determine whether the data is generated by the master or its slave as follows:

	D01	D00
Master mode	0	0
Slave mode	0	1

In the secondary communication interval, the data available at DOUT represents the contents of the register read with the eight MSBs set to 0 and the read/write bit set to logic 1 during the primary interval. If no register read is requested, the second word is all 0s.

The output data at DOUT from the ’AC01 operating in slave mode is available at either the falling edge of the external frame sync signal (which is the delayed frame sync from the master) or the rising edge of the external SCLK signal, whichever comes first.

When the AC01 receives an interrupt, the service routine first determines whether the received data is from the master or the slave by checking the LSBs of the received data at bit positions D01 and D00. If the LSBs are 00, the received data is stored in a memory buffer, which holds all data received from the master. The service routine also sends data from a memory buffer, which holds all data going the master. If the LSBs of the received data are 01, data is received from and sent to the slave 'AC01, using dedicated memory buffers. This can be used with double buffering or circular buffers to allow the ISR to notify the main routine. This occurs when new input data is available for processing and/or space is available to accept processed data for output to or from the master and the slave.

Example 3–3. Master-Slave Mode

```
*********************************************************************
* This file includes the AC01 registers initialization
* All registers have 2 control bits that initiates serial communication
* There are 2 communication modes – primary and secondary communications
* During primary communication the control bits D00 and D01 are 11 to request
* for a secondary communication. In the secondary serial communications the
* control bits D15 and D14 perform same control function as primary.
* The R/W~ bit at reset is set to 0 placing the device in write mode.
* DS7–DS00 –> Number of SCLK's between FS~ and FSD~. When slaves are used,
* this should be the last register to be programmed The minimum value for this
* register should be decimal 18. This means that the time interval between the
* FS~ and FSD~ with an SCLK * frequency of 2.592 MHz (with an MCLK of 10.368 MHz)
* is 18 1/2.592 MHz = 6.94 us
; DS7–DS00 –> # of frame sync signals generated by AC01
; Therefore, number of frame sync signals number of slave = devices + 1
; In programming the AC01 REG7 is the last register to be programmed if the
; configuration is master salve mode configuration
*********************************************************************
K_NOP_ADDR      .set   0 << 8
K_REG_0         .set   K_NOP_ADDR
K_A_ADDR        .set   1 << 8              ; REG 1 address
K_A_REG         .set   36
K_REG_1         .set   K_A_ADDR|K_A_REG    ; FCLK = 144KHz => A =24h
K_B_ADDR        .set   2 << 8              ; REG 2 address
K_B_REG         .set   18
K_REG_2         .set   K_B_ADDR|K_B_REG    ; Sampling rate = 8KHz
K_AA_ADDR       .set   3 << 8              ; Register 3 address
K_AA_REG        .set   0
K_REG_3         .set   K_AA_ADDR|K_AA_REG  ; no shift
K_GAIN_ADDR     .set   4 << 8              ; Register 4 address
K_MONITOR_GAIN  .set   00b << 4            ; Monitor output gain = squelch
K_ANLG_IN_GAIN  .set   01b << 2            ; Analog input gain = 0dB
K_ANLG_OUT_GAIN .set   01b << 0            ; Analog output gain = 0dB
K_REG_4         .set   K_GAIN_ADDR|K_MONITOR_GAIN|K_ANLG_IN_GAIN|K_ANLG_OUT_GAIN
K_ANLG_CNF_ADDR .set   5 << 8              ; Register 5 address
K_ANLG_RESRV    .set   0 << 3              ; Must be set to 0
K_HGH_FILTER    .set   0 << 2              ; High pass filter is enabled
K_ENBL_IN       .set   01b << 0            ; Enables IN+ and IN–
```

Example 3–3. Master-Slave Mode (Continued)

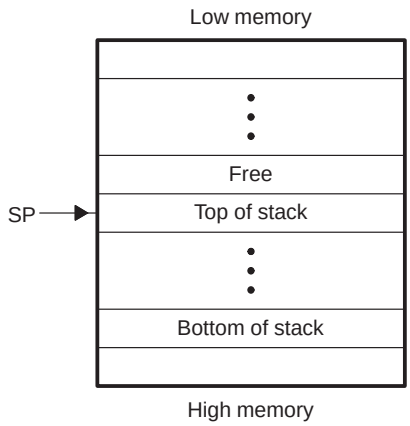
```
K_REG_5          .set   K_ANLG_CNF_ADDR|K_ANLG_RESRV|K_HGH_FILTER|K_ENBL_IN
K_DGTL_CNF_ADDR  .set   6 << 8              ; Register 6 address
K_ADC_DAC        .set   0 << 5              ; ADC and DAC is inactive
K_FSD_OUT        .set   0 << 4              ; Enabled FSD output
K_16_BIT_COMM    .set   0 << 3              ; Normal 16–bit mode
K_SECND_COMM     .set   0 << 2              ; Normal secondary communication
K_SOFT_RESET     .set   0 << 1              ; Inactive reset
K_POWER_DWN      .set   0 << 0              ; Power down external
K_REG_HIGH_6     .set   K_DGTL_CNF_ADDR|K_ADC_DAC|K_FSD_OUT|K_16_BIT_COMM
K_REG_LOW_6      .set   K_SECND_COMM|K_SOFT_RESET|K_POWER_DWN
K_REG_6          .set   K_REG_HIGH_6|K_REG_LOW_6
K_FRME_SYN_ADDR  .set   7 << 8              ; Register 7 address
K_FRME_SYN       .set   18 << 8             ;
K_REG_7          .set   K_FRME_SYN_ADDR|K_FRME_SYN
K_FRME_NUM_ADDR  .set   8 << 8              ; Register 8 address
K_FRME_NUM       .set   2 << 8              ;
K_REG_8          .set   K_FRME_NUM_ADDR|K_FRME_NUM
; primary word with D01 and D00 bits set to 11 will cause a
; secondary communications interval to start when the frame
; sync goes low next
K_SCND_CONTRL    .set   11b << 0            ; Secondary comm.bits
AIC_REG_START_LIST  .sect  ”aic_reg”        ; includes the aic table
    .word  AIC_REG_END_LIST–AIC_REG_START_LIST–1
    .word  K_REG_1
    .word  K_REG_2
    .word  K_REG_3
    .word  K_REG_4
    .word  K_REG_5
    .word  K_REG_6
    .word  K_REG_8
    .word  K_REG_7                          ; this should be the last
                                            ; register to be programmed
AIC_REG_END_LIST
K_XRDY           .set   0800h               ; XRDY bit in SPC1
```

3.3 Software Stack

A ’54x device has a software stack whose location is determined by the contents of the stack pointer (SP). When using the software stack, you must accommodate anticipated storage requirements. The system stack fills from high to low memory addresses, as shown in Figure 3–3. The SP always points to the last element pushed onto the stack. A push performs a predecrement and a pop performs a postincrement of the system stack pointer. The stack is used during subroutine calls and returns and inside the subroutine as temporary storage.

The CALL, CALLD, CC, CCD, CALA, and CALAD instructions push the value of the program counter (PC) onto the stack. The RET and RETD instructions pops the stack and places its value into the program counter. The contents of any memory-mapped register can be pushed onto and popped from the stack, using the PSHM and POPM instructions. Two additional instructions, PSHD, and POPD, are included in the instruction set so that data memory values can be pushed onto and popped from the stack.

Figure 3–3. System Stack



3.4 Context Switching

Before you execute a routine, you must save its context and restore the context after the routine has finished. This procedure is called context switching, and involves pushing the PC onto the stack. Context switching is useful for subroutine calls, especially when making extensive use of the auxiliary registers, accumulators, and other memory-mapped registers.

Due to system and CPU requirements, the order of saving and restoring can vary. Some repeat instructions, such as RPTB, are interruptible. To nest repeat block instructions, you must ensure that the block-repeat counter (BRC), block-repeat start address (RSA), and block-repeat end address (REA) registers are saved and restored.

You must also ensure that the block-repeat active flag (BRAF) is properly set. Since the block-repeat flag can be deactivated by clearing the BRAF bit of the ST1 register, the order in which you push the block-repeat counter and ST1 is important. If the BRC register is pushed onto the stack prior to ST1, any PC discontinuity in RPTB can give a wrong result, since BRAF is cleared in ST1. Thus, you must restore BRC before restoring the ST1 register.

A context save complements the restored contents. To ensure the integrity of the code, determine what contents must be restored so that no sequencing is lost.

Example 3–4. Context Save and Restore for TMS320C54x

```
      .title     “CONTEXT SAVE/RESTORE on SUBROUTINE or INTERRUPT
CONTEXT_RESTORE  .macro
      POPM  PMST     ;Restore PMST register
      POPM  RSA      ;Restore block repeat start address
      POPM  REA      ;Restore block repeat end address
      POPM  BRC      ;Restore block repeat counter
      POPM  IMR      ;Restore interrupt mask register
      POPM  BK       ;Restore circular size register
      POPM  ST1      ;Restore ST1
      POPM  ST0      ;Restore ST0
      POPM  AR0      ;Restore AR0
      POPM  AR1      ;Restore AR1
      POPM  AR2      ;Restore AR2
      POPM  AR3      ;Restore AR3
      POPM  AR4      ;Restore AR4
      POPM  AR5      ;Restore AR5
      POPM  AR6      ;Restore AR6
      POPM  AR7      ;Restore AR7
      POPM  T        ;Restore temporary register
      POPM  TRN      ;Restore transition register
      POPM  BL       ;Restore lower 16 bits of accB
      POPM  BH       ;Restore upper 16 bits of accB
      POPM  BG       ;Restore 8 guard bits of accB
      POPM  AL       ;Restore lower 16 bits of accA
      POPM  AH       ;Restore upper 16 bits of accA
      POPM  AG       ;Restore 8 guard bits of accA
      .endm
CONTEXT_SAVE     .macro
      PSHM  AG       ;Save 8 guard bits of accA
      PSHM  AH       ;Save upper 16 bits of accA
      PSHM  AL       ;Save lower 16 bits of accA
      PSHM  BG       ;Save 8 guard bits of accB
      PSHM  BH       ;Save upper 16 bits of accB
      PSHM  BL       ;Save lower 16 bits of accB
      PSHM  TRN      ;Save transition register
      PSHM  T        ;Save temporary register
      PSHM  AR7      ;Save AR7
      PSHM  AR6      ;Save AR6
      PSHM  AR5      ;Save AR5
      PSHM  AR4      ;Save AR4
      PSHM  AR3      ;Save AR3
      PSHM  AR2      ;Save AR2
      PSHM  AR1      ;Save AR1
      PSHM  AR0      ;Save AR0
      PSHM  ST0      ;Save ST0
      PSHM  ST1      ;Save ST1
      PSHM  BK       ;Save circular size register
      PSHM  IMR      ;Save interrupt mask register
      PSHM  BRC      ;Save block repeat counter
      PSHM  REA      ;Save block repeat end address
      PSHM  RSA      ;Save block repeat start address
      PSHM  PMST     ;Save PMST register
      .endm
```

3.5 Interrupt Handling

The ’54x CPU supports 16 user-maskable interrupts. The vectors for interrupts not used by a ’54x device can function as software interrupts, using the INTR and TRAP instructions. TRAP and INTR allow you to execute any of the 32 available ISRs. You can define other locations in the interrupt vector table. The INTR instruction sets the INTM bit to 1, clears the corresponding interrupt flag to 0, and makes the $\overline{\text{IACK}}$ signal active, but the TRAP instruction does not. INTR and TRAP are nonmaskable interrupts.

When a maskable interrupt occurs, the corresponding flag is set to 1 in the interrupt flag register (IFR). Interrupt processing begins if the corresponding bit in IMR register is set to 1 and the INTM bit in the ST1 register is cleared. The IFR register can be read and action taken if an interrupt occurs. This is true even when the interrupt is disabled. This is useful when not using an interrupt-driven interface, such as in a subroutine call when INT1 has not occurred.

When interrupt processing begins, the PC is pushed onto the stack and the interrupt vector is loaded into the PC. Interrupts are then disabled by setting INTM = 1. The program continues from the address loaded in the PC. Since all interrupts are disabled, the program can be processed without any interruptions, unless the ISR reenables them. Except for very simple ISRs, it is important to save the processor context during execution of the routine.

During the time the ’AC01 is reset, the DSP initializes the serial port and sets up the interrupt. To set up the interrupts, it performs the following operations:

- Enables unmasked interrupts by clearing the interrupt mode bit (INTM)
- Clears prior receive interrupts by writing the current contents of the appropriate receive interrupt flag in the IFR back to the IFR
- Enables receive interrupts by setting the appropriate receive interrupt flag in the interrupt mask register (IMR)

The initialization of the IMR and IFR registers and the INTM bit is included in the serial port and the ’AC01 initialization.

Example 3–5 processes the receive interrupt 1 service routine. The routine collects 256 samples in the first buffer and changes the address to the second buffer for the next 256 samples while processing the first buffer.

Example 3–5. Receive Interrupt Service Routine

```
; TEXAS INSTRUMENTS INCORPORATED
   .mmregs
   .include  ”INTERRPT.INC”
   .include  ”main.inc”
RCV_INT1_DP      .usect    ”rcv_vars”,0
d_index_count    .usect    ”rcv_vars”,1
d_rcv_in_ptr     .usect    ”rcv_vars”,1          ; save/restore input bffr ptr
d_xmt_out_ptr    .usect    ”rcv_vars”,1          ; save/restore output bffr ptr
d_frame_flag     .usect    ”rcv_vars”,1
input_data       .usect    ”inpt_buf”,K_FRAME_SIZE*2 ; input data array
output_data      .usect    ”outdata”,K_FRAME_SIZE*2  ; output data array
   .def          receive_int1
   .def          d_frame_flag
   .def          RCV_INT1_DP
   .def          input_data,output_data
   .def          d_xmt_out_ptr
   .def          d_rcv_in_ptr
;-------------------------------------------------------------------------------
;  Functional Description
;   This routine services receive interrupt1. Accumulator A, AR2 and AR3
;  are pushed onto the stack since AR2 and AR3 are used in other applications.
;  A 512 buffer size for both input and output.
;  After every 256 collection of input samples a flag is set to process the
;  data. No circular buffering scheme is used here.
;  After collecting 256 samples in the 1st bffr, then the second buffer
;  address is loaded and collect data in the second buffer while processing
;  the first buffer and vice versa.
;-------------------------------------------------------------------------------
   .asg          AR2,GETFRM_IN_P                 ; get frame input data pointer
   .asg          AR3,GETFRM_OUT_P                ; get frame output data pointer
   .asg          AR2,SAVE_RSTORE_AR2
   .asg          AR3,SAVE_RSTORE_AR3
   .sect         ”main_prg”
receive_int1:
       PSHM      AL
       PSHM      AH
       PSHM      AG
       PSHM      BL
       PSHM      BH
       PSHM      BG
; AR2, AR3 are used in other routines, they need to be saved and restored
; since receive interrupt uses AR2 and AR3 as pointers
       PSHM      SAVE_RSTORE_AR2                 ; Since AR2 and AR3 are used
       PSHM      SAVE_RSTORE_AR3                 ; in other routines, they need
       PSHM      BRC
       LD        #RCV_INT1_DP,DP                 ; init. DP
       MVDK      d_rcv_in_ptr,GETFRM_IN_P        ; restore input bffr ptr
       MVDK      d_xmt_out_ptr,GETFRM_OUT_P      ; restore output bffr ptr
       ADDM      #1,d_index_count                ; increment the index count
       LD        #K_FRAME_SIZE,A
       SUB       d_index_count, A
       BC        get_samples,AGT                 ;check for a frame of samples
```

Example 3–5. Receive Interrupt Service Routine (Continued)

```
frame_flag_set
       ADDM   #1,d_int_count
       ST     #K_FRAME_FLAG,d_frame_flag       ; set frame flag
       ST     #0,d_index_count                 ; reset the counter
       LD     #input_data+K_FRAME_SIZE,A       ; second input bffr starting addr
       LD     #output_data+K_FRAME_SIZE,B      ; second output bffr starting addr
       BITF   d_int_count,2                    ; check for 1st/2nd  bffr
       BC     reset_buffer,NTC
       SUB    #K_FRAME_SIZE,A                  ; 1st input address
       SUB    #K_FRAME_SIZE,B                  ; 1st output address
       ST     #K_0,d_int_count
reset_buffer
       STLM   A,GETFRM_IN_P                    ; input buffer address
       STLM   B,GETFRM_OUT_P                   ; output buffer address
get_samples
       LDM    DRR1,A                           ; load the input sample
       STL    A,*GETFRM_IN_P+                  ; write to buffer
       LD     *GETFRM_OUT_P+,A                 ; if not true, then the filtered
       AND    #0fffch,A                        ; signal is send as output
       STLM   A,DXR1                           ; write to DXR1
       MVKD   GETFRM_IN_P,d_rcv_in_ptr         ; save input buffer ptr
       MVKD   GETFRM_OUT_P,d_xmt_out_ptr       ; save out bffr ptr
       POPM   BRC
       POPM   SAVE_RSTORE_AR3                  ; restore AR3
       POPM   SAVE_RSTORE_AR2                  ; restore AR2
       POPM   BG
       POPM   BH
       POPM   BL
       POPM   AG
       POPM   AH
       POPM   AL
       POPM   ST1
       POPM   ST0
       RETE                                    ; return and enable interrupts
       .end
```

3.6 Interrupt Priority

Interrupt prioritization allows interrupts that occur simultaneously to be serviced in a predefined order. For instance, infrequent but lengthy ISRs can be interrupted frequently. In Example 3–6, the ISR for the INT1 bit includes context save and restore macros. When the routine has finished processing, the IMR is restored to its original state. Notice that the RETE instruction not only pops the next program counter address from the stack, but also clears the INTM bit to 0. This enables all interrupts that have their IMR bit set.

Example 3–6. Interrupt Service Routine (ISR)

```
      .title  “Interrupt Service Routine”
      .mmregs
int1:
      CONTEXT_STORE              ; push the contents of accumulators and registers on stack
      STM     #K_INT0,IMR        ; Unmask only INT0~
      RSBX    INTM               ; Enable all Interrupts
;
; Main Processing for Receive Interrupt 1
.
.
.
      SSBX    INTM               ; Disable all interrupts
      CONTEXT_RESTORE            ; pop accumulators and registers
      RETE                       ; return and enable interrupts
      .end
```

There is a potential conflict between the INTM bit disable and context restore. If an interrupt 0 (INT0) occurs during context restore, the macro CONTEXT_RESTORE is executed before servicing INT0. This can trigger an INT0. If INTM is cleared during the context restore, it branches to the INT0 service routine. If you reenable the interrupts when INTM returns from INT0, a conflict occurs, because INTM is set to 0 and its original contents are lost. To preserve the contents of the INTM bit, do not enable the interrupts when INTM returns from the INT0 service routine. During interrupt priorities, preserve the INTM and IMR bits for the system requirements.

3.7 Circular Addressing

Circular addressing is an important feature of the ’54x instruction set. Algorithms for convolution, correlation, and FIR filters can use circular buffers in memory. In these algorithms, the circular buffers implement a sliding window that contains the most recent data. As new data comes in, it overwrites the oldest data. The size, the bottom address, and the top address of the circular buffer are specified by the block size register (BK) and a user-selected auxiliary register (ARn). A circular buffer size of R must start on a K-bit boundary (that is, the K LSBs of the starting address of the circular buffer must be 0), where K is the smallest integer that satisfies $2^K > R$.

Circular addressing can be used for different functions of an application. For example, it can be used for collecting the input samples in a block. It can also be used in processing samples in blocks and data in the output buffer. In Example 3–7, a frame of 256 samples is collected from the serial port to process the data using the circular addressing mode. The output from the processed block is sent to the D/A converter through the serial port register using circular buffers. A ping-pong buffering scheme is used. While processing the first buffer, samples are collected in the second buffer, and vice versa. The real-time operation of the system is not disturbed and no data samples are lost.

Example 3–7. Circular Addressing Mode

```
; TEXAS INSTRUMENTS INCORPORATED
   .mmregs
   .include     ”INTERRPT.INC”
   .include     ”main.inc”
RCV_INT1_DP     .usect ”rcv_vars”,0
d_index_count   .usect ”rcv_vars”,1
d_rcv_in_ptr    .usect ”rcv_vars”,1              ; save/restore input bffr ptr
d_xmt_out_ptr   .usect ”rcv_vars”,1              ; save/restore output bffr ptr
d_frame_flag    .usect ”rcv_vars”,1
input_data      .usect ”inpt_buf”,K_FRAME_SIZE*2  ; input data array
output_data     .usect ”outdata”,K_FRAME_SIZE*2   ; output data array
   .def         receive_int1
   .def         d_frame_flag
   .def         RCV_INT1_DP
   .def         input_data,output_data
   .def         d_xmt_out_ptr
   .def         d_rcv_in_ptr
;-------------------------------------------------------------------------------
;      Functional Description
;    This routine services receive interrupt1. Accumulator A, AR2 and AR3
;   are pushed onto the stack since AR2 and AR3 are used in other applications.
;   A 512 buffer size of both input and output uses circular addressing.
;   After every 256 collection of input samples a flag is set to process the
;   data. A PING/PONG buffering scheme is used such that upon processing

;   PING buffer, samples are collected in the PONG buffer and vice versa.
;-------------------------------------------------------------------------------
```

Example 3–7. Circular Addressing Mode (Continued)

```
    .asg   AR2,GETFRM_IN_P                    ; get frame input data pointer
    .asg   AR3,GETFRM_OUT_P                   ; get frame output data pointer
    .asg   AR2,SAVE_RSTORE_AR2
    .asg   AR3,SAVE_RSTORE_AR3
    .sect  ”main_prg”
receive_int1:
    PSHM   AL
    PSHM   AH
    PSHM   AG
    PSHM   BL
    PSHM   BH
    PSHM   BG
; AR2, AR3 are used in other routines, they need to be saved and restored
; since receive interrupt uses AR2 and AR3 as pointers
    PSHM   SAVE_RSTORE_AR2                    ; Since AR2 and AR3 are used
    PSHM   SAVE_RSTORE_AR3                    ; in other routines, they need
    PSHM   BRC
    STM    #2*K_FRAME_SIZE,BK                 ; circular buffer size of in,out
                                              ; arrays
    LD     #RCV_INT1_DP,DP                    ; init. DP
    MVDK   d_rcv_in_ptr,GETFRM_IN_P           ; restore input circular bffr ptr
    MVDK   d_xmt_out_ptr,GETFRM_OUT_P         ; restore output circular bffr ptr
    ADDM   #1,d_index_count                   ; increment the index count
    LD     #K_FRAME_SIZE,A
    SUB    d_index_count, A
    BC     get_samples,AGT                    ;check for a frame of samples
frame_flag_set
    ST     #K_FRAME_FLAG,d_frame_flag         ; set frame flag
    ST     #0,d_index_count                   ; reset the counter
get_samples
    LDM    DRR1,A                             ; load the input sample
    STL    A,*GETFRM_IN_P+%                   ; write to buffer
    LD     *GETFRM_OUT_P+%,A                  ; if not true, then the filtered
    AND    #0fffch,A                          ; signal is send as output
    STLM   A,DXR1                             ; write to DXR1
    MVKD   GETFRM_IN_P,d_rcv_in_ptr           ; save input circular buffer ptr
    MVKD   GETFRM_OUT_P,d_xmt_out_ptr         ; save out circular bffr ptr
    POPM   BRC
    POPM   SAVE_RSTORE_AR3                    ; restore AR3
    POPM   SAVE_RSTORE_AR2                    ; restore AR2
    POPM   BG
    POPM   BH
    POPM   BL
    POPM   AG
    POPM   AH
    POPM   AL
    POPM   ST1
    POPM   ST0
    RETE                                      ; return and enable interrupts
    .end
```

3.8 Buffered Serial Port

The buffered serial port (BSP) is made up of a full-duplex, double-buffered serial port interface, which functions in a similar manner to the ’54x standard serial port, and an autobuffering unit (ABU). The serial port section of the BSP is an enhanced version of the ’54x standard serial port. The ABU is an additional section of logic which allows the serial port section to read/write directly to ’54x internal memory independent of the CPU. This results in a minimum overhead for serial port transfers and faster data rates. The full duplex BSP serial interface provides direct communication with serial devices such as codecs, serial A/D converters, and other serial devices with a minimum of external hardware. The double-buffered BSP allows transfer of a continuous communication stream in 8-,10-,12- or 16-bit data packets. This section uses the ’542 device to verify the BSP functionality.

The autobuffering process occurs between the ABU and the 2K-word block of ABU memory. Each time a serial port transfer occurs, the data involved is automatically transferred to or from a buffer in the 2K-word block of memory under control of the ABU. No interrupts are generated with each word transfer in autobuffering mode. Interrupts are generated to the CPU each time one of the half-boundaries is crossed. When autobuffering capability is disabled (standard mode), transfers with the serial port are performed under user control (software). When autobuffering is enabled, word transfers can be done directly between the serial port and ’54x internal memory using the ABU embedded address generators. The ABU has its own set of circular addressing registers with corresponding address generation units. The length and starting addresses of the buffers are user programmable. A buffer empty/full interrupt can be posted to the CPU.

The six MSBs in the BSP control extension register (BSPCE) configure the ABU. Bits 14 and 11 are read only and the remaining bits are read/write. Figure 3–4 shows the bit positions of BSPCE and Table 3–1 provides a summary of each bit.

Figure 3–4. BSP Control Extension Register (BSPCE) Diagram

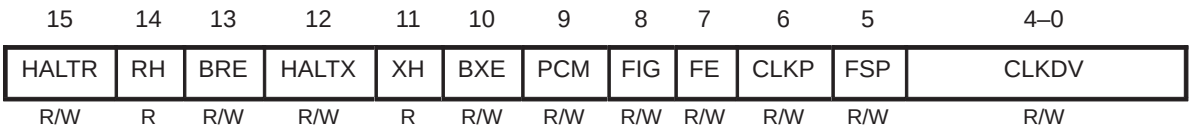


Table 3–1. BSP Control Extension Register (BSPCE) Bit Summary

Bit	Name	Function
15	HALTR	Autobuffering Receive Halt. This control bit determines whether autobuffering receive is halted when the current half of the buffer has been received.
		HALTR = 0 — Autobuffering continues to operate when the current half of the buffer has been received.
		HALTR = 1 — Autobuffering is halted when the current half of the buffer has been received. When this occurs, the BRE bit is cleared to 0 and the serial port continues to operate in standard mode.
14	RH	Receive Buffer Half Received. This read-only bit indicates which half of the receive buffer has been filled. Reading RH when the RINT interrupt occurs (seen either as a program interrupt or by polling IFR) is a convenient way to identify which boundary has just been crossed.
		RH = 0 — The first half of the buffer has been filled and that receptions are currently placing data in the second half of the buffer.
		RH = 1 — The second half of the buffer has been filled and that receptions are currently placing data in the first half of the buffer.
13	BRE	Autobuffering Receive Enable. This control bit enables autobuffering receive.
		BRE = 0 — Autobuffering is disabled and the serial port interface operates in standard mode.
		BRE = 1 — Autobuffering is enabled for the receiver.
12	HALTX	Autobuffering Transmit Halt. This control bit determines whether autobuffering transmit is halted when the current half of the buffer has been transmitted.
		HALTX = 0 — Autobuffering continues to operate when the current half of the buffer has been transmitted.
		HALTX = 1 — Autobuffering is halted when the current half of the buffer has been transmitted. When this occurs, the BXE bit is cleared to 0 and the serial port continues to operate in standard mode.
11	XH	Transmit Buffer Half Transmitted. This read-only bit indicates which half of the transmit buffer has been transmitted. Reading XH when the XINT interrupt occurs (seen either as a program interrupt or by polling IFR) is a convenient way to identify which boundary has just been crossed.
		XH = 0 — The first half of the buffer has been transmitted and transmissions are currently taking data from the second half of the buffer.
		XH = 1 — The second half of the buffer has been transmitted and transmissions are currently taking data from the first half of the buffer.

Table 3–1. BSP Control Extension Register (BSPCE) Bit Summary (Continued)

Bit	Name	Function
10	BXE	Autobuffering Transmit Enable. This control bit enables the autobuffering transmit. BXE = 0 Autobuffering is disabled and the serial port operates in standard mode. BXE = 1 Autobuffering is enabled for the transmitter.
9	PCM	Pulse Code Modulation Mode. This control bit puts the serial port in pulse code modulation (PCM) mode. The PCM mode only affects the transmitter. BDXR-to-BXSR transfer is not affected by the PCM bit value. PCM = 0 Pulse code modulation mode is disabled. PCM = 1 Pulse code modulation mode is enabled. In PCM mode, BDXR is transmitted only if its most significant (2^{15}) bit is set to 0. If this bit is set to 1, BDXR is not transmitted and BDX is put in high impedance during the transmission period.
8	FIG	Frame Ignore. This control bit operates only in transmit continuous mode with external frame and in receive continuous mode. FIG = 0 Frame sync pulses following the first frame pulse restart the transfer. FIG = 1 Frame sync pulses following the first frame pulse that initiates a transfer operation are ignored.
7	FE	Format Extension. The FE bit in conjunction with FO in the SPC register specifies the word length. When FO FE = 00, the format is 16-bit words; when FO FE = 01, the format is 10-bit words; when FO FE = 10, the format is 8-bit words; and when FO FE = 11, the format is 12-bit words. Note that for 8-, 10-, and 12-bit words, the received words are right justified and the sign bit is extended to form a 16-bit word. Words to transmit must be right justified.
6	CLKP	Clock Polarity. This control bit specifies when the data is sampled by the receiver and transmitter. CLKP = 0 Data is sampled by the receiver on BCLKR falling edge and sent by the transmitter on BCLKX rising edge. CLKP = 1 Data is sampled by the receiver on BCLKR rising edge and sent by the transmitter on BCLKX falling edge.
5	FSP	Frame Sync Polarity. This control bit specifies whether frame sync pulses (BFSX and BFSR) are active high or low. FSP = 0 Frame sync pulses (BFSX and BFSR) are active high. FSP = 1 Frame sync pulses (BFSX and BFSR) are active low.

Table 3–1. BSP Control Extension Register (BSPCE) Bit Summary (Continued)

Bit	Name	Function
4–0	CLKDV	Internal Transmit Clock Division factor. When the MCM bit of BSPC is set to 1, CLKX is driven by an on-chip source having a frequency equal to 1/(CLKDV+1) of CLKOUT. CLKDV range is 0–31. When CLKDV is odd or equal to 0, the CLKX duty cycle is 50%. When CLKDV is an even value (CLKDV=2p), the CLKX high and low state durations depend on CLKP. When CLKP is 0, the high state duration is p+1 cycles and the low state duration is p cycles; when CLKP is 1, the high state duration is p cycles and the low state duration is p+1 cycles.

The autobuffering process for transmit is illustrated in Figure 3–5 and for receive in Figure 3–6. When a process is activated upon request from the serial port (XRDY = 1 or RRDY = 1), four actions are performed:

1) ’54x internal memory access,

2) address register update,

3) decision for interrupt,

4) autodisabling management.

An interrupt is generated whenever the first or second half-of-buffer is processed. The RH and XH bits in BSPCE allow you to know which half has been processed when an interrupt boundary is found. For further details on BSP operation, refer to the *TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals*.

Figure 3–5. Autobuffering Process for Transmit

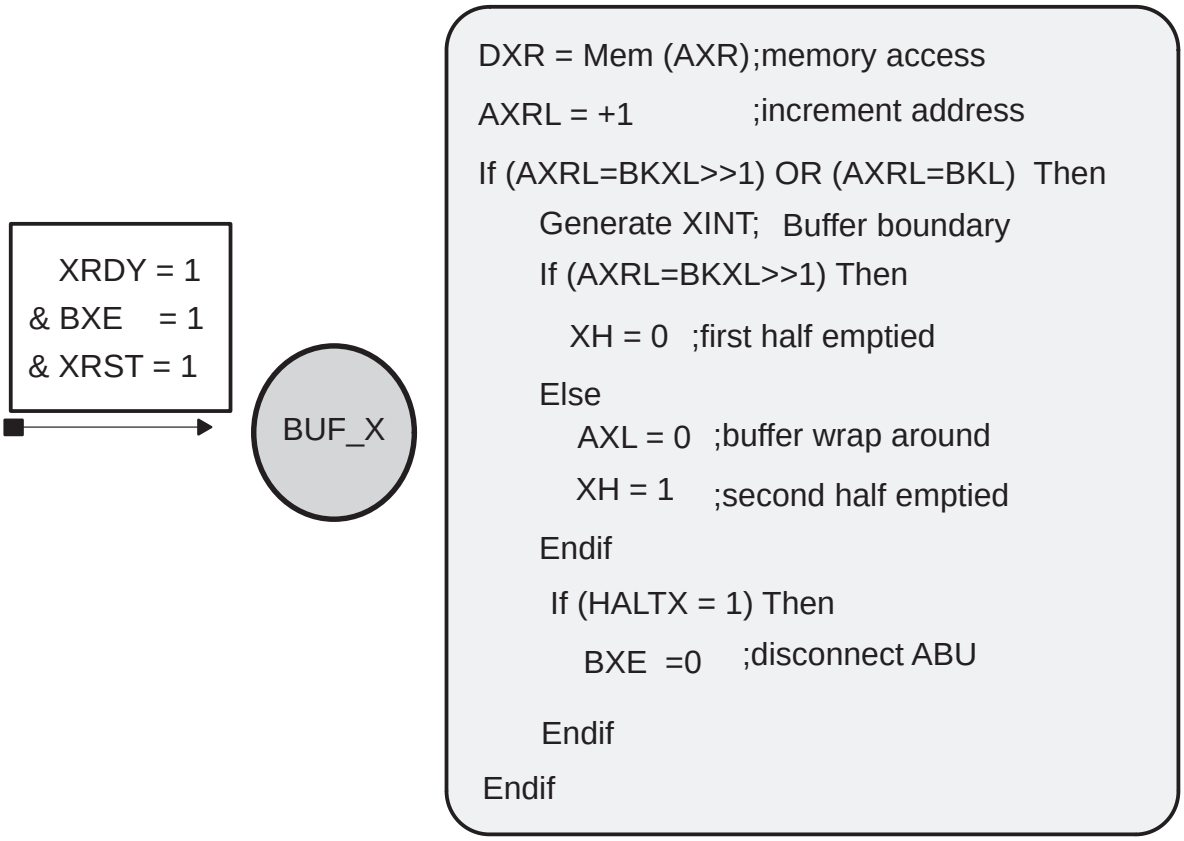
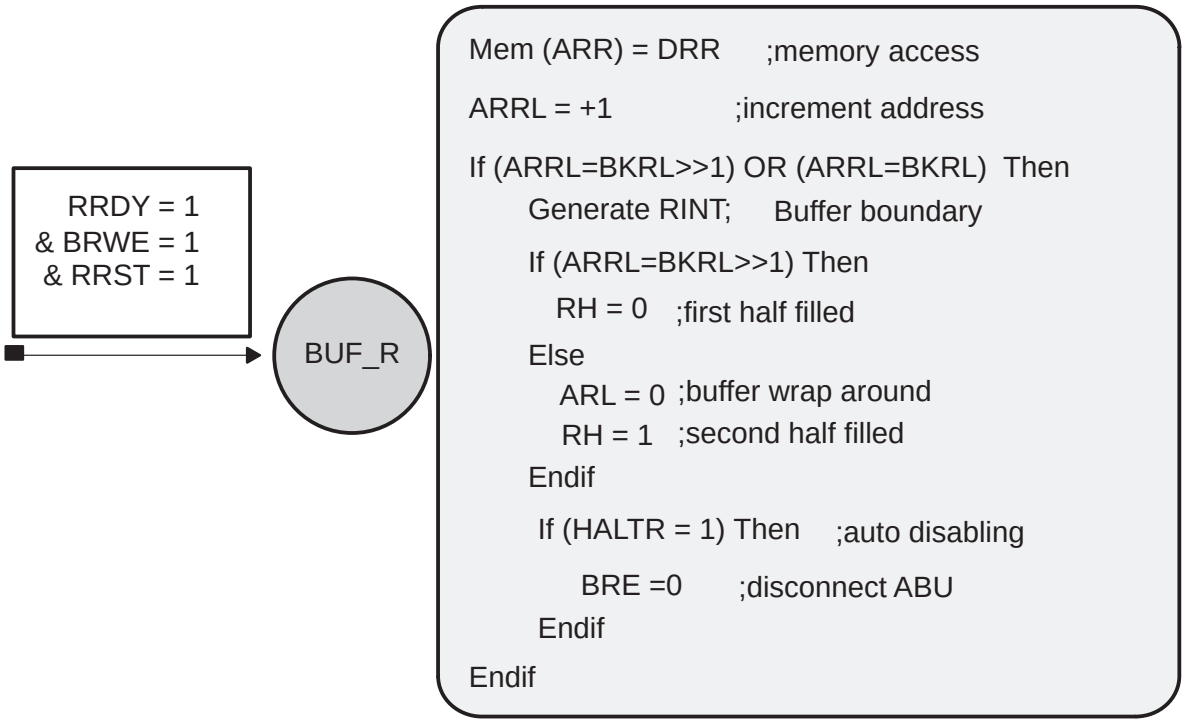


Figure 3–6. Autobuffering Process for Receive



Initialization Examples

In order to start or restart BSP operation in standard mode, the same steps are performed in software as with initializing the serial port (see *TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals*), in addition to which, the BSPCE register must be initialized to configure any of the enhanced features desired. To start or restart the BSP in autobuffering mode, a similar set of steps must also be performed, in addition to which, the autobuffering registers must be initialized.

As an illustration of the proper operation of a buffered serial port, Example 3–8 and Example 3–9 define a sequence of actions. The ’542 peripheral configuration has been used as a reference for these examples. The examples show the actions for initializing the BSP for autobuffering mode operation. In both cases, assume that transmit and receive interrupts are used to service the ABU interrupts.

Example 3–8 initializes the serial port for transmit operations only, with burst mode, external frame sync, and external clock selected. The selected data format is 16 bits, with frame sync and clock polarities selected to be high true. Transmit autobuffering is enabled by setting the BXE bit in the BSPCE, and HALTX has been set to 1, which causes transmission to halt when half of the defined buffer is transmitted.

Example 3–9 initializes the serial port for receive operations only. Receive autobuffering is enabled by setting the BRE and HALTR bits in the BSPCE to 1.

The complete initialization code is shown in Example 3–10 on page 3-36.

Example 3–8. BSP Transmit Initialization Routine

	Action	Description
1)	Reset and initialize the serial port by writing 0008h to SPC.	This places both the transmit and receive portions of the serial port in reset and sets up the serial port to operate with externally generated FSX and CLKX signals and FSX required for transmit/receive of each 16-bit word.
2)	Clear any pending serial port interrupts by writing 0020h to IFR.	Eliminate any interrupts that may have occurred before initialization.
3)	Enable the serial port interrupts by ORing 0020h with IMR.	Enable transmit interrupts.
4)	Enable interrupts globally (if necessary) by clearing the INTM bit in ST1.	Interrupts must be globally enabled for the CPU to respond.
5)	Initialize the ABU transmit by writing 1400h to BSPCE.	This causes the autobuffering mode to stop when the current half-of-buffer has been transmitted.
6)	Write the buffer start address to AXR.	Identify the transmit buffer register address.
7)	Write the buffer size to BKX.	Identify the buffer size of the ABU.
8)	Start the serial port by writing 0048h to SPC.	This takes the transmit portion of the serial port out of reset and starts operations with the conditions defined in steps 1 and 5.

Example 3–9. BSP Receive Initialization Routine

	Action	Description
1)	Reset and initialize the serial port by writing 0000h to SPC.	This places both the transmit and receive portions of the serial port in reset and sets up the serial port to operate with externally generated FSR and CLKR signals and FSR required for transmit/receive of each 16-bit word.
2)	Clear any pending serial port interrupts by writing 0010h to IFR.	Eliminate any interrupts that may have occurred before initialization.
3)	Enable the serial port interrupts by ORing 0010h with IMR.	Enable receive interrupts.
4)	Enable interrupts globally (if necessary) by clearing the INTM bit in ST1.	Interrupts must be globally enabled for the CPU to respond.
5)	Initialize the ABU transmit by writing A000h to BSPCE.	This causes the autobuffering mode to stop when the current half-of-buffer has been received.
6)	Write the buffer start address to ARR.	Identify the receive buffer register address.
7)	Write the buffer size to BKR.	Identify the buffer size of the ABU.
8)	Start the serial port by writing 0088h to SPC.	This takes the receive portion of the serial port out of reset and starts operations with the conditions defined in steps 1 and 5.

Example 3–10. BSP initialization Routine

```
             .mmregs
K_STACK_SIZE .set   100
STACK        .usect “stack”,K_STACK_SIZE
SYSTEM_STACK .set   STACK_K_STACK_SIZE
             .def   SYSTEM_STACK
             .ref   main_start
             .ref   bsp_receive_int
             .ref   bsp_transmit_int
             .sect  ”vectors”
reset: BD    main_start          ; RESET vector
       STM   #SYSTEM_STACK,SP
nmi:   RETE
       NOP
       NOP
       NOP                       ;NMI~
; software interrupts
sint17       .space  4*16
sint18       .space  4*16
sint19       .space  4*16
sint20       .space  4*16
sint21       .space  4*16
sint22       .space  4*16
sint23       .space  4*16
sint24       .space  4*16
sint25       .space  4*16
sint26       .space  4*16
sint27       .space  4*16
sint28       .space  4*16
sint29       .space  4*16
sint30       .space  4*16
int0:  RETE
       NOP
       NOP                       ; INT0
       NOP
int1:  RETE                      ;
       NOP
       NOP                       ; INT1
       NOP
int2:  RETE
       NOP
       NOP
       NOP
tint:  RETE
       NOP
       NOP                       ; TIMER
       NOP
brint: BD    bsp_receive_int     ; Buffered serial port receive intr
       PSHM  ST0
       PSHM  ST1
bxint: BD    bsp_transmit_int    ; Buffered serial port transmit intr
       PSHM  ST0
       PSHM  ST1
trint: RETE                      ; TDM serial port transmit intr
       NOP
```

Example 3–10. BSP initialization Routine (Continued)

```
       NOP
       NOP
txint: RETE                        ; TDM serial port transmit interrupr
       NOP
       NOP
       NOP
int3:  RETE
       NOP
       NOP                         ;INT3
       NOP
hpiint:RETE                        ; HPI interrupt
       NOP
       NOP
       NOP
       .end
   .mmregs
   .include     ”interrpt.inc”
   .include     ”init_ser.inc”
K_AUTO_BFFR_SIZE .set      8
rtop_bffr     .usect “auto_rcv”,K_AUTO_BFFR_SIZE ; starting address of receive bffr
xtop_bffr     .usect “auto_xmt”,K_AUTO_BFFR_SIZE ; starting address of transmit bffr
   .def      rtop_bffr
   .def      xtop_bffr
   .def      serial_init
;  This routine initializes the BSP of 542. The serial port is put
;  in reset by writting 0’s to RRST and XRST bits and pulled out of reset by
;  writting 1’s to both RRST and XRST bits. The BSPCE register is init such
;  that autobuffer is enabled. Also HALTX and HALTR are enabled to halt the
;  autobuffering scheme whenever half buffer is either received or transmitted
;
;-------------------------------------------------------------------------------
   .sect  ”ser_cnfg”
serial_init:
*******************************************************************************
* We need atleast 12 cycles to pull the AIC out of reset.
*******************************************************************************
   RSBX   XF                       ; Put the AC01 in reset
   STM    #K_SERIAL_RST, BSPC      ;reset the serial port with
                                   ;0000 0000 0000 1000
   STM    #K_ABUC,BSPCE            ; enable auto–buffer
   STM    #K_SERIAL_OUT_RST, BSPC  ;bring ser.port out of reset with
                                   ;0000 0000 1100 1000
   RSBX   INTM                     ; Enable interrupts
   LD     #0,DP
   ORM    #(K_BXINT|K_BRINT),IMR   ; Enable both BSP receive
                                   ; transmit interrupt
                                   ; 0000 0000 0011 0000
   STM    #(K_BXINT|K_BRINT),IFR   ; clear BXINT,BRINT
; Pull the AC01 out of reset – the AC01 requires that it be held in reset for
; 1 MCLK, which is equivalent to 96.45ns (based on an MCLK of 10.368MHz)
   SSBX   XF                       ; Pull AC01 out of reset
   RET
   .end
```

Example 3–10. BSP initialization Routine (Continued)

```
*************************************************************************
* FILENAME: SERIAL_INIT.INC"
* This include file includes the BSPC register configuration
; SPC Register Organization
*  ---------------------------------------------------------
*  | 15  | 14  |   13   |   12    | 11   |  10  | 9  |  8  |
*  |-----|-----|--------|---------|------|------|----|-----|
*  |FREE | SOFT| RSRFULL| XSREMPTY| XRDY | RRDY | IN1| IN0 |
*  ---------------------------------------------------------
*  -------------------------------------------------
*  | 7   | 6   |   5 | 4  |  3  |  2 | 1   |  0  |
*  |-----|-----|-----|----|-----|----|-----|-----|
*  |RRST | XRST| TXM | MCM| FSM | FO | DLB | RES |
*  -------------------------------------------------
*************************************************************************
;Bit   Name      Function
;0     Reserved  Always read as 0
;1     DLB       Digital loop back : 0 –> Disabled, 1 _. Enabled
;2     FO        Format bit: 0 – > data transfered as 8 bit bytes,
;                          1 –> 16 bit words
;3     FSM       Frame sync pulse: 0 –> serial port in continuous mode,
;                                  1 –> FSM is required
;4     MCM       Clock mode bit: 0 –> CLKX obtained from CLKX pin
;                               1–> CLKX obtained from CLKX
;5     TXM       Transmit mode bit: 0 –> Frame sync pulses generated externally
;                                  1–> Internally generated frame sync
;6     XRST      Transmit reset bit:0 –> reset the serial port,
;                                  1–> bring serial port out of reset
;7     RRST      Receive reset bit: 0 –> reset the serial port,
;                                  1–> bring serial port out of reset
;8     IN0       Read–only bit reflecting the state of the CLKR pin
;9     IN1       Read–only bit reflecting the state of the CLKX pin
;10    RRDY      Transition from 0 to 1 indicates data is ready to be read
;11    XRDY      Transition from 0 to 1 indicates data is ready to be sent
;12    XSREMPTY  Transmit shift register empty  ( Read–only)
;                                  0 –> tramsitter has experienced underflow,
;                                  1–> has not expereinced underflow
;13    RSRFUL    Receive shift register full flag   (Read–only):
;                                  0 –> Receiver has experienced overrun,
;                                  1–> receiver has not experienced overrun
;14    SOFT      Soft bit – 0 –> immdeiate stop, 1–> stop after word completion
;15    FREE      Free run bit: 0 –> behaviour depends on SOFT bit,
;                               1–> free run regardless of SOFT bit
K_0             .set   00000000b << 8   ; bits 15–8 to 0 at reset
K_RRST          .set   0b << 7          ; First write to BSPC is 0
;                                       ; second write is 1
K_XRST          .set   0b << 6          ; First write to BSPC is 0
;                                       ; second write is 1
K_TXM           .set   0b << 5
K_MCM           .set   0b << 4
K_FSM           .set   1b << 3          ; Frame Sync mode
K_ZERO          .set   000b << 0
K_SERIAL_RST    .set   K_0|K_RRST|K_XRST|K_TXM|K_MCM|K_FSM|K_ZERO
;                                       ; first write to BSPC regsiter
```

Example 3–10. BSP initialization Routine (Continued)

```
K_RRST1          .set   1b << 7          ; second write to BSPC
K_XRST1          .set   1b << 6          ; second write to BSPC
K_SERIAL_OUT_RST .set   K_0|K_RRST1|K_XRST1|K_TXM|K_MCM|K_FSM|K_ZERO
                                         ; second write to SPC1 regsiter
K_TRGCR_ADDR     .set   14h              ; Timer Control Register I/O
                                         ; address
K_0              .set   0h
K_8000           .set   8000h            ; set bit 15 to pull AIC out
                                         ; of reset
; BSPCE Register Organization
*  ------------------------------------------------------------------------
*  | 15  | 14 | 13 | 12   | 11 | 10 | 9  | 8  | 7  |  6 |  5  | 4      0 |
*  |-----|----|----|------|----|----|----|----|----|----|-----|----------|
*  |HALTR| RH | BRE| HALTX| XH | BXE| PCM| FIG| FE |CLKP| FSP |  CLKDIV  |
*  ------------------------------------------------------------------------
; Auto-Buffering Control Register (ABU)
K_CLKDV          .set   00000 << 0       ; External clock
K_FSP            .set   0 << 5           ; Frame sync pulses are active
                                         ; high at reset 0
K_CLKP           .set   0 << 6           ; CLKP = 0 at reset
K_FE             .set   0 << 7           ; 16 bit format
K_FIG            .set   0 << 8           ; at reset =0
K_PCM            .set   0 << 9           ; no PCM mode
K_BXE            .set   1 << 10          ; enable transmit auto-buffer
                                         ; mode
K_HALTX          .set   1 << 12          ; auto-buffer is halted after
                                         ; half buffer has been trans
                                         ; mitted
K_BRE            .set   1 << 13          ; enable receive auto-buffer
K_HALTR          .set   1 << 15          ; auto-buffer is halted after
                                         ; half buffer has been received
K_SPIC           .set   K_CLKDV|K_FSP|K_CLKP|K_FE|K_FIG|K_PCM
K_ABUC           .set   K_SPIC|K_BXE|K_HALTX|K_BRE|K_HALTR
K_XH             .set   0800h            ; transmit half buffer check
K_RH             .set   4000h            ; receive half buffer check
* FILENAME: INTERRUPT.INC
* This file includes the IMR and IFR configuration
; 542Interrupt Mask Register (IMR) Organization
*-------------------------------------------------------------------------------
*| 15    10|    9  | 8  |  7   |   6   |  5   |  4    | 3    | 2    | 1   | 0  |
*|---------|-------|----|------|-------|------|-------|------|------|-----|----|
*|Reserved | HPIINT|INT3| XINT1| RITN1 | XINT0| RINT0 | TINT | INT2 |INT1 |INT0|
*-------------------------------------------------------------------------------
*******************************************************************************
K_IMR_RESR       .set   0000000b << 9       ; reserved space
K_HPIINT         .set   1b << 8             ; disable HPI interrupt
K_TXINT          .set   1b << 7             ; enable TDM transmit 1
K_TRINT          .set   1b << 6             ; enable TDM receive 1
K_BXINT          .set   1b << 5             ; enable BSP transmit
                                            ; interrupt
K_BRINT          .set   1b << 4             ; enable BSP receive intr
K_TINT           .set   1b << 3             ; enable timer interrupt
K_INT2           .set   1b << 2             ; enable INT2
K_INT1           .set   1b << 1             ; enable INT1
```

Example 3–10. BSP initialization Routine (Continued)

```
K_INT0          .set   1b << 0              ; enable INT0
       .mmregs
       .include  ”interrpt.inc”
       .include  ”init_ser.inc”
       .ref      rtop_bffr
       .ref      wrt_cnfg                  ; initializes AC01
       .def      aic_init
;  5.2 Functional Description
;      This routine disables IMR and clears any pending interrupts before
;      initializing AC01. The wrt_cnfg function configures the AC01
;-------------------------------------------------------------------------------
       .sect  ”aic_cnfg”
aic_init:
       CALLD  wrt_cnfg                     ; initialize AC01
       ANDM   #~(K_BXINT|K_BRINT),IMR      ; disable receive_int1
       LD     #rtop_bffr,A
       STLM   A,AXR                        ; PING/PONG buffering scheme
       STLM   A,ARR                        ; is used
       STM    #8,BKX                       ; transmit circular size
       STM    #8,BKR                       ; receive circular size
       ORM    #(K_BXINT|K_BRINT),IMR       ; enable the RINT1, INT1
       RETD
       STM    #(K_BXINT|K_BRINT),IFR       ; service any pending interrupt
       .end
       .include  ”aic_cfg.inc”
       .def      wrt_cnfg
;      Writes new configuration data into the AC01. Assuming a system
;      which processes speech signals and *  requires the following parameters
;      Low pass filter cut–off frequency = 3.6 kHz
;      Sampling rate = 8000 Hz
;      Assume the Master clock MCLK = 10.368 MHz
;      This example demonstrates how to program these parameters –
;      the registers affected are:
;      Register A which determines the division of the MCLK frequency
;      to generate the internal filter clock FCLK.
;      It also determines the –3 dB corner frequency of the low–pass filter
;      Register B  which determines the division of FCLK to generate
;      the sampling (conversion) frequency
;      It also determines the –3dB corner frequency of the high–pass filter
;-------------------------------------------------------------------------------
       .asg   AR1,AIC_REG_P
       .sect  ”aic_cnfg”
wrt_cnfg:
       STM    #aic_reg_tble,AIC_REG_P          ;init AR1
       RPT    #AIC_REG_END_LIST–AIC_REG_START_LIST
       MVPD   #AIC_REG_START_LIST,*AIC_REG_P+   ; move the table
       STM    #aic_reg_tble,AIC_REG_P    ;init AR1
       STM    #K_REG_0,BDXR              ;primary data word – a jump start!
wait_xrdy
       BITF   BSPC,K_XRDY                ;test XRDY bit in BSPC
       BC     wait_xrdy,NTC              ;loop if not set
       STM    #K_SCND_CONTRL,BDXR        ;send primary word with D01–D00 = 11 to
                                         ;signify secondary communication
       LD     *AIC_REG_P+,A
```

Example 3–10. BSP initialization Routine (Continued)

```
       STLM   A,BRC                          ;gives the # of regsiters to be
       NOP                                   ;initialized
       RPTB   aic_cfg_complte–1
wait_xrdy1
       BITF   BSPC,K_XRDY                    ;test XRDY bit in BSPC
       BC     wait_xrdy1,NTC                 ;loop if not set
       LD     *AIC_REG_P+,A                  ;Read the register contents
       STLM   A, BDXR
wait_xrdy2
       BITF   BSPC,K_XRDY                    ;test XRDY bit in BSPC
       BC     wait_xrdy2,NTC                 ;loop if not set
       STM    #K_SCND_CONTRL,BDXR            ;set to read the next register
aic_cfg_complte                              ;contents
       RET
       .end
*************************************************************************
 * FILENAME: AIC_CFG.INC
* This file includes the AC01 registers initialization
* All registers have 2 control bits that initiates serial communication
* There are 2 communication modes – primary and secondary communications
* During primary communication the control bits D00 and D01 are 11 to request
* for a secondary communication. In the secondary serial communications the
* control bits D15 and D14 perform same control function as primary.
* The R/W~ bit at reset is set to 0 placing the device in write mode.
*************************************************************************
K_NOP_ADDR       .set   0 << 8
K_REG_0          .set   K_NOP_ADDR
K_A_ADDR         .set   1 << 8               ; REG 1 address
K_A_REG          .set   36
K_REG_1          .set   K_A_ADDR|K_A_REG     ; FCLK = 144KHz => A =24h
K_B_ADDR         .set   2 << 8               ; REG 2 address
K_B_REG          .set   18
K_REG_2          .set   K_B_ADDR|K_B_REG     ; Sampling rate = 8KHz
K_AA_ADDR        .set   3 << 8               ; Register 3 address
K_AA_REG         .set   0
K_REG_3          .set   K_AA_ADDR|K_AA_REG   ; ; no shift
K_GAIN_ADDR      .set   4 << 8               ; Register 4 address
K_MONITOR_GAIN   .set   00b << 4             ; Monitor output gain = squelch
K_ANLG_IN_GAIN   .set   01b << 2             ; Analog input gain = 0dB
K_ANLG_OUT_GAIN  .set   01b << 0             ; Analog output gain = 0dB
K_REG_4          .set   K_GAIN_ADDR|K_MONITOR_GAIN|K_ANLG_IN_GAIN|K_ANLG_OUT_GAIN
K_ANLG_CNF_ADDR  .set   5 << 8               ; Register 5 address
K_ANLG_RESRV     .set   0 << 3               ; Must be set to 0
K_HGH_FILTER     .set   0 << 2               ; High pass filter is enabled
K_ENBL_IN        .set   01b << 0             ; Enables IN+ and IN–
K_REG_5          .set   K_ANLG_CNF_ADDR|K_ANLG_RESRV|K_HGH_FILTER|K_ENBL_IN
K_DGTL_CNF_ADDR  .set   6 << 8               ; Register 6 address
K_ADC_DAC        .set   0 << 5               ; ADC and DAC is inactive
K_FSD_OUT        .set   0 << 4               ; Enabled FSD output
K_16_BIT_COMM    .set   0 << 3               ; Normal 16–bit mode
K_SECND_COMM     .set   0 << 2               ; Normal secondary communication
K_SOFT_RESET     .set   0 << 1               ; Inactive reset
K_POWER_DWN      .set   0 << 0               ; Power down external
K_REG_HIGH_6     .set   K_DGTL_CNF_ADDR|K_ADC_DAC|K_FSD_OUT|K_16_BIT_COMM
```

Example 3–10. BSP initialization Routine (Continued)

```
K_REG_LOW_6      .set   K_SECND_COMM|K_SOFT_RESET|K_POWER_DWN
K_REG_6          .set   K_REG_HIGH_6|K_REG_LOW_6
K_FRME_SYN_ADDR  .set   7 << 8              ; Register 7 address
K_FRME_SYN       .set   0 << 8              ;
K_REG_7          .set   K_FRME_SYN_ADDR|K_FRME_SYN
K_FRME_NUM_ADDR  .set   8 << 8              ; Register 8 address
K_FRME_NUM       .set   0 << 8              ;
K_REG_8          .set   K_FRME_NUM_ADDR|K_FRME_NUM
; primary word with D01 and D00 bits set to 11 will cause a secondary
; communications interval to start when the frame sync goes low next
K_SCND_CONTRL           .set   11b << 0     ; Secondary communication request
AIC_REG_START_LIST      .sect  "aic_reg"    ; includes the aic table
                 .word  AIC_REG_END_LIST-AIC_REG_START_LIST-1
                 .word  K_REG_1
                 .word  K_REG_2
                 .word  K_REG_3
                 .word  K_REG_4
                 .word  K_REG_5
                 .word  K_REG_6
                 .word  K_REG_7
                 .word  K_REG_8
AIC_REG_END_LIST
aic_reg_tble             .usect "aic_vars",10
K_XRDY           .set   0800h               ; XRDY bit in BSPC
       .mmregs
       .include  "INTERRPT.INC"
       .include  "init_ser.inc"
       .def      bsp_receive_int
       .def      bsp_transmit_int
;       bsp transmit and receive interrupt service routine
;      This routine performs BSP receive interrupt and transmit interrupt in
;      autobuffering mode. Since HALTX and HALTR = 1, the autobuffering mode
;      is disabled thus BXE and BRE =1 to continue in autobuffering mode.
;-------------------------------------------------------------------------------
       .sect     "main_prg"
bsp_receive_int:
       LD        #0,DP
       ORM       #K_BRE,BSPCE              ; enable the auto-buffer mode
       POPM      ST1
       POPM      ST0
       RETE                                ; return and enable interrupts
*******************************************************************************
bsp_transmit_int:
       LD        #0,DP
       ORM       #K_BXE,BSPCE              ; enable the auto-buffer mode
       POPM      ST1
       POPM      ST0
       RETE                                ; return and enable interrupts
       .end
```

Example 3–10. BSP initialization Routine (Continued)

```
       .mmregs
       .include  ”init_54x.inc”
       .include  ”main.inc”
       .include  ”init_ser.inc”
       .ref   d_auto_bffr_flag,d_auto_bffr_reg,rtop_bffr
       .ref   RCV_INT1_DP
       .ref   aic_init,serial_init,init_54,init_bffr_ptr_var
       .def   main_start
       .sect  ”main_prg”
main_start:
   CALL   init_54                    ; initialize ST0,ST1 PMST and other regsiters
   CALL   init_bffr_ptr_var          ; init tables,vars,bffrs,ptr
   CALL   serial_init                ; initialize serial_port 1
   .if K_DEFAULT_AC01 = 1
   CALLD  aic_init                   ; Configures AC01
   LD     #0,DP
   NOP
   .else                             ; default AC01 config
   LD     #rtop_bffr,A               ; sampling rate = 16KHz
   STLM   A,AXR                      ; init. the buffers
   STLM   A,ARR
   STM    #8,BKX                     ; transmit circular size
   STM    #8,BKR                     ; receive circular size
   .endif
start_loop
   LD     #RCV_INT1_DP,DP            ; restore the DP
loop:
   BITF   d_auto_bffr_flag,1         ; check if auto–buffering scheme
   BC     loop,NTC                   ; is enabled
   ST     #0,d_auto_bffr_flag
   LD     #rtop_bffr,A               ; PING buffer address
   MVKD   BSPCE,d_auto_bffr_reg      ; load the status of SPCE
   BITF   d_auto_bffr_reg,K_RH       ; check if first is emptied
   BC     half_buffer_empty,NTC
   ADD    #K_AUTO_BFFR_SIZE/2,A      ; PONG buffer
half_buffer_empty
   STLM   A, AXR                     ; like PING/PONG buffer scheme
   B      loop
.end
```

Chapter 4

Signal Processing

Certain features of the ’54x architecture and instruction set facilitate the solution of numerically intensive problems. Some examples include filtering, encoding techniques in telecommunication applications, and speech recognition. This chapter discusses digital filters that use both fixed and adaptive coefficients and fast Fourier transforms.

Topic **Page**

4.1 Finite Impulse Response (FIR) Filters

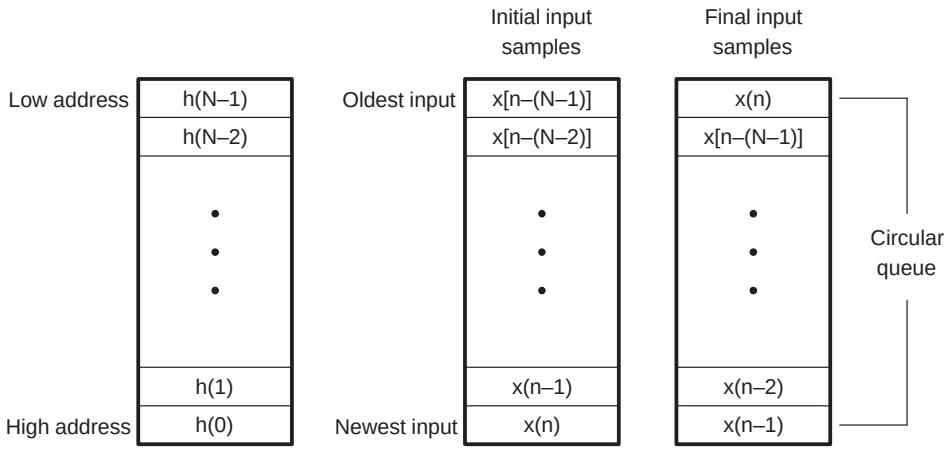
Digital filters are a common requirement for digital signal processing systems. There are two types of digital filters: finite impulse response (FIR) and infinite impulse response (IIR). Each of these can have either fixed or adaptive coefficients.

If an FIR filter has an impulse response, h(0), h(1), ..., h(N–1), and x(n] represents the input of the filter at time n, the output y(n] at time n is given by the following equation:

$$y(n) = h(0)x(n) + h(1)x(n–1) + h(2)x(n–2) + ...+ h(N–1)x[n–(N–1)]$$

Figure 4–1 illustrates a method using circular addressing to implement an FIR filter. To set up circular addressing, initialize BK to block length N. The locations for d_data_buffer, and impulse responses, COFF_FIR, must start from memory locations whose addresses are multiples of the smallest power of 2 that is greater than N. For instance, if N = 11, the first address for d_data_buffer must be a multiple of 16. Thus, the lowest four bits of the beginning address must be 0.

Figure 4–1. Data Memory Organization in an FIR Filter



In Example 4–1, N is 16 and the circular buffer starts at an address whose four LSBs are 0.

Example 4–1. FIR Implementation Using Circular Addressing Mode With a Multiply and Accumulate (MAC) Instruction

```
; TEXAS INSTRUMENTS INCORPORATED
   .mmregs
   .include  "main.inc"
; the 16 tap FIR coefficients
COFF_FIR_START   .sect  "coff_fir"    ; filter coefficients
   .word  6Fh
   .word  0F3h
   .word  269h
   .word  50Dh
   .word  8A9h
   .word  0C99h
   .word  0FF8h
   .word  11EBh
   .word  11EBh
   .word  0FF8h
   .word  0C99h
   .word  8A9h
   .word  50Dh
   .word  269h
   .word  0F3h
   .word  6Fh
COFF_FIR_END
FIR_DP           .usect    "fir_vars",0
d_filin          .usect    "fir_vars",1
d_filout         .usect    "fir_vars",1
fir_coff_table   .usect    "fir_coff",20
d_data_buffer    .usect    "fir_bfr",40  ; buffer size for the filter
                 .def      fir_init      ; initialize FIR filter
                 .def      fir_task      ; perform FIR filtering
;-------------------------------------------------------------------------
; Functional Description
;  This routine initializes circular buffers both for data and coeffs.
;-------------------------------------------------------------------------
          .asg      AR0, FIR_INDEX_P
          .asg      AR4,FIR_DATA_P
          .asg      AR5,FIR_COFF_P
          .sect     "fir_prog"
fir_init:
          STM       #fir_coff_table,FIR_COFF_P
          RPT       #K_FIR_BFFR-1                    ; move FIR coeffs from program
          MVPD      #COFF_FIR_START,*FIR_COFF_P+     ; to data
          STM       #K_FIR_INDEX,FIR_INDEX_P
          STM       #d_data_buffer,FIR_DATA_P        ; load cir_bfr address for the
                                                     ; recent samples
          RPTZ      A,#K_FIR_BFFR
          STL       A,*FIR_DATA_P+                   ; reset the buffer
          STM       #(d_data_buffer+K_FIR_BFFR-1), FIR_DATA_P
          RETD
          STM       #fir_coff_table, FIR_COFF_P
```

Example 4–1. FIR Implementation Using Circular Addressing Mode with a Multiply and Accumulate (MAC) Instruction (Continued)

```
;-------------------------------------------------------------------------
;  Functional Description
;
;   This subroutine performs FIR filtering using MAC instruction.
;   accumulator A (filter output) = h(n)*x(n–i) for i = 0,1...15
;-------------------------------------------------------------------------
       .asg      AR6,INBUF_P
       .asg      AR7,OUTBUF_P
       .asg      AR4,FIR_DATA_P
       .asg      AR5,FIR_COFF_P
       .sect     ”fir_prog”
fir_task:
;      LD        #FIR_DP,DP
       STM       #K_FRAME_SIZE–1,BRC          ; Repeat 256 times
       RPTBD     fir_filter_loop–1
       STM       #K_FIR_BFFR,BK               ; FIR circular bffr size
       LD        *INBUF_P+, A                 ; load the input value
fir_filter:
       STL       A,*FIR_DATA_P+%              ; replace oldest sample with newest
                                              ; sample
       RPTZ      A,(K_FIR_BFFR–1)
       MAC       *FIR_DATA_P+0%,*FIR_COFF_P+0%,A  ; filtering
       STH       A, *OUTBUF_P+                  ; replace the oldest bffr value
fir_filter_loop
    RET
```

In a second method, two features of the ’54x device facilitate implementation of the FIR filters: circular addressing and the FIRS instruction. The FIR filter shown in Figure 4–2, with symmetric impulse response about the center tap, is widely used in digital signal processing applications because of its linear phase response. In applications such as speech processing, linear phase response is required to avoid phase distortion, which degrades the quality of the signal waveforms. The output of the filter for length N is given by:

$$y(n) = \sum_{k=0}^{N/2-1} h(k)[x(n-k) + x(n-(N-1+k))] \qquad n = 0, 1, 2$$

Figure 4–2. Block Diagram of an Nth-Order Symmetric FIR Filter

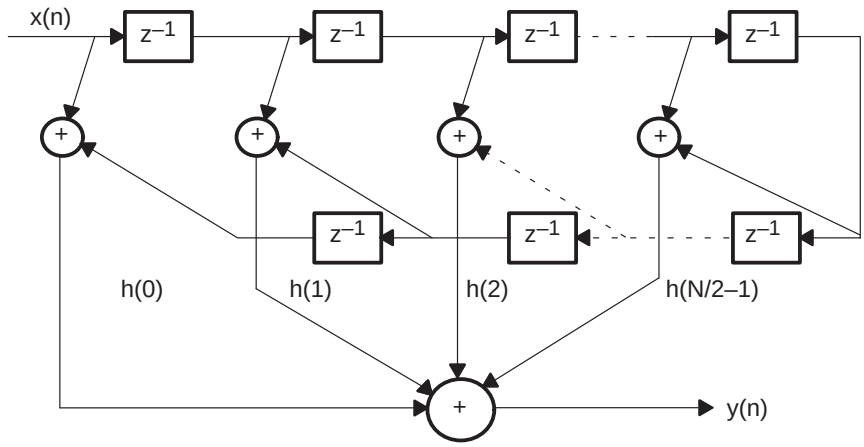
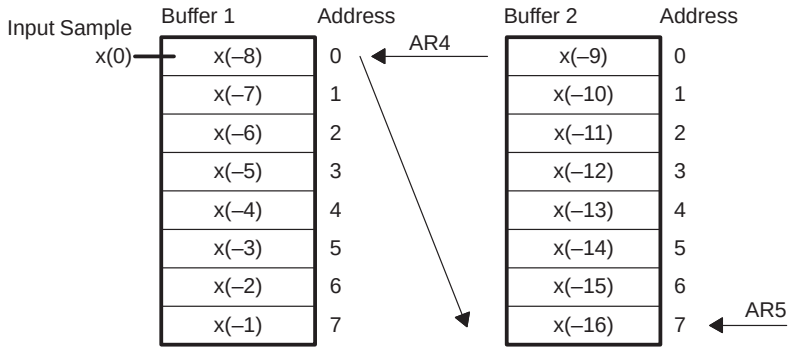


Figure 4–3 shows the storage diagram of the input sequence for two circular buffers. To build the buffers, the value of N/2 is loaded into a circular buffer size register. AR4 is set to point to the top of buffer 1 and AR5 points to the bottom of buffer 2. The data at the top of buffer 1 is moved to the bottom of buffer 2 for the delayed operation before storing new sample data in buffer 1. The processor then performs the adds and multiplies h(0){x(0)+x(–N+1)}. After each iteration of the filtering routine, AR4 points to the next time slot window for the data move and AR5 points to the next input sample. For the next iteration of the filtering routine, AR4 points to address 1 and AR5 points to address N/2–2.

Figure 4–3. Input Sequence Storage



Example 4–2 shows how to implement a symmetric FIR filter on the ’54x. It uses the symmetrical finite impulse response (FIRS) instruction and the repeat next instruction and clear accumulator (RPTZ) instruction together. FIRS can add two data values (input sequences stored in data memory) in parallel with multiplication of the previous addition result using an impulse response stored in program memory. FIRS becomes a single-cycle instruction when used with the single-repeat instruction. To perform the delayed operation in this storage scheme, two circular buffers are used for the input sequence.

Example 4–2. Symmetric FIR Implementation Using FIRS Instruction

```
; TEXAS INSTRUMENTS INCORPORATED
           .mmregs
           .include   ”main.inc”
FIR_COFF   .sect      ”sym_fir”                    ; filter coefficients
           .word      6Fh
           .word      0F3h
           .word      269h
           .word      50Dh
           .word      8A9h
           .word      0C99h
           .word      0FF8h
           .word      11EBh
d_datax_buffer        .usect  ”cir_bfr”,20
d_datay_buffer        .usect  ”cir_bfr1”,20
           .def       sym_fir_init                  ; initialize symmetric FIR
           .def       sym_fir_task
;-------------------------------------------------------------------------------
;          Functional Description
;          This routine initializes circular buffers both for data and coeffs.
;-------------------------------------------------------------------------------
           .asg       AR0,SYMFIR_INDEX_P
           .asg       AR4,SYMFIR_DATX_P
           .asg       AR5,SYMFIR_DATY_P
           .sect      ”sym_fir”
sym_fir_init:
           STM        #d_datax_buffer,SYMFIR_DATX_P  ; load cir_bfr address
                                                     ; for the 8 most
                                                     ; recent samples STM #d_datay
_buffer+K_FIR_BFFR/2–1,SYMFIR_DATY_P
                                                     ; load cir_bfr1 address
                                                     ; for the 8 old samples
           STM        #K_neg1,SYMFIR_INDEX_P         ; index offset –
                                                     ; whenever the pointer
                                                     ; hits the top of the bffr,
                                                     ; it automatically hits
                                                     ; bottom address of
                                                     ; buffer and decrements
                                                     ; the counter
           RPTZ       A,#K_FIR_BFFR
           STL        A,* SYMFIR_DATX_P+
           STM        #d_datax_buffer, SYMFIR_DATX_P
           RPTZ       A,#K_FIR_BFFR
           STL        A,* SYMFIR_DATY_P–
           RETD
           STM        #d_datay_buffer+K_FIR_BFFR/2–1, SYMFIR_DATY_P
```

Example 4–2. Symmetric FIR Implementation Using FIRS Instruction (Continued)

```
;-------------------------------------------------------------------------------
;  Functional Description
;This program uses the FIRS instruction to implement symmetric FIR filter
;Circular addressing is used for data buffers. The input scheme for the data;
;samples is divided into two circular buffers. The first buffer contains
;samples from X(–N/2) to X(–1) and the second buffer contains samples from
;X(–N) to X(–N/2–1).
;-------------------------------------------------------------------------------
           .asg       AR6,INBUF_P
           .asg       AR7,OUTBUF_P
           .asg       AR4,SYMFIR_DATX_P
           .asg       AR5,SYMFIR_DATY_P
           .sect      ”sym_fir”
sym_fir_task:
           STM        #K_FRAME_SIZE–1,BRC
           RPTBD      sym_fir_filter_loop–1
           STM        #K_FIR_BFFR/2,BK
           LD         *INBUF_P+, B
symmetric_fir:
           MVDD       *SYMFIR_DATX_P,*SYMFIR_DATY_P+0% ; move X(–N/2) to X(–N)
           STL        B,*SYMFIR_DATX_P     ; replace oldest sample with newest
                                           ; sample
           ADD        *SYMFIR_DATX_P+0%,*SYMFIR_DATY_P+0%,A   ; add X(0)+X(–N/2–1)
           RPTZ       B,#(K_FIR_BFFR/2–1)
           FIRS        *SYMFIR_DATX_P+0%,*SYMFIR_DATY_P+0%,FIR_COFF
           MAR        *+SYMFIR_DATX_P(2)%  ; to load the next newest sample
           MAR        *SYMFIR_DATY_P+%     ; position for the X(–N/2) sample
           STH        B, *OUTBUF_P+
sym_fir_filter_loop
           RET
           .end
```

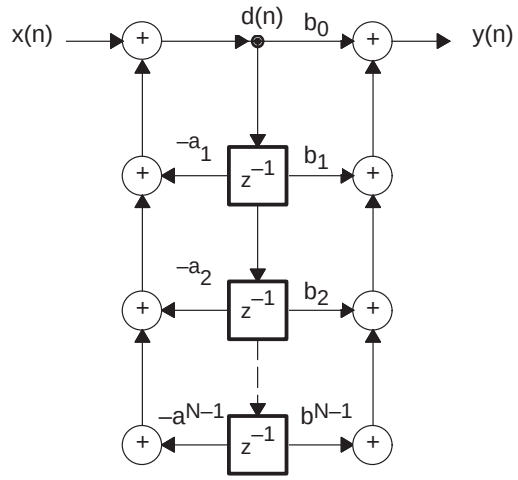
4.2 Infinite Impulse Response (IIR) Filters

IIR filters are widely used in digital signal processing applications. The transfer function of an IIR filter is given by:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}} = \frac{Y(z)}{x(z)}$$

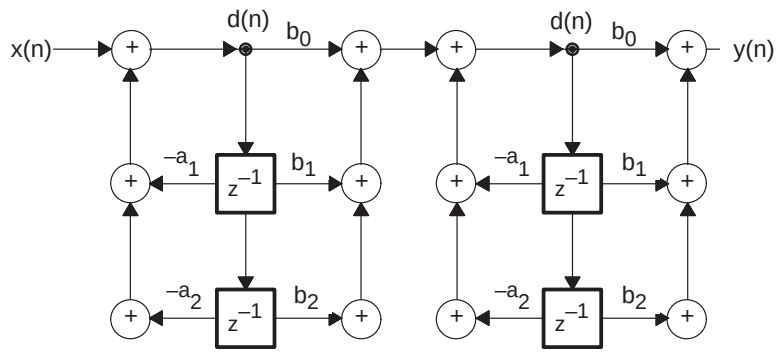
The transfer function has both poles and zeros. Its output depends on both input and past output. IIR filters need less computation than FIR filters. However, IIR filters have stability problems. The coefficients are very sensitive to coefficient quantization. Figure 4–4 shows a typical diagram of an IIR filter.

Figure 4–4. Nth-Order Direct-Form Type II IIR Filter



Most often, IIR filters are implemented as a cascade of second-order sections, called biquads. The block diagram is shown in Figure 4–5.

Figure 4–5. Biquad IIR Filter



Example 4–3. Two-Biquad Implementation of an IIR Filter

```
; TEXAS INSTRUMENTS INCORPORATED
          .mmregs
          .include         ”main.inc”
          .sect         ”iir_coff”
iir_table_start
*
*  second–order section # 01
*
          .word         –26778          ;A2
          .word         29529           ;A1/2
          .word         19381           ;B2
          .word         –23184          ;B1
          .word         19381           ;B0
*
*  second–order section # 02
*
          .word         –30497          ;A2
          .word         31131           ;A1/2
          .word         11363           ;B2
          .word         –20735          ;B1
          .word         11363           ;B0
iir_table_end
iir_coff_table          .usect ”coff_iir”,16
IIR_DP                  .usect ”iir_vars”,0
d_iir_d                 .usect ”iir_vars”,3*2
d_iir_y                 .usect ”iir_vars”,1
          .def          iir_init
          .def          iir_task
;-------------------------------------------------------------------------------
;  Functional Description
;   This routine initializes buffers both for data and coeffs.
;-------------------------------------------------------------------------------
          .asg          AR5,IIR_DATA_P      ; data samples pointer
          .asg          AR4,IIR_COFF_P      ; IIR filter coffs pointer
          .sect         ”iir
iir_init:
    STM #iir_coff_table,IIR_COFF_P
        RPT    #K_IIR_SIZE–1                ; move IIR coeffs from program
        MVPD   #iir_table_start,*IIR_COFF_P+ ; to data
;       LD     #IIR_DP,DP
        STM    #d_iir_d,IIR_DATA_P          ;AR5:d(n),d(n–1),d(n–2)
        RPTZ   A,#5                         ;initial d(n),d(n–1),d(n–2)=0
        STL    A,*IIR_DATA_P+
        RET
```

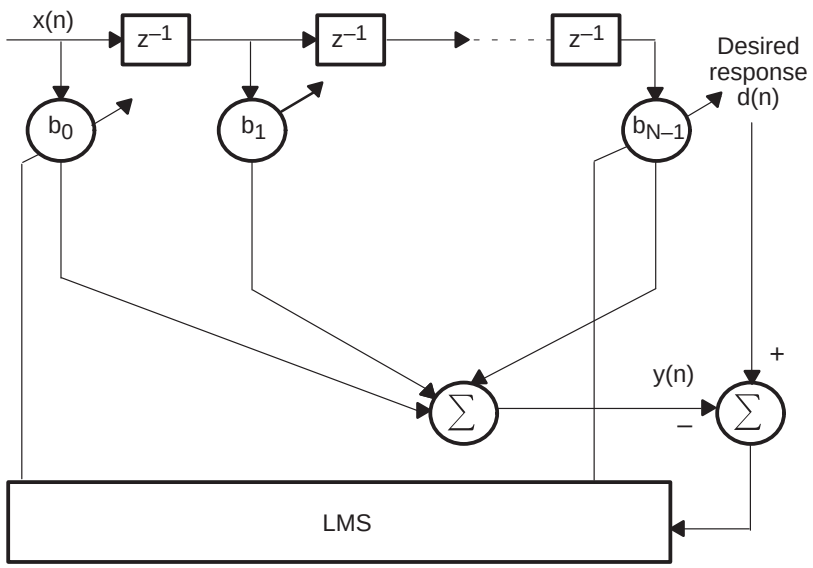
Example 4–3.Two-Biquad Implementation of an IIR Filter (Continued)

```
;  Functional Description
;
;      This subroutine performs IIR filtering using biquad sections
;      IIR Low pass filter design
;      Filter type : Elliptic Filter
;      Filter order : 4 order (cascade: 2nd order + 2nd order)
;      cut freq. of pass band : 200 Hz
;      cut freq. of stop band : 500
;                                        B0
;      ... -----> + ------> d(n) ---- x -> + -----.... -->
;      |          |                |
;      |    A1    |        B1      |
;      + <- x -- d(n-1) -- x -> +  |
;      |          |                |
;      |    A2    |        B2      |
;      + <- x -- d(n-2) -- x -> +
;
;          second order IIR
;-------------------------------------------------------------------------------
    .asg   AR5,IIR_DATA_P               ; data samples pointer
    .asg   AR4,IIR_COFF_P               ; IIR filter coffs pointer
    .asg   AR6,INBUF_P
    .asg   AR7,OUTBUF_P
    .asg   AR1,IIR_INDEX_P
    .sect  "iir"
iir_task:
    STM    #K_FRAME_SIZE-1,BRC          ; Perform filtering for 256 samples
    RPTB   iir_filter_loop-1
    LD     *INBUF_P+,8,A                ; load the input value
iir_filter:
    STM    #d_iir_d+5,IIR_DATA_P        ;AR5:d(n),d(n-1),d(n-2)
    MVPD   #iir_table_start,*IIR_COFF_P+ ; to data
    STM    #iir_coff_table,IIR_COFF_P   ;AR4:coeff of IIR filter A2,A1,B2,B1,B0
    STM    #K_BIQUAD-1,IIR_INDEX_P
feedback_path:
    MAC    *IIR_COFF_P+,*IIR_DATA_P-,A  ;input+d(n-2)*A2
    MAC    *IIR_COFF_P,*IIR_DATA_P,A    ;input+d(n-2)*A2+d(n-1)*A1/2
    MAC    *IIR_COFF_P+,*IIR_DATA_P-,A  ; A = A+d(n-1)*A1/2
    STH    A,*IIR_DATA_P+               ;d(n) = input+d(n-2)*A2+d(n-1)*A1
    MAR    *IIR_DATA_P+
* Forward path
    MPY    *IIR_COFF_P+,*IIR_DATA_P-,A  ;d(n-2)*B2
    MAC    *IIR_COFF_P+,*IIR_DATA_P,A   ;d(n-2)*B2+d(n-1)*B1
    DELAY  *IIR_DATA_P-                 ;d(n-2)=d(n-1)
eloop:
    BANZD  feedback_path, *IIR_INDEX_P-
    MAC    *IIR_COFF_P+,*IIR_DATA_P,A   ;d(n-2)*B2+d(n-1)*B1+d(n)*B0
    DELAY  *IIR_DATA_P-                 ;d(n-1)=d(n)
    STH    A,d_iir_y                    ;output=d(n-2)*B2+d(n-1)*B1+d(n)*B0
    LD     d_iir_y,2,A                  ; scale the output
    STL    A, *OUTBUF_P+                ; replace the oldest bffr value
iir_filter_loop
    RET
    .end
```

4.3 Adaptive Filtering

Some applications for adaptive FIR and IIR filtering include echo and acoustic noise cancellation. In these applications, an adaptive filter tracks changing conditions in the environment. Although in theory, both FIR and IIR structures can be used as adaptive filters, stability problems and the local optimum points of IIR filters makes them less attractive for this use. FIR filters are used for all practical applications. The LMS, ST||MPY, and RPTBD instructions on the ’54x can reduce the execution time of code for adaptive filtering. The block diagram of an adaptive FIR filter is shown in Figure 4–6. The Adaptive filtering routine is shown in Example 4–4, page 4-14.

Figure 4–6. Adaptive FIR Filter Implemented Using the Least-Mean-Squares (LMS) Algorithm



On the ’54x, one coefficient can be adapted by using the least-mean-squares (LMS) algorithm, which is given by

$$b_k(i+1) = b_k(i) + 2\beta e(i)\mathrm{x(i-k)},$$

where:

$e(i) = d(i) - y(i)$

The output of the adaptive filter is given by

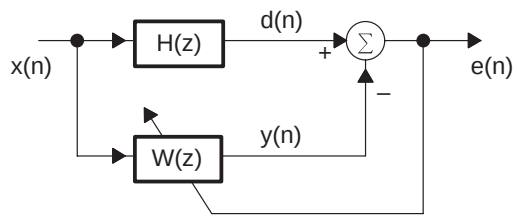
$$y(i) = \sum_{k=0}^{N-1} b_k\, x\,(i-k)$$

The LMS instruction can perform a MAC instruction and an addition with rounding in parallel. The LMS algorithm calculates the filter output and updates each coefficient in the filter in parallel by using the LMS instruction, along with the ST||MPY and RPTBD instructions. For each coefficient in the filter at a given instant, 2βe(i) is a constant. This factor can be computed once and stored in the temporary register, T, to use in each of the updates. The ST||MPY instruction multiplies a data sample by this factor, then the LMS instruction updates a coefficient in the filter and accumulates the filtered output. Since the factor is stored in T, the adaptive filtering in a time-slot window is performed in N cycles.

An adaptive filter can be used in modeling to imitate the behavior of a physical dynamic system. Figure 4–7 shows a block diagram of system identification, where the adaptive filter adjusts itself to cause its output to match that of the unknown system. H(z) is the impulse response of an unknown system; for example, the echo response in the case of an acoustic echo cancellation system. The signal x(n) trains the system. The size of the adaptive filter is chosen to be N, where N is the number of taps (coefficients) of the known system, W(z).

Two circular buffers store the input sequence. AR3 points to the first buffer, AR2 points to the coefficients of H(z), AR4 points to the coefficients of W(z), and AR5 points to the second buffer. The newest sample is stored in a memory location that is input to the adaptive filter. The input sample is subtracted from the output of the adaptive filter to obtain the error data for the LMS algorithm. In this example, the adaptive filter output is computed for the newest sample and the filter coefficients are updated using the previous sample. Thus, there is an inherent delay between the update of the coefficient and the output of the adaptive filter.

Figure 4–7. System Identification Using Adaptive Filter



Example 4–4. System Identification Using Adaptive Filtering Techniques

```
; TEXAS INSTRUMENTS INCORPORATED
          .mmregs
          .include  ”main.inc”
scoff     .sect     ”coeffh”
          .include  ”impulse.h”
          ADAPT_DP  .usect  ”adpt_var”,0
d_primary           .usect  ”adpt_var”,1
d_output            .usect  ”adpt_var”,1
d_error             .usect  ”adpt_var”,1
d_mu                .usect  ”adpt_var”,1
d_mu_e              .usect  ”adpt_var”,1
d_new_x             .usect  ”adpt_var”,1
d_adapt_count       .usect  ”adpt_var”,1
hcoff               .usect ”bufferh”, H_FILT_SIZE    ; H(z) coeffs
wcoff               .usect ”bufferw”, ADPT_FILT_SIZE  W(z) coeffs
xh                  .usect ”bufferx”, H_FILT_SIZE     ; input data to H(z)
xw                  .usect ”bufferp”, ADPT_FILT_SIZE ; input data–adaptive filter
             .def          adapt_init,adapt_task
;-------------------------------------------------------------------------
;
;  Functional Description
;
;  This subroutine moves filter coefficients from program to data space.
;  Initializes the adaptive coefficients, buffers, vars, and sets the circular
;  buffer address for processing.
;-------------------------------------------------------------------------
   .asg      AR0,INDEX_P
   .asg      AR1,INIT_P                          ; initialize buffer pointer
   .asg      AR3,XH_DATA_P                       ; data coeff buffer pointer
   .asg      AR5,XW_DATA_P                       ; data coeff buffer pointer
                                                 ; for cal.y output
   .sect ”filter”
adapt_init:
;  initialize input data location, input to hybrid, with zero.
   STM    #xh,INIT_P
   RPTZ   A,#H_FILT_SIZE–1
   STL    A,*INIT_P+
;  initialize input data location, input to adaptive filter, with Zero.
   STM    #xw,INIT_P
   RPTZ   A,#ADPT_FILT_SIZE–1
   STL    A,*INIT_P+
;  initialize adaptive coefficient with Zero.
   STM    #wcoff,INIT_P
   RPTZ   A,#ADPT_FILT_SIZE–1
   STL    A,*INIT_P+
;  initialize temporary storage locations with zero
   STM    #d_primary,INIT_P
   RPTZ   A,#6
   STL    A,*INIT_P+
;  copy system coefficient into RAM location, Reverse order STM #hcoff,INIT_P
   RPT    #H_FILT_SIZE–1
   MVPD   #scoff,*INIT_P+
```

Example 4–4. System Identification Using Adaptive Filtering Techniques (Continued)

```
;   LD  #ADAPT_DP,DP                  ;set DP now and not worry about it
    ST  #K_mu,d_mu
    STM #1,INDEX_P                    ; increment value to be used by
                                      ; dual address
;   associate auxiliary registers for circular computation
    STM #xh+H_FILT_SIZE–1,XH_DATA_P   ; last input of hybrid buffer
    RETD
    STM #xw+ADPT_FILT_SIZE–1,XW_DATA_P ;last element of input buffer
;-------------------------------------------------------------------------------
;
;   Functional Description
;
;   This subroutine performs the adaptive filtering. The newest sample is stored
;   in a separate location since filtering and adaptation are performed at the
;   same time. Otherwise the oldest sample is over written before updating
;   the w(N–1) coefficient.
;
;   d_primary = xh *hcoff
;   d_output = xw *wcoff
;   LMS algorithm:
;   w(i+1) = w(i)+d*mu_error*xw(n–i) for i = 0,1,...127 and n = 0,1,.....
;-------------------------------------------------------------------------------
    .asg   AR2,H_COFF_P               ; H(Z) coeff buffer pointer
    .asg   AR3,XH_DATA_P              ; data coeff buffer pointer
    .asg   AR6,INBUF_P                ; input buffer address pointer
    .asg   AR7,OUTBUF_P               ; output buffer address pointer
                                      ; for cal. primary input
    .asg   AR4,W_COFF_P               ; W(z) coeff buffer pointer
    .asg   AR5,XW_DATA_P              ; data coeff buffer pointer
    .sect  ”filter”
adapt_task:
    STM    #H_FILT_SIZE,BK            ; first circular buffer size
    STM    #hcoff,H_COFF_P            ; H_COFF_P –-> last of sys coeff
    ADDM   #1,d_adapt_count
    LD     *INBUF_P+, A               ; load the input sample
    STM    #wcoff,W_COFF_P            ; reset coeff buffer
    STL    A,d_new_x                  ; read in new data
    LD     d_new_x,A                  ;
    STL    A,*XH_DATA_P+0%            ; store in the buffer
    RPTZ   A,#H_FILT_SIZE–1           ; Repeat 128 times
    MAC    *H_COFF_P+0%,*XH_DATA_P+0%,A  ; mult & acc:a = a + (h * x)
    STH    A,d_primary                ; primary signal
;   start simultaneous filtering and updating the adaptive filter here.
    LD     d_mu_e,T                   ; T = step_size*error
    SUB    B,B                        ; zero acc B
    STM    #(ADPT_FILT_SIZE–2),BRC    ; set block repeat counter
    RPTBD  lms_end–1
    MPY    *XW_DATA_P+0%, A           ; error * oldest sample
    LMS    *W_COFF_P, *XW_DATA_P      ; B = filtered output (y)
                                      ; Update filter coeff
    ST     A, *W_COFF_P+              ; save updated filter coeff
    || MPY *XW_DATA_P+0%,A            ; error *x[n–(N–1)]
```

Example 4–4. System Identification Using Adaptive Filtering Techniques (Continued)

```
   LMS *W_COFF_P, *XW_DATA_P          ; B = accum filtered output y
                                      ; Update filter coeff
lms_end
   STH    A, *W_COFF_P                ; final coeff
   MPY    *XW_DATA_P,A                ; x(0)*h(0)
   MVKD   #d_new_x,*XW_DATA_P         ; store the newest sample
   LMS    *W_COFF_P,*XW_DATA_P+0%
   STH    B, d_output                 ; store the filtered output
   LD     d_primary,A
   SUB    d_output,A
   STL    A, d_error                  ; store the residual error signal
   LD     d_mu,T
   MPY    d_error,A                   ; A=u*e
   STH    A,d_mu_e                    ; save the error *step_size
   LD     d_error,A                   ; residual error signal
   STL    A, *OUTBUF_P+
   LD     #K_FRAME_SIZE,A             ; check if a frame of samples
   SUB    d_adapt_count,A             ; have been processed
   BC     adapt_task,AGT
   RETD
   ST     #K_0,d_adapt_count          ; restore the count
   .end
* This is an input file used by the adaptive filter program.
* The transfer function is the system to be identified by the adaptive filter
   .word  0FFFDh
   .word  24h
   .word  6h
   .word  0FFFDh
   .word  3h
   .word  3h
   .word  0FFE9h
   .word  7h
   .word  12h
   .word  1Ch
   .word  0FFF3h
   .word  0FFE8h .word  0Ch
   .word  3h
   .word  1Eh
   .word  1Ah
   .word  22h
   .word  0FFF5h
   .word  0FFE5h
   .word  0FFF1h
   .word  0FFC5h
   .word  0Ch
   .word  0FFE8h
   .word  37h
   .word  0FFE4h
   .word  0FFCAh
   .word  1Ch
   .word  0FFFDh
   .word  21h
   .word  0FFF7h
```

Example 4–4. System Identification Using Adaptive Filtering Techniques (Continued)

```
.word  2Eh
.word  28h
.word  0FFC6h
.word  53h
.word  0FFB0h
.word  55h
.word  0FF36h
.word  5h
.word  0FFCFh
.word  0FF99h
.word  64h
.word  41h
.word  0FFF1h
.word  0FFDFh
.word  0D1h
.word  6Ch
.word  57h
.word  36h
.word  0A0h
.word  0FEE3h
.word  6h
.word  0FEC5h
.word  0ABh
.word  185h
.word  0FFF6h
.word  93h
.word  1Fh
.word  10Eh
.word  59h
.word  0FEF0h
.word  96h
.word  0FFBFh
.word  0FF47h
.word  0FF76h
.word  0FF0Bh
.word  0FFAFh
.word  14Bh
.word  0FF3Bh
.word  132h
.word  289h
.word  8Dh
.word  0FE1Dh
.word  0FE1Bh
.word  0D4h
.word  0FF69h
.word  14Fh
.word  2AAh
.word  0FD43h
.word  0F98Fh
.word  451h
.word  13Ch
.word  0FEF7h
.word  0FE36h
```

Example 4–4. System Identification Using Adaptive Filtering Techniques (Continued)

```
    .word  80h
    .word  0FFBBh
    .word  0FC8Eh
    .word  10Eh
    .word  37Dh
    .word  6FAh
    .word  1h
    .word  0FD89h
    .word  198h
    .word  0FE4Ch
    .word  0FE78h
    .word  0F215h
    .word  479h
    .word  749h
    .word  289h
    .word  0F667h
    .word  304h
    .word  5F8h
    .word  34Fh
    .word  47Bh
    .word  0FF7Fh
    .word  85Bh
    .word  0F837h
    .word  0F77Eh
    .word  0FF80h
    .word  0B9Bh
    .word  0F03Ah
    .word  0EE66h
    .word  0FE28h
    .word  0FAD0h
    .word  8C3h
    .word  0F5D6h
    .word  14DCh
    .word  0F3A7h
    .word  0E542h
    .word  10F2h
    .word  566h
    .word  26AAh
    .word  15Ah
    .word  2853h
    .word  0EE95h
    .word  93Dh
    .word  20Dh
    .word  1230h
    .word  238Ah
```

4.4 Fast Fourier Transforms (FFTs)

FFTs are an efficient class of algorithms for the digital computation of the N-point discrete Fourier transform (DFT). In general, their input sequences are assumed to be complex. When input is purely real, their symmetric properties compute the DFT very efficiently.

One such optimized real FFT algorithm is the packing algorithm. The original 2N-point real input sequence is packed into an N-point complex sequence. Next, an N-point FFT is performed on the complex sequence. Finally the resulting N-point complex output is unpacked into the 2N-point complex sequence, which corresponds to the DFT of the original 2N-point real input.

Using this strategy, the FFT size can be reduced by half, at the FFT cost function of O(N) operations to pack the input and unpack the output. Thus, the real FFT algorithm computes the DFT of a real input sequence almost twice as fast as the general FFT algorithm. The following subsections show how to perform a 16-point real FFT (2N = 16).

4.4.1 Memory Allocation for Real FFT Example

The memory organization for the real FFT example in Chapter10, *Application Code Examples*, uses the memory configuration shown in Figure 4–8 on page 4-20.

The following tables give the organization of values in data memory from the beginning of the real FFT algorithm to its end. Initially, the original 2N-point real input sequence, a(n), is stored in the lower half of the 4N-word data processing buffer, as shown in Figure 4–9 on page 4-21.

Figure 4–8. Memory Allocation for Real FFT Example

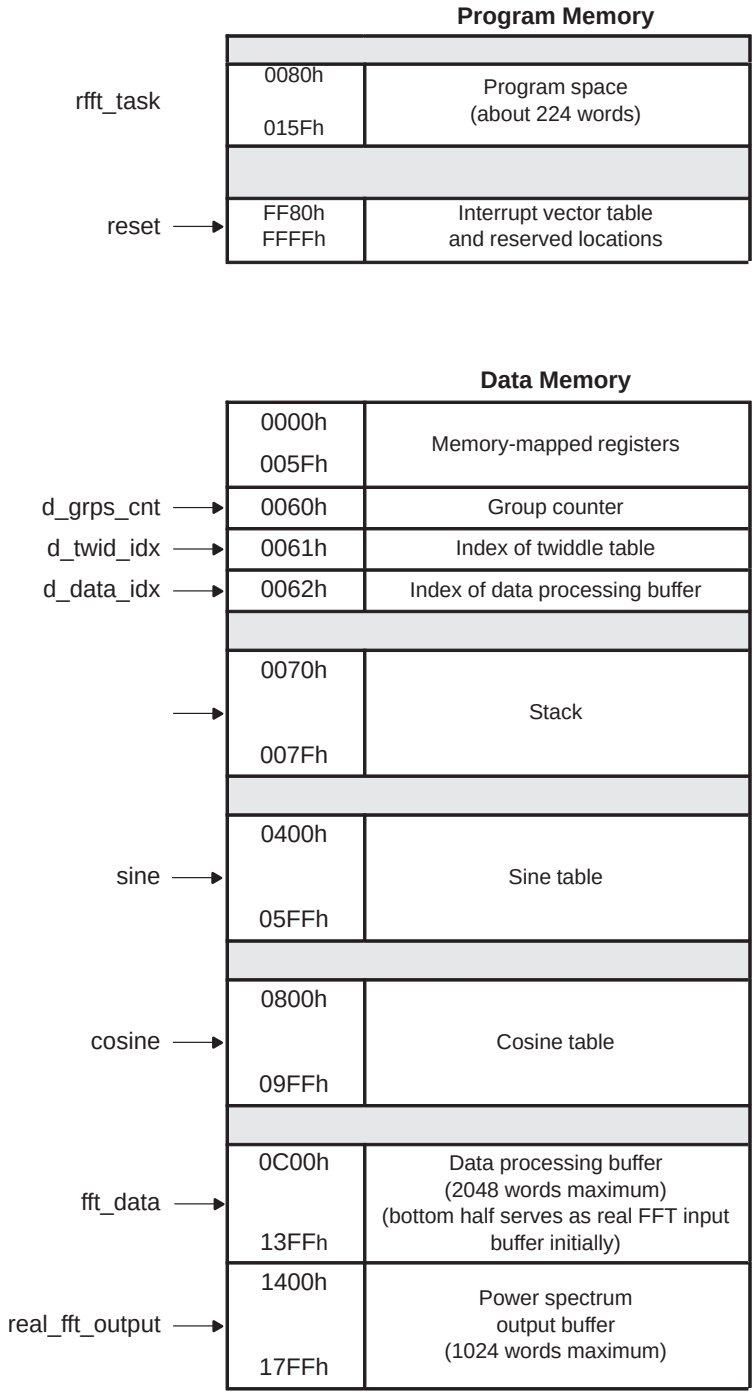


Figure 4–9. Data Processing Buffer

	Data Memory
0C00h	
0C01h	
0C02h	
0C03h	
0C04h	
0C05h	
0C06h	
0C07h	
0C08h	
0C09h	
0C0Ah	
0C0Bh	
0C0Ch	
0C0Dh	
0C0Eh	
0C0Fh	
0C10h	a(0)
0C11h	a(1)
0C12h	a(2)
0C13h	a(3)
0C14h	a(4)
0C15h	a(5)
0C16h	a(6)
0C17h	a(7)
0C18h	a(8)
0C19h	a(9)
0C1Ah	a(10)
0C1Bh	a(11)
0C1Ch	a(12)
0C1Dh	a(13)
0C1Eh	a(14)
0C1Fh	a(15)

4.4.2 Real FFT Example

The ’54x real FFT algorithm is a radix-2, in-place DFT algorithm. It is shown in the following subsections in four phases :

1) Packing and bit-reversal of input
2) N-point complex FFT
3) Separation of odd and even parts
4) Generation of final output

Initially, any real input sequences of 16 to 1024 points can be used by simply modifying the constants K_FFT_SIZE and K_LOGN appropriately, defined in file main.inc. (The real input size is described as 2N and the FFT size in phase two as N.) For a 256-point real input, for example, K_FFT_SIZE must be set to 128, not 256, and K_LOGN must be 7, not 8. Input data is assumed to be in Q15 format.

4.4.2.1 Phase 1: Packing and Bit-Reversal of Input

In phase 1, the input is bit-reversed so that the output at the end of the entire algorithm is in natural order. First, the original 2N-point real input sequence is copied into contiguous sections of memory labeled real_fft_input and interpreted as an N-point complex sequence, d[n]. The even-indexed real inputs form the real part of d[n] and the odd-indexed real inputs form the imaginary part. This process is called packing. (n is a variable indicating time and can vary from 0 to infinity, while N is a constant). Next, this complex sequence is bit-reversed and stored into the data processing buffer, labeled fft_data.

1) Arrange the real input sequence, a(n) for n = 0, 1, 2, ... n – 1, as shown in Figure 4–9. Divide a(n) into two sequences as shown in Figure 4–10. The first is the original input sequence from 0C10h to 0C1Fh. The other is a packed sequence:

   for n = 0, 1, 2, ... , N – 1

2) Form the complex FFT input, d(n), by using r(n) for the real part and i(n) for the imaginary part:

   $d(n) = r(n) + j\, i(n)$

3) Store d(n) in the upper half of the data processing buffer in bit-reversed order as shown in Figure 4–10 on page 4-23.

Figure 4–10. Phase 1 Data Memory

0C00h	r(0) = a(0)
0C01h	i(0) = a(1)
0C02h	r(4) = a(8)
0C03h	i(4) = a(9)
0C04h	r(2) = a(4)
0C05h	i(2) = a(5)
0C06h	r(6) = a(12)
0C07h	i(6) = a(13)
0C08h	r(1) = a(2)
0C09h	i(1) = a(3)
0C0Ah	r(5) = a(10)
0C0Bh	i(5) = a(11)
0C0Ch	r(3) = a(6)
0C0Dh	i(3) = a(7)
0C0Eh	r(7) = a(14)
0C0Fh	i(7) = a(15)
0C10h	a(0)
0C11h	a(1)
0C12h	a(2)
0C13h	a(3)
0C14h	a(4)
0C15h	a(5)
0C16h	a(6)
0C17h	a(7)
0C18h	a(8)
0C19h	a(9)
0C1Ah	a(10)
0C1Bh	a(11)
0C1Ch	a(12)
0C1Dh	a(13)
0C1Eh	a(14)
0C1Fh	a(15)

4.4.2.2 Phase 2: N-Point Complex FFT

In phase 2, an N-point complex FFT is performed in place in the data-processing buffer. The twiddle factors are in Q15 format and are stored in two separate tables, pointed to by sine and cosine. Each table contains 512 values, corresponding to angles ranging from 0 to almost 180 degrees. The indexing scheme used in this algorithm permits the same twiddle tables for inputs of different sizes. Since circular addressing indexes the tables, the starting address of each table must line up to an address with 0s in the eight LSBs.

1) Perform an N-point complex FFT on d(n). The resulting sequence is

   $D[k] = F\{d(n)\} = R[k] + j\, I[k]$

   where R[k] and I[k] are the real and imaginary parts of D[k], respectively.

2) Since the FFT computation is done in place, the resulting sequence, D[k], occupies the upper half of the data-processing buffer, as shown. The lower half of the data processing buffer still contains the original real input sequence, a(n). This is overwritten in phase 3.

3) All the information from the original 2N-point real sequence, a(n), is contained in this N-point complex sequence, D[k]. The remainder of the algorithm unpacks D[k] into the final 2N-point complex sequence, A[k] = F{a(n)}.

Figure 4–11. Phase 2 Data Memory

0C00h	R[0]
0C01h	I[0]
0C02h	R[1]
0C03h	I[1]
0C04h	R[2]
0C05h	I[2]
0C06h	R[3]
0C07h	I[3]
0C08h	R[4]
0C09h	I[4]
0C0Ah	R[5]
0C0Bh	I[5]
0C0Ch	R[6]
0C0Dh	I[6]
0C0Eh	R[7]
0C0Fh	I[7]
0C10h	a(0)
0C11h	a(1)
0C12h	a(2)
0C13h	a(3)
0C14h	a(4)
0C15h	a(5)
0C16h	a(6)
0C17h	a(7)
0C18h	a(8)
0C19h	a(9)
0C1Ah	a(10)
0C1Bh	a(11)
0C1Ch	a(12)
0C1Dh	a(13)
0C1Eh	a(14)
0C1Fh	a(15)

4.4.2.3 Phase 3: Separation of Odd and Even Parts

Phase 3 separates the FFT output to compute four independent sequences: RP, RM, IP, and IM, which are the even real, odd real, even imaginary, and the odd imaginary parts, respectively.

1) D[k] is separated into its real even part, RP[k], real odd part, RM[k], imaginary even part, IP[k], and imaginary odd part, IM[k,] according to the following equations:

   $RP[k] = RP[N-k] = 0.5 * (R[k] + R[N-k])$

   $RM[k] = -RM[N-k] = 0.5 * (R[k] - R[N-k])$

   $IP[k] = IP[N-k] = 0.5 * (I[k] + I[N-k])$

   $IM[k] = -IM[N-k] = 0.5 * (I[k] - I[N-k])$

   $RP[0] = R[0]$

   $IP[0] = I[0]$

   $RM[0] = IM[0] = RM[N/2] = IM[N/2] = 0$

   $RP[N/2] = R[N/2]$

   $IP[N/2] = I[N/2]$

2) The table below shows the organization of the values at the end of phase three. The sequences RP[k] and IP[k] are stored in the upper half of the data processing buffer in ascending order; the sequences RM[k] and IM[k] are stored in the lower half in descending order.

Figure 4–12. Phase 3 Data Memory

0C00h	RP[0] = R[0]
0C01h	IP[0] = I[0]
0C02h	RP[1]
0C03h	IP[1]
0C04h	RP[2]
0C05h	IP[2]
0C06h	RP[3]
0C07h	IP[3]
0C08h	RP[4] = R[4]
0C09h	IP[4] = I[4]
0C0Ah	RP[5]
0C0Bh	IP[5]
0C0Ch	RP[6]
0C0Dh	IP[6]
0C0Eh	RP[7]
0C0Fh	IP[7]
0C10h	a(0)
0C11h	a(1)
0C12h	IM[7]
0C13h	RM[7]
0C14h	IM[6]
0C15h	RM[6]
0C16h	IM[5]
0C17h	RM[5]
0C18h	IM[4] = 0
0C19h	RM[4] = 0
0C1Ah	IM[3]
0C1Bh	RM[3]
0C1Ch	IM[2]
0C1Dh	RM[2]
0C1Eh	IM[1]
0C1Fh	RM[1]

4.4.2.4 Phase 4: Generation of Final Output

Phase 4 performs one more set of butterflies to generate the 2N-point complex output, which corresponds to the DFT of the original 2N-point real input sequence. The output resides in the data processing buffer.

1) The four sequences, RP[k], RM[k], IP[k], and IM[k], are used to compute the real FFT of a(n) according to the following equations.

$$AR[k] = AR[2N - k] = RP[k] + cos(k\,\pi\,/N) * IP[k] - sin(k\,\pi\,/N) * RM[k]$$

$$AI[k] = -AI[2N - k] = IM[k] - cos(k\,\pi\,/N) * RM[k] - sin(k\,\pi\,/N) * IP[k]$$

$$AR[0] = RP[0] + IP[0]$$

$$AI[0] = IM[0] - RM[0]$$

$$AR[N] = R[0] - I[0]$$

$$AI[N] = 0$$

where:

$$A[k] = A[2N-k] = AR[k] + j\,AI[k] = F\{a(n)\}$$

2) The real FFT outputs fill up the entire 4N-word data processing buffer. These outputs are real/imaginary, interleaved, and in natural order, as shown in Figure 4–13 on page 4-29. The values RM[0] and IM[0] are not stored because they are not used to compute the final outputs in phase 4.

Figure 4–13. Phase 4 Data Memory

0C00h	AR[0]
0C01h	AI[0]
0C02h	AR[1]
0C03h	AI[1]
0C04h	AR[2]
0C05h	AI[2]
0C06h	AR[3]
0C07h	AI[3]
0C08h	AR[4]
0C09h	AI[4]
0C0Ah	AR[5]
0C0Bh	AI[5]
0C0Ch	AR[6]
0C0Dh	AI[6]
0C0Eh	AR[7]
0C0Fh	AI[7]
0C10h	AR[8]
0C11h	AI[8]
0C12h	AR[9]
0C13h	AI[9]
0C14h	AR[10]
0C15h	AI[10]
0C16h	AR[11]
0C17h	AI[11]
0C18h	AR[12]
0C19h	AI[12]
0C1Ah	AR[13]
0C1Bh	AI[13]
0C1Ch	AR[14]
0C1Dh	AI[14]
0C1Eh	AR[15]
0C1Fh	AI[15]

Chapter 5

Resource Management

This chapter introduces features of the ’54x that improve system performance. These features allow you to conserve power and manage memory. You can improve the performance of any application through efficient memory management. Some issues include:

- On-chip memory versus off-chip memory
- Random access variables that use direct memory addressing versus aggregate variables that include structures/arrays
- The use of pointers for accessing the arrays and pointers
- Alignment of long words to even addresses
- The K-boundary requirement for circular buffers
- Allocation of stack

This chapter also discusses unique features of the ’548 and ’LC548 that help when an application needs a large amount of memory.

Topic **Page**

5.1 Memory Allocation

The ’54x can access a large amount of program and data memory (64K words each), but can handle only a limited amount of on-chip memory. On-chip memory accesses reduce the cycle time, since there are eight different internal buses on the ’54x but there is only one external bus for off-chip accesses. This means that off-chip operation requires more cycles to perform an operation than on-chip operation.

The DSP uses wait-state generators to interface to slower memories. The system, then, cannot run at full speed. If on-chip memory consists of dual access RAM (DARAM), accessing two operands from the same block does not incur a penalty. Using single access RAM (SARAM), however, incurs a cycle penalty. You can use on-chip ROM for tables to make efficient use of memory.

Random-access variables use direct addressing mode. This allocates all the random variables on a single data page, using one data page initialization for the application. Data-page relative memory addressing makes efficient use of memory resources. Each data variable has an associated lifetime. When that lifecycle is over, the data variable ceases to exist. Thus, if two data variables have non-overlapping lifetimes, both can occupy the same physical memory. All random variables, then, can form unions in the linker command file.

The actual lifetime of a variable determines whether it is retained across the application or only in the function. By careful organization of the code memory, resources can be used optimally. Aggregate variables, such as arrays and structures, are accessed via pointers located within that program’s data page. Aggregate variables reside elsewhere in memory. Depending upon the lifetime of the arrays or structures, these can also form unions accordingly.

Memory management is required for interrupt-driven tasks. Often, programmers assume that all CPU resources are available when required. This may not be the case if tasks are interrupted periodically. These interrupts do not require many CPU resources, but they force the system to respond within a certain time. To ensure that interrupts occur within the specified time and the interrupted code resumes as soon as possible, you must use low overhead interrupts. If the application requires frequent interrupts, you can use some of the CPU resources for these interrupts. For example, when all CPU resources are used, simply saving and restoring the CPU’s contents increases the overhead for an interrupt service routine (ISR).

A dedicated auxiliary register is useful for servicing interrupts. Allowing interrupts at certain places in the code permits the various tasks of an application to reuse memory. If the code is fully interruptible (that is, interrupts can occur anywhere and interrupt response time is assured within a certain period),

memory blocks must be kept separate from each other. On the other hand, if a context switch occurs at the completion of a function rather than in the middle of execution, the variables can be overlapped for efficiency. This allows variables to use the same memory addresses at different times.

Long words must be aligned at even boundaries for double-precision operations; that is, the most significant word at an even address and the least significant word at an odd address. Circular buffers start at a K boundary, where K is the smallest integer that satisfies $2^K > R$ and R is the size of the circular buffer. If an application uses circular buffers of different sizes, you must use the align directive to align the buffers to correct sizes. You can do this by allocating the largest buffer size as the first alignment, the next highest as the second alignment, and so forth. Example 5–1 shows the memory management alignment feature where the largest circular buffer is 1024 words, and therefore, is assigned first. A 256-word buffer is assigned next. Unused memory can be used for other functions without conflict.

Example 5–1. Memory Management

```
DRAM      : origin = 0x0100, length = 0x1300
inpt_buf  : {} >  DRAM,align(1024)PAGE 1
outdata   : {} >  DRAM,align(1024)PAGE 1
UNION     :    >  DRAM align(1024)  PAGE 1
{
   fft_bffr
   adpt_sct:
   {
   *(bufferw)
   .+=80h;
   *(bufferp)
   }
}
UNION     :    >  DRAM align(256) PAGE 1
{
   fir_bfr
   cir_bfr
   coff_iir
   bufferh
   twid_sin
}

UNION     :    >  DRAM align(256) PAGE 1
{
   fir_coff
   cir_bfr1
   bufferx
   twid_cos
}
```

Stack allocation can also benefit from efficient memory management. The stack grows from high to low memory addresses. The stack pointer (SP) decrements the stack by 1 before pushing its contents, which must be preserved, onto the stack and post increments after a pop. The bottom location is added to the stack, giving the actual stack size. The last element is always empty. Whether the stack is on chip or off chip affects the cycle count for accessing data.

Example 5–2 shows stack initialization when the application is written in assembly. The variable SYSTEM_STACK holds the size of the stack. It is loaded into the SP, which points to the end of the stack. The predecrement during the push operation and the postincrement during the pop cannot overflow the stack. Example 5–3 shows stack initialization when the application is written in C.

Example 5–2. Stack Initialization for Assembly Applications

```
K_STACK_SIZE      .set      100
STACK             .usect    “stack”, K_STACK_SIZE
SYSTEM_STACK      .set      STACK+K_STACK_SIZE
       .ref       SYSTEM_STACK
       STM        #SYSTEM_STACK, SP    ; initialization
                                       ; of SP– this is done
                                       ; vectors.asm
stack  : {} DRAM        PAGE 1         ; initialization of stack
                                       ; in linker command file
```

The compiler uses a stack to allocate local variables, pass arguments, and save the processor status. The stack size is set by the linker and the default size is 1 K words. In Example 5–3, the .stack section creates a stack size of 1 K words. A section of 100 words is created, referenced as top_stck and btm_stck, for the CPU requirements. The rest of the stack (1024 – 100) words can be used for passing arguments and local variables. Only the btm_stck is referenced in the code; hence, several sections can be created within the 1 K words of the stack.

Example 5–3. Stack Initialization for C Applications

```
    .ref   btm_stck      ; bottom of stack label
    STM    #btm_stck, SP ; initialization of SP – this is done
                         ; vectors.asm
.stack :
    {
    top_stck =.;         /* top of stack */
    .+=100;              /* size of the stack */
    btm_stck =.;         /* this is done in linker command
    }                    file */
```

5.2 Overlay Management

Some systems use a memory configuration in which all or part if the memory space is overlaid. This allows the system to map different banks of physical memory into and out of a single address range. Multiple banks of physical memory can overlay each other at one address range. This is achieved by setting the OVLY bit in the PMST register. This is particularly useful in loading the coefficients of a filter, since program and data use the same physical memory.

If an application needs more than 64K words of either data or program memory, two options are available. The first extends the 16-bit address line to a 16 + n-address line for the extended memory space. The ’548 provides 16 + 7 address lines to access 8M words of program space. The other option uses an external device that provides upper addresses beyond the 16-bit memory range. The DSP writes a value to a register located in its I/O space, whose data lines are the higher address bits. It implements bank switching to cross the 64K boundary. Since the bank switch requires action from the DSP, frequent switching between the banks is not very efficient. It is more efficient to partition tasks within a bank and switch banks only when starting new tasks.

The ’LC548 is designed to support a much larger program space of 8M words. Its memory-mapped register controls paging, and its extra instructions address extended program space. The OVLY bit configures the 8M words for on- or off-chip memory. If OVLY = 1, the lower half (128 pages) is a shared, on-chip, 32K-word block and the remaining 4M words are off-chip. If OVLY = 0, the entire 8M words of memory are off chip.

5.3 Memory-to-Memory Moves

There are various reasons for performing memory-to-memory moves. These reasons include making copies of buffers to preserve the original, moving contents from ROM to RAM, and moving copies of code from their load location to their execution location. Example 5–4 implements memory-to-memory moves on the ’54x using single-instruction repeat loops.

Example 5–4. Memory-to-Memory Block Moves Using the RPT Instruction

```
;
      .mmregs
      .text
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the MVDD instruction to move
; information in data memory to other data memory
; locations.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
MOVE_DD:
   STM   #4000h,AR2   ;Load pointer to source in
                      ;data memory.
   STM   #100h,AR3    ;Load pointer to
                      ;destination in data memory.
   RPT   #(1024–1)    ;Move 1024 value.
   MVDD  *AR2+,*AR3+
   RET
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the MVDP instruction to move external
; data memory to internal program memory.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
MOVE_DP:
   STM   #0E000h,AR1  ;Load pointer to source in
                      ;data memory.
   RPT   #(8192–1)    ;Move 8K to program memory space.
   MVDP  *AR1+,#800h
   RET
```

Example 5–4. Memory-to-Memory Block Moves Using the RPT Instruction (Continued)

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the MVPD instruction to move external
; program memory to internal data memory.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
MOVE_PD:
      STM      #0100h,AR1   ;Load pointer to
                            ;destination in data memory.
      RPT      #(128–1)     ;Move 128 words from external
      MVPD     #3800h,*AR1+ ;program to internal data
                            ;memory.
      RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the READA instruction to move external
; program memory to internal data memory. This differs
; from the MVPD instruction in that the accumulator
; contains the address in program memory from which to
; transfer. This allows for a calculated, rather than
; pre-determined, location in program memory to be
; specified.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
READ_A:
      STM      #0100h,AR1   ;Load pointer to
                            ;destination in data memory.
      RPT      #(128–1)     ;Move 128 words from external
      READA    *AR1+        ;program to internal data
                            ;memory.
      RET
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the WRITEA instruction to move data
; memory to program memory. The calling routine must
; contain the destination program memory address in the
; accumulator.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
WRITE_A:
      STM      #380h,AR1    ;Load pointer to source in
                            ;data memory.
      RPT      #(128–1)     ;Move 128 words from data
      WRITA    *AR1+        ;memory to program memory.
      RET
```

5.4 Power Management

The ’54x family of DSPs exhibits very low power dissipation and flexible power management. This is important in developing applications for portable systems, particularly wireless systems. Three aspects of power management are discussed here: on- versus off-chip memory, the use of $\overline{\text{HOLD}}$, and the use of IDLE modes.

To fetch and execute instructions from on-chip memory requires less power than accessing them from off-chip memory. The difference between these two accesses becomes noteworthy if a large piece of code resides off chip and is used more frequently than the on-chip code. The code can be partitioned so that the code that consumes the most power and is used most frequently is placed on-chip. (Masked ROM devices are another alternative for very high-performance applications.)

If the program is executed from internal memory and no external access occurs, switching of address outputs can be disabled with the AVIS bit in the PMST register. This feature saves a significant amount of power. However, once the AVIS bit is set, the address bus is still driven in its previous state. The external bus function in the bank-switching control register (BSCR) contributes to the state of the address, control, and data lines. If it is disabled, the address and data buses, along with the control lines, become inactive after the current bus cycle.

The $\overline{\text{HOLD}}$ signal and the HM bit initiate a power-down mode by either shutting off CPU execution or continuing internal CPU execution if external access is not necessary. This makes external memory available for other processors. The timers and serial ports are not used, and the device can be interrupted and serviced.

Using the IDLE1, IDLE2, and IDLE3 modes dissipates less power than normal operation. The system clock is not halted in IDLE1, but CPU activities are stopped. Peripherals and timers can bring the device out of power-down mode. The system can use the timer interrupt as a wake-up if the device needs to be in power-down mode periodically. The IDLE2 instruction halts both CPU and peripherals. Unlike the IDLE1 mode, an external interrupt wakes up the processor in IDLE2. The IDLE2 mode saves a significant amount of power, compared to IDLE1. The IDLE3 mode shuts off the internal clock, also saving power.

Chapter 6

Arithmetic and Logical Operations

This chapter shows how the ’54x supports typical arithmetic and logical operations, including multiplication, addition, division, square roots, and extended-precision operations.

Topic **Page**

6.1 Division and Modulus Algorithm

The ’54x implements division operations by using repeated conditional subtraction. Example 6–1 uses four types of integer division and modulus:

Type I: 32-bit by 16-bit unsigned integer division and modulus

Type II: 32-bit by 16-bit signed integer division and modulus

Type III: 16-bit by 16-bit unsigned integer division and modulus

Type IV: 16-bit by 16-bit signed integer division and modulus

SUBC performs binary division like long division. For 16-bit by 16-bit integer division, the dividend is stored in low part accumulator A. The program repeats the SUBC command 16 times to produce a 16-bit quotient in low part accumulator A and a 16-bit remainder in high part accumulator B. For each SUBC subtraction that results in a negative answer, you must left-shift the accumulator by 1 bit. This corresponds to putting a 0 in the quotient when the divisor does not go into the dividend. For each subtraction that produces a positive answer, you must left shift the difference in the ALU output by 1 bit, add 1, and store the result in accumulator A. This corresponds to putting a 1 in the quotient when the divisor goes into the dividend.

Similarly, 32-bit by 16-bit integer division is implemented using two stages of 16-bit by 16-bit integer division. The first stage takes the upper 16 bits of the 32-bit dividend and the 16-bit divisor as inputs. The resulting quotient becomes the higher 16 bits of the final quotient. The remainder is left shifted by 16 bits and adds the lower 16 bits of the original dividend. This sum and the 16-bit divisor become inputs to the second stage. The lower 16 bits of the resulting quotient is the final quotient and the resulting remainder is the final remainder.

Both the dividend and divisor must be positive when using SUBC. The division algorithm computes the quotient as follows:

1) The algorithm determines the sign of the quotient and stores this in accumulator B.

2) The program determines the quotient of the absolute value of the numerator and the denominator, using repeated SUBC commands.

3) The program takes the negative of the result of step 2, if appropriate, according to the value in accumulator B.

For unsigned division and modulus (types I and III), you must disable the sign extension mode (SXM = 0). For signed division and modulus (types II and IV), turn on sign extension mode (SXM = 1). The absolute value of the numerator must be greater than the absolute value of the denominator.

Example 6–1. Unsigned/Signed Integer Division Examples

```
;;=============================================================================== ;;
;; File Name: DIV_ASM.ASM
;;
;; Title: Divide & Modulus – Assembly Math Utilities.
;;
;; Original draft: Alex Tessaralo
;; Modified for ’54x: Simon Lau & Philip Jones
;;  Texas Instruments Inc.
;; ;;============================================================================
;;
;; Target:   C54X
;; ;;============================================================================
;;
;; Contents:     DivModUI32          ; 32–bit By 16–bit Unsigned Integer Divide
;;                                   ; And Modulus.
;;               DivModUI16          ; 16–bit By 16–bit Unsigned Integer Divide
;;                                   ; And Modulus.
;;               DivModI32           ; 32–bit By 16–bit Signed Integer Divide
;;                                   ; And Modulus.
;;               DivModI16           ; 16–bit By 16–bit Signed Integer Divide
;;                                   ; And Modulus.
;; ;;============================================================================
;;
;; History:  mm/dd/yy | Who      | Description Of Changes.
;;  ---------+----------+-------------------------------------
;;  08/01/96 | Simon L. | Original draft.
;;
;;=============================================================================
;;=============================================================================== ;;
;; Module Name: DivModUI32
;; ;;============================================================================
;;
;; Description: 32 Bit By 16 Bit Unsigned Integer Divide And Modulus
;; ;;-----------------------------------------------------------------------------;;
;; Usage ASM:
;;  .bss      d_NumH,1               ; 00000000h to FFFFFFFFh
;;  .bss      d_NumL,1
;;  .bss      d_Den,1                ; 0000h to FFFFh
;;  .bss      d_QuotH,1              ; 00000000h to FFFFFFFFh
;;  .bss      d_QuotL,1
;;  .bss      d_Rem,1                ; 0000h to FFFFh
;;
;;  CALL      DivModUI32
;; ;;-----------------------------------------------------------------------------;;
;; Input:    d_NumH
;;           d_NumL
;;           d_Den
```

Example 6–1. Unsigned/Signed Integer Division Examples (Continued)

```
;;
;; Modifies:  SXM
;;  accumulator A
;;
;; Output:    d_QuotH
;;            d_QuotL
;;            d_Rem
;;
;;-------------------------------------------------------------------------------;;
;; Algorithm: Quot          = Num/Den
;;      Rem                 = Num%Den
;;;;   NumH                 = n3|n2          QuotH  = q3|q2
;;      NumL                 = n1|n0          QuotL  = q1|q0
;;      Den                  = d1|d0          Rem    = r1|r0
;;
;;      Phase1: t1|t0|q3|q2 = A     (after repeating SUBC 16 times)
;;
;;      d1|d0 ) 00|00|n3|n2 = A     (before)
;;
;;
;;      Phase2: r1|r0|q1|q0     = A     (after repeating SUBC 16 times)
;;
;;      d1|d0 )   t1|t0|n1|n0   = A     (before)
;;
;;      NOTES:    Sign extension mode must be turned off.
;; ;;=============================================================================
    .def          DivModUI32
    .ref          d_NumH
    .ref          d_NumL
    .ref          d_Den
    .ref          d_QuotH
    .ref          d_QuotL
    .ref          d_Rem
    .textDivModUI32:
    RSBX          SXM                ; sign extention mode off
    LD            d_NumH,A
    RPT#(16-1)
    SUBC          d_Den,A
    STL A,        d_QuotH
    XOR           d_QuotH,A          ; clear AL
    OR            d_NumL,A           ; AL = NumL
    RPT#(16-1)
    SUBC          d_Den,A
    STL A,        d_QuotL
    STH A,d_Rem
    RET
;;=============================================================================== ;;
;; Module Name: DivModUI16
;; ;;=============================================================================
;;
;; Description: 16 Bit By 16 Bit Unsigned Integer Divide And Modulus
;; ;;-------------------------------------------------------------------------------;;
;; Usage ASM:
```

Example 6–1. Unsigned/Signed Integer Division Examples (Continued)

```
;;     .bss      d_Num,1             ; 0000h to FFFFh
;;     .bss      d_Den,1             ; 0000h to FFFFh
;;     .bss      d_Quot,1            ; 0000h to FFFFh
;;     .bss      d_Rem,1             ; 0000h to FFFFh
;;
;;     CALL      DivModUI16
;; ;;-------------------------------------------------------------------------;;
;; Input:        d_Num
;;               d_Den
;;
;; Modifies:     SXM
;;               accumulator A
;;
;; Output:       d_Quot
;;               d_Rem
;; ;;-------------------------------------------------------------------------;;
;; Algorithm: Quot   = Num/Den
;;     Rem = Num%Den
;;
;;     Num = n1|n0             Quot    = q1|q0
;;     Den = d1|d0             Rem     = r1|r0
;;
;;        r1|r0|q1|q0 = A     (after repeating SUBC 16 times)
;;
;;     d1|d0 ) 00|00|n1|n0 = A     (before)
;;
;;     NOTES:  Sign extension mode must be turned off.
;; ;;=========================================================================
       .def      DivModUI16
       .ref      d_Num
       .ref      d_Den
       .ref      d_Quot
       .ref      d_Rem
       .text
DivModUI16:
       RSBX      SXM                         ; sign extention mode off
       LD        @d_Num,A
       RPT       #(16-1)
       SUBC      @d_Den,A
       STL       A,@d_Quot
       STH       A,@d_Rem
       RET
;;=========================================================================== ;;
;; Module Name: DivModI32
;; ;;=========================================================================
;;
;; Description: 32 Bit By 16 Bit Signed Integer Divide And Modulus.
;; ;;-------------------------------------------------------------------------;;
;; Usage ASM:
;;     .bss      d_NumH,1                    ; 80000001h to 7FFFFFFFh
;;     .bss      d_NumL,1
;;     .bss      d_Den,1                     ;     8000h to 7FFFh
;;     .bss      d_QuotH,1                   ; 80000001h to 7FFFFFFFh
```

Example 6–1. Unsigned/Signed Integer Division Examples (Continued)

```
;;     .bss      d_QuotL,1
;;     .bss      d_Rem,1                      ;     8000h to 7FFFh
;;
;;     CALL   DivModI32
; ;;-------------------------------------------------------------------------;;
;; Input:        d_NumH
;;               d_NumL
;;               d_Den
;;
;; Modifies:     SXM
;;               T
;;               accumulator A
;;               accumulator B
;;
;; Output:       d_QuotH
;;               d_QuotL
;;               d_Rem
;; ;;-------------------------------------------------------------------------;;
;; Algorithm: Quot   = Num/Den
;;     Rem           = Num%Den
;;;;       Signed division is similar to unsigned division except that
;;     the sign of Num and Den must be taken into account.
;;     First the sign is determined by multiplying Num by Den.
;;     Then division is performed on the absolute values.
;;
;;     NumH   = n3|n2          QuotH  = q3|q2
;;     NumL   = n1|n0          QuotL  = q1|q0
;;     Den    = d1|d0          Rem    = r1|r0
;;
;;     Phase1: t1|t0|q3|q2 = A       (after repeating SUBC 16 times)
;;
;;     d1|d0 ) 00|00|n3|n2 = A       (before)
;;
;;
;;     Phase2: r1|r0|q1|q0 = A       (after repeating SUBC 16 times)
;;
;;     d1|d0 ) t1|t0|n1|n0 = A       (before)
;;
;;     NOTES:  Sign extension must be turned on.
;; ;;=========================================================================
       .def      DivModI32
       .ref      d_NumH
       .ref      d_NumL
       .ref      d_Den
       .ref      d_QuotH
       .ref      d_QuotL
       .ref      d_Rem
       .text
DivModI32:
       SSBX      SXM                    ; sign extention mode on
       LD        d_Den,16,A
       MPYA      d_NumH                 ; B has sign of quotient
       ABS       A
```

Example 6–1. Unsigned/Signed Integer Division Examples (Continued)

```
       STH A      ,d_Rem                    ; d_Rem = abs(Den) temporarily
       LD         d_NumH,16,A
       ADDS       d_NumL,A
       ABS        A
       STH        A,d_QuotH                 ; d_QuotH = abs(NumH) temporarily
       STL        A,d_QuotL                 ; d_QuotL = abs(NumL) temporarily
       LD         d_QuotH,A
       RPT        #(16–1)
       SUBC       d_Rem,A
       STL        A,d_QuotH                 ; AH = abs(QuotH)
       XOR        d_QuotH,A                 ; clear AL
       OR         d_QuotL,A                 ; AL = abs(NumL)
       RPT        #(16–1)
       SUBC       d_Rem,A
       STL        A,d_QuotL                 ; AL = abs(QuotL)
       STH        A,d_Rem                   ; AH = Rem
       BCD        DivModI32Skip,BGEQ        ; if B neg, then Quot =
                                            ; –abs(Quot)
       LD         d_QuotH,16,A
       ADDS       d_QuotL,A
       NEG        A
       STH        A,d_QuotH
       STL        A,d_QuotL
DivModI32Skip:
       RET
;;=========================================================================== ;;
;; Module Name: DivModI16
;; ;;=========================================================================
;;
;; Description: 16 Bit By 16 Bit Signed Integer Divide And Modulus.
;; ;;-------------------------------------------------------------------------;;
;; Usage ASM:
;;        .bss      d_Num,1             ; 8000h to 7FFFh (Q0.15 format)
;;        .bss      d_Den,1             ; 8000h to 7FFFh (Q0.15 format)
;;        .bss      d_Quot,1            ; 8000h to 7FFFh (Q0.15 format)
;;        .bss      d_Rem,1             ; 8000h to 7FFFh (Q0.15 format)
;;
;;        CALL      DivModI16
;; ;;-------------------------------------------------------------------------;;
;; Input:           d_Num
;;                  d_Den
;;
;; Modifies:        AR2
;;                  T
;;                  accumulator A
;;                  accumulator B
;;                  SXM
;;
;; Output:          d_Quot
;;                  d_Rem
;; ;;-------------------------------------------------------------------------;;
;; Algorithm:       Quot      = Num/Den
;;                  Rem       = Num%Den
```

Example 6–1. Unsigned/Signed Integer Division Examples (Continued)

```
;;
;;     Signed division is similar to unsigned division except that
;;     the sign of Num and Den must be taken into account.
;;     First the sign is determined by multiplying Num by Den.
;;     Then division is performed on the absolute values.
;;
;;                  Num       = n1|n0        Quot    = q1|q0
;;                  Den       = d1|d0        Rem     = r1|r0
;;
;;            r1|r0|q1|q0     = A       (after repeating SUBC 16 times)
;;             ___________
;;         d1|d0 )   00|00|n1|n0  = A     (before)
;;
;;     NOTES:  Sign extension mode must be turned on.
;;
;;=============================================================================
          .def      DivModI16
          .ref      d_Num
          .ref      d_Den
          .ref      d_Quot
          .ref      d_Rem
          .text
DivModI16:
          SSBX      SXM                     ; sign extention mode on
          STM       #d_Quot,AR2
          LD        d_Den,16,A
          MPYA      d_Num                   ; B has sign of quotient
          ABS       A
          STH       A,d_Rem                 ; d_Rem = abs(Den) temporarily
          LD        d_Num,A
          ABS       A                       ; AL = abs(Num)
          RPT       #(16–1)          SUBC      d_Rem,A
          STL       A,d_Quot                ; AL = abs(Quot)
          STH       A,d_Rem                 ; AH = Rem
          LD        #0,A
          SUB       d_Quot,16,A             ; AH = –abs(Quot)
          SACCD     A,*AR2,BLT              ; If B neg, Quot = –abs(Quot)
          RET
;;============================================================================= ;;
;;End Of File.
;;=============================================================================
```

6.2 Sines and Cosines

Sine-wave generators are used in signal processing systems, such as communications, instrumentation, and control. In general, there are two methods to generate sine and cosine waves. The first is the table look-up method, which is used for applications not requiring extreme accuracy. This method uses large tables for precision and accuracy and requires more memory. The second method is the Taylor series expansion, which is more efficient. This method determines the sine and cosine of an angle more accurately and uses less memory than table look-up, and it is discussed here.

The first four terms of the expansion compute the angle. The Taylor series expansions for the sine and cosine of an angle are:

$$\begin{aligned}
sin(\theta) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \\
&= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}\left(1 - \frac{x^2}{8.9}\right) \\
&= x - \frac{x^2}{3!} + \frac{x^5}{5!}\left(1 - \frac{x^2}{6.7}\left(1 - \frac{x^2}{8.9}\right)\right) \\
&= x - \frac{x^2}{3!}\left(1 - \frac{x^2}{4.5}\left(1 - \frac{x^2}{6.7}\left(1 - \frac{x^2}{8.9}\right)\right)\right) \\
&= x\left(1 - \frac{x^2}{2.3}\left(1 - \frac{x^2}{4.5}\left(1 - \frac{x^2}{6.7}\left(1 - \frac{x^2}{8.9}\right)\right)\right)\right)
\end{aligned}$$

$$\begin{aligned}
cos(\theta) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} \\
&= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!}\left(1 - \frac{x^2}{7.8}\right) \\
&= 1 - \frac{x^2}{2!} + \frac{x^4}{4!}\left(1 - \frac{x^2}{5.6}\left(1 - \frac{x^2}{7.8}\right)\right) \\
&= 1 - \frac{x^2}{2}\left(1 - \frac{x^2}{3.4}\left(1 - \frac{x^2}{5.6}\left(1 - \frac{x^2}{7.8}\right)\right)\right)
\end{aligned}$$

The following recursive formulas generate the sine and cosine waves:

$$sin\ n\theta = 2\ cos(\theta)sin\{(n-1)\theta\} - sin\{(n-2)\theta\}$$

$$cos\ n\theta = 2\ cos(\theta)cos\{(n-1)\theta\} - cos\{(n-2)\theta\}$$

These equations use two steps to generate a sine or cosine wave. The first evaluates cos(θ) and the second generates the signal itself, using one multiply and one subtract for a repeat counter, n.

Example 6–2 and Example 6–3 assumes that the delayed cos((n–1)) and cos((n–2)) are precalculated and are stored in memory. The Taylor series expansion to evaluate the delayed cos((n–1)), cos((n–2))/sin((n–1)), and sin((n–2)) values for a given θ can also be used.

Example 6–2. Generation of a Sine Wave

```
; Functional Description
;  This function evaluates the sine of an angle using the Taylor series
;  expansion.
;  sin(theta) = x(1–x^2/2*3(1–x^2/4*5(1–x^2/6*7(1–x^2/8*9))))
;
          .mmregs
          .def          d_x,d_squr_x,d_coff,d_sinx,C_1
d_coff    .sect         "coeff"
          .word         01c7h
          .word         030bh
          .word         0666h
          .word         1556h
d_x       .usect        "sin_vars",1
d_squr_x  .usect         "sin_vars",1
d_temp    .usect        "sin_vars",1
d_sinx    .usect        "sin_vars",1
C_1       .usect        "sin_vars",1
          .text
sin_start:
          STM       #d_coff,AR3         ; c1=1/72,c2=1/42,c3=1/20,
                                        ; c4=1/6
          STM       #d_x,AR2            ; input value
          STM       #C_1,AR4            ; A1, A2, A3, A4
          sin_angle:
          LD        #d_x,DP
          ST        #6487h,d_x          ; pi/4
          ST        #7fffh,C_1
          SQUR      *AR2+,A             ; let x^2 = P
          ST        A,*AR2              ; AR2  – > x^2
          || LD     *AR4,B              ;
          MASR      *AR2+,*AR3+,B,A     ; (1–x^2)/72
          MPYA      A                   ; 1–x^2(1–x^2)/72
                                        ; T = x^2
          STH       A,*AR2
          MASR      *AR2–,*AR3+,B,A     ; A = 1–x^2/42(1–x^2/72)
                                        ; T =x^2(1–x^2/72)
          MPYA      *AR2+               ; B = A(32–16)*x^2
          ST        B,*AR2              ;
          || LD     *AR4,B              ; B = C_1
          MASR      *AR2–,*AR3+,B,A     ; A = 1–x^2/20(1–x^2/42(1–x^2/72)
          MPYA      *AR2+               ; B = A(32–16)*x^2
          ST        B,*AR2
          || LD     *AR4,B
          MASR      *AR2–,*AR3+,B,A     ; AR2 – > d_squr_x
          MPYA      d_x
          STH       B, d_sinx           ; sin(theta)
          RET
          .end
```

Example 6–2. Generation of a Sine Wave (Continued)

```
; Functional Description
; This function generates the sine of angle. Using the recursive given above, the
; cosine of the angle is found and the recursive formula is used to generate the
; sine wave. The sin(n–1) and sin(n–2) can be calculated using the Taylor
; series expansion or can be pre–calculated.
                .mmregs
                .ref      cos_prog,cos_start
d_sin_delay1    .usect    "cos_vars",1
d_sin_delay2    .usect    "cos_vars",1
K_sin_delay_1   .set      0A57Eh     ; sin(–pi/4)
K_sin_delay_2   .set      8000h      ; sin(–2*pi/4);
K_2             .set      2h         ; cicular buffer size
K_256           .set      256        ; counter
K_THETA         .set      6487h      ; pi/4
                .text
start:
       LD     #d_sin_delay1,DP
       CALL   cos_start
       STM    #d_sin_delay1,AR3      ; intialize the buffer
       RPTZ   A,#3h
       STL    A,*AR3+
       STM    #1,AR0
       STM    #K_2,BK
       STM    #K_256–1,BRC
       STM    #d_sin_delay1,AR3
       ST     #K_sin_delay_1,*AR3+%  ; load calculated initial values of sin((n–1) )
       ST     #K_sin_delay_2,*AR3+%  ; load calculated initial values of sin((n–2) )
                                     ; this generates the sine_wave
sin_generate:
       RPTB   end_of_sine
       MPY    *AR2,*AR3+0%,A         ; cos(theta)*sin{(n–1)theta}
       SUB    *AR3,15,A              ; 1/2*sin{(n–2)theta)
       SFTA   A,1,A                  ; sin(n*theta)
       STH    A,*AR3                 ; store
end_of_sine
       NOP
       NOP
       B      sin_generate
       .end
```

Example 6–3. Generation of a Cosine Wave

```
; Functional Description
; this computes the cosine of an angle using the Taylor Series Expansion
       .mmregs
       .def   d_x,d_squr_x,d_coff,d_cosx,C_7FFF
       .def   cos_prog,cos_start
       STH    A,*AR3                          ; store
       .word  024ah                           ; 1/7.8
       .word  0444h                           ; 1/5.6
       .word  0aa9h                           ; 1/3.4
d_x    .usect "cos_vars",1
d_squr_x      .usect    "cos_vars",1
d_cosx        .usect    "cos_vars",1
C_7FFF        .usect    "cos_vars",1
K_THETA       .set      6487h                ; pi/4
K_7FFF        .set      7FFFh
              .text
cos_start:
              STM       #d_coff,AR3          ;c1=1/56,c2=1/30,c3=1/12
              STM       #d_x,AR2             ; input theta
              STM       #C_7FFF,AR4          ; A1, A2, A3, A4
cos_prog:
              LD        #d_x,DP
              ST        #K_THETA,d_x         ; input theta
              ST        #K_7FFF,C_7FFF
              SQUR      *AR2+,A              ; let x^2 = P
              ST        A,*AR2               ; AR2  – > x^2
              || LD     *AR4,B               ;
              MASR      *AR2+,*AR3+,B,A      ; (1–x^2)/72
              MPYA      A                    ; 1–x^2(1–x^2)/72
                                             ; T = x^2
              STH       A,*AR2
              MASR      *AR2–,*AR3+,B,A      ; A = 1–x^2/42(1–x^2/72)
                                             ; T =x^2(1–x^2/72)
              MPYA      *AR2+                ; B = A(32–16)*x^2
              ST        B,*AR2               ;
              || LD     *AR4,B               ; B = C_1
              MASR      *AR2–,*AR3+,B,A      ; A = 1–x^2/20(1–x^2/42(1–x^2/72))
              SFTA      A,–1,A               ; –1/2
              NEG       A
              MPYA      *AR2+                ; B = A(32–16)*x^2
              RETD
              ADD       *AR4,16,B
              STH       B,*AR2               ; cos(theta)
              .end
              .mmregs
              .ref      cos_prog,cos_start
d_cos_delay1  .usect    "cos_vars",1
d_cos_delay2  .usect    "cos_vars",1
d_theta       .usect    "cos_vars",1
```

Example 6–3. Generation of a Cosine Wave (Continued)

```
K_cos_delay_1    .set      06ed9h                ; cos(–pi/6)
K_cos_delay_2    .set      4000h                 ; cos(–2*pi/6);
K_2              .set      2h                    ; cicular buffer size
K_256            .set      256                   ; counter
K_theta          .set      4303h                 ; sin(pi/2–pi/6)= cos(pi/6)
                                                 ; cos(pi/2–pi/x)
                                                 ; .052= 4303h
                 .text
start:
       LD        #d_cos_delay1,DP
       CALL      cos_start
       CALL      cos_prog                        ; calculate cos(theta)
       STM       #d_cos_delay1,AR3
       RPTZ      A,#3h
       STL       A,*AR3+
       STM       #d_cos_delay1,AR3
       ST        #K_cos_delay_1,*AR3+
       ST        #K_cos_delay_2,*AR3
       STM       #d_cos_delay1,AR3               ; output vaues
       ST        #K_theta,d_theta
       STM       #1,AR0
       STM       #K_2,BK
       STM       #K_256–1,BRC
cos_generate:
       RPTB      end_of_cose
       MPY       *AR2,*AR3+0%,A                  ; cos(theta)*cos{(n–1)theta}
       SUB       *AR3,15,A                        ; 1/2*cos{(n–2)theta)
       SFTA      A,1,A                           ; cos(n*theta)
       STH       A,*AR3                          ; store
       PORTW     *AR3,56h                         ; write to a port
end_of_cose
       NOP
       NOP
       B         cos_generate                    ; next sample
       .end
```

6.3 Square Roots

Example 6–4 uses a 6-term Taylor series expansion to approximate the square root of a single-precision 32-bit number. A normalized, 32-bit, left-justified number is passed to the square root function. The output is stored in the upper half of the accumulator, and the EXP and NORM instructions normalize the input value. The EXP instruction computes an exponent value in a single cycle and stores the result in T, allowing the NORM instruction to normalize the number in a single cycle. If the exponent is an odd power, the mantissa is (multiplied by 1 divided by the square root of 2) to compensate after finding the square root of the 32-bit number. The exponent value is negated to denormalize the number.

$$y^{0.5} = (1 + x)^{0.5}$$

where :

$$x = y–1$$

$$= 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{5x^4}{128} + \frac{7x^5}{256}$$

$$= 1 + \frac{x}{2} - 0.5\left(\frac{x}{2}\right)^2 + 0.5\left(\frac{x}{2}\right)^3 - 0.625\left(\frac{x}{2}\right)^4 + 0.875\left(\frac{x}{2}\right)^5$$

where :

$$0.5 \leqq x < 1$$

Example 6–4. Square Root Computation

```
*************************************************************************
* Six term Taylor Series is used here to compute the square root of a number
* y^0.5 = (1+x)^0.5 where x = y–1
*   = 1+(x/2)–0.5*((x/2)^2+0.5*((x/2)^3–0.625*((x/2)^4+0.875*((x/2)^5)
* 0.5 <= x < 1
*************************************************************************
       .mmregs
       .sect     ”squr_var”
d_part_prod     .word  0
d_part_shift    .word  0
C_8000          .word  0
C_sqrt_one_half .word  0
d_625           .word  0
d_875           .word  0
tmp_rg1         .word  0
K_input         .set   800h              ; input # = 0.0625
K_8000          .set   8000h             ; –1 or round off bit
K_4000          .set   4000h             ; 0.5 coeff
```

Example 6–4. Square Root Computation (Continued)

```
K_SQRT_HALF   .set   5a82h              ; 1/sqrt2
K_625         .set   –20480             ; coeff 0.625
K_875         .set   28672              ; coeff 0.875
              .text
sqroot:
              LD     #d_part_prod,DP
              ST     #K_8000,C_8000
              ST     #K_input,d_part_prod
              ST     #K_SQRT_HALF,C_sqrt_one_half
              ST     #K_875,d_875
              ST     #K_625,d_625
              LD     d_part_prod,16,A    ; load the #
              EXP    A
              nop                        ; dead cycle
              NORM   A
              ADDS   C_8000,A            ; round off bit
              STH    A, d_part_prod      ; normalized input
              LDM    T,B
              SFTA   B,–1,B              ; check for odd or even power
              BCD    res_even,NC
              NE     B                   ; negate the power
              STL    B,d_part_shift      ; this shift is used to denormalize the #
              LD     d_part_prod,16,B    ; load the normalized input #
              CALLD  sq_root             ; square root program
              ABS    B
              NOP                        ; cycle for delayed slot
              LD     B,A                 ;
              BD     res_common
              SUB    B,B                 ; zero B
              MACAR  C_sqrt_one_half,B   ; square root of 1/2
                                         ; odd power
res_even
              LD     d_part_prod,16,B
              CALLD  sq_root
              ABS B
              NOP                        ; cycle for the delayed slot
res_common
              LD     d_part_shift,T      ; right shift value
              RETD
              STH    B,d_part_prod
              LD     d_part_prod,TS,A    ; denormaliize the #
sq_root:
              SFTA   B,–1,B              ; x/2 = y–1/2
              SUB    #K_4000,16,B,B
              STH    B,tmp_rg1           ; tmp_rg1 = x/2
              SUB    #K_8000,16,B        ; B = 1+x/2
              SQUR   tmp_rg1,A           ; A (x/2)^2, T = x/2
              NEG    A                   ; A = –A
              ADD    A,–1,B              ; B = 1+x/2–.5(x/2)^2
```

Example 6–4. Square Root Computation (Continued)

```
SQUR   A,A                ; A = (x/2)^4
MACA   d_625,B            ; 0.625*A+B
                          ; T =0.625
LD     tmp_rg1,T          ; T = x/2
MPYA   A                  ; (x/2)^4*x/2
MACA   d_875,B            ; 0.875*A+B
SQUR   tmp_rg1,A          ; x/2^2; T = x/2
MPYA   A                  ; A = x/2*x/2^2
RETD
ADD    A,–1,B
ADDS   C_8000,B           ; round off bit
.end
```

6.4 Extended-Precision Arithmetic

Numerical analysis, floating-point computations, and other operations may require arithmetic operations with more than 32 bits of precision. Since the ’54x devices are 16/32-bit fixed-point processors, software is required for arithmetic operations with extended precision. These arithmetic functions are performed in parts, similar to the way in which longhand arithmetic is done.

The ’54x has several features that help make extended-precision calculations more efficient. One of the features is the carry bit, which is affected by most arithmetic ALU instructions, as well as the rotate and shift operations. The carry bit can also be explicitly modified by loading ST0 and by instructions that set or reset status register bits. For proper operation, the overflow mode bit should be reset (OVM = 0) to prevent the accumulator from being loaded with a saturation value.

The two ’54x internal data buses, CB and DB, allow some instructions to handle 32-bit operands in a single cycle. The long-word load and double-precision add/subtract instructions use 32-bit operands and can efficiently implement multi-precision arithmetic operations.

The hardware multiplier can multiply signed/unsigned numbers, as well as multiply two signed numbers and two unsigned numbers. This makes 32-bit multiplication efficient.

6.4.1 Addition and Subtraction

The carry bit, C, is set in ST0 if a carry is generated when an accumulator value is added to:

- The other accumulator
- A data-memory operand
- An immediate operand

A carry can also be generated when two data-memory operands are added or when a data-memory operand is added to an immediate operand. If a carry is not generated, the carry bit is cleared.

The ADD instruction with a 16-bit shift is an exception because it only sets the carry bit. This allows the ALU to generate the appropriate carry when adding to the lower or upper half of the accumulator causes a carry.

Figure 6–1 shows several 32-bit additions and their effect on the carry bit.

Figure 6–1. 32-Bit Addition

```
C  MSB             LSB          C  MSB             LSB
X  F F F F F F F F F F ACC      X  F F F F F F F F F F ACC
  +                 1             +F F F F F F F F F F
1  0 0 0 0 0 0 0 0 0 0            1  F F F F F F F F F E

C  MSB             LSB          C  MSB             LSB
X  0 0 7 F F F F F F F ACC      X  0 0 7 F F F F F F F ACC
  +                 1             +F F F F F F F F F F
0  0 0 8 0 0 0 0 0 0 0            1  0 0 7 F F F F F F E

C  MSB             LSB          C  MSB             LSB
X  F F 8 0 0 0 0 0 0 0 ACC      X  F F 8 0 0 0 0 0 0 0 ACC
  +                 1             +F F F F F F F F F F
0  F F 8 0 0 0 0 0 0 1            1  F F 7 F F F F F F F

ADDC
C  MSB             LSB          C  MSB             LSB
1  0 0 0 0 0 0 0 0 0 0 ACC      1  F F F F F F F F F F ACC
  +                 0 (ADDC)     +                 0 (ADDC)
0  0 0 0 0 0 0 0 0 0 1            1  0 0 0 0 0 0 0 0 0 0

ADD Smem,16,src
C  MSB             LSB          C  MSB             LSB
1  F F 8 0 0 0 F F F F ACC      1  F F 8 0 0 0 F F F F ACC
  +0 0 0 0 0 1 0 0 0 0           +0 0 7 F F F 0 0 0 0
1  F F 8 0 0 1 F F F F            1  F F F F F F F F F F
```

Example 6–5 adds two 64-bit numbers to obtain a 64-bit result. The partial sum of the 64-bit addition is efficiently performed by the DLD and DADD instructions, which handle 32-bit operands in a single cycle. For the upper half of a partial sum, the ADDC (ADD with carry) instruction uses the carry bit generated in the lower 32-bit partial sum. Each partial sum is stored in two memory locations by the DST (long-word store) instruction.

Example 6–5. 64-Bit Addition

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 64-bit Addition
;
;       X3 X2 X1 X0
;     + Y3 Y2 Y1 Y0
;     -------------
;       W3 W2 W1 W0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ADD64: DLD   @X1,A      ;A = X1 X0
       DADD  @Y1,A      ;A = X1 X0 + Y1 Y0
       DST   A,@W1
       DLD   @X3,A      ;A = X3 X2
       ADDC  @Y2,A      ;A = X3 X2 + 00 Y2 + C
       ADD   @Y3,16,A   ;A = X3 X2 + Y3 Y2 + C
       DST   A,@W3
       RET
```

Similar to addition, the carry bit is reset if a borrow is generated when an accumulator value is subtracted from:

- The other accumulator
- A data-memory operand
- An immediate operand

A borrow can also be generated when two data-memory operands are subtracted or when an immediate operand is subtracted from a data-memory operand. If a borrow is not generated, the carry bit is set.

The SUB instruction with a 16-bit shift is an exception because it only resets the carry bit. This allows the ALU to generate the appropriate carry when subtracting from the lower or the upper half of the accumulator causes a borrow.

Figure 6–2 shows several 32-bit subtractions and their effect on the carry bit.

Figure 6–2. 32-Bit Subtraction

```
C  MSB             LSB        C  MSB             LSB
X  0 0 0 0 0 0 0 0 0 0 ACC    X  F F 0 0 0 0 0 0 0 0 ACC
  –                   1          –F F F F F F F F F F
0  F F F F F F F F F F        0  0 0 0 0 0 0 0 0 0 1

C  MSB             LSB        C  MSB             LSB
X  0 0 7 F F F F F F F ACC    X  0 0 7 F F F F F F F ACC
  –                   1          –F F F F F F F F F F
1  0 0 7 F F F F F F E        C  F F 8 0 0 0 0 0 0 0

C  MSB             LSB        C  MSB             LSB
X  F F 8 0 0 0 0 0 0 0 ACC    X  F F 8 0 0 0 0 0 0 0 ACC
  –                   1          –F F F F F F F F F F
1  F F 7 F F F F F F F        0  F F 8 0 0 0 0 0 0 1
```

SUBB

```
C  MSB             LSB        C  MSB             LSB
0  0 0 0 0 0 0 0 0 0 0 ACC    0  F F F F F F F F F F ACC
  –                   0 (SUBB)   –                   0(SUBB)
0  F F F F F F F F F F        1  F F F F F F F F F E
```

SUB Smem,16,src

```
C  MSB             LSB        C  MSB             LSB
1  F F 8 0 0 0 F F F F ACC    0  F F 8 0 0 0 F F F F ACC
  –0 0 0 0 0 1 0 0 0 0          –F F F F F F 0 0 0 0
0  0 0 7 F F F F F F F        0  F F 8 0 0 1 F F F F
```

Example 6–6 subtracts two 64-bit numbers on the ’54x. The partial remainder of the 64-bit subtraction is efficiently performed by the DLD (long word load) and the DSUB (double precision subtract) instructions, which handle 32-bit operands in a single cycle. For the upper half of a partial remainder, the SUBB (SUB with borrow) instruction uses the borrow bit generated in the lower 32-bit partial remainder. Each partial remainder is stored in two consecutive memory locations by a DST.

Example 6–6. 64-Bit Subtraction

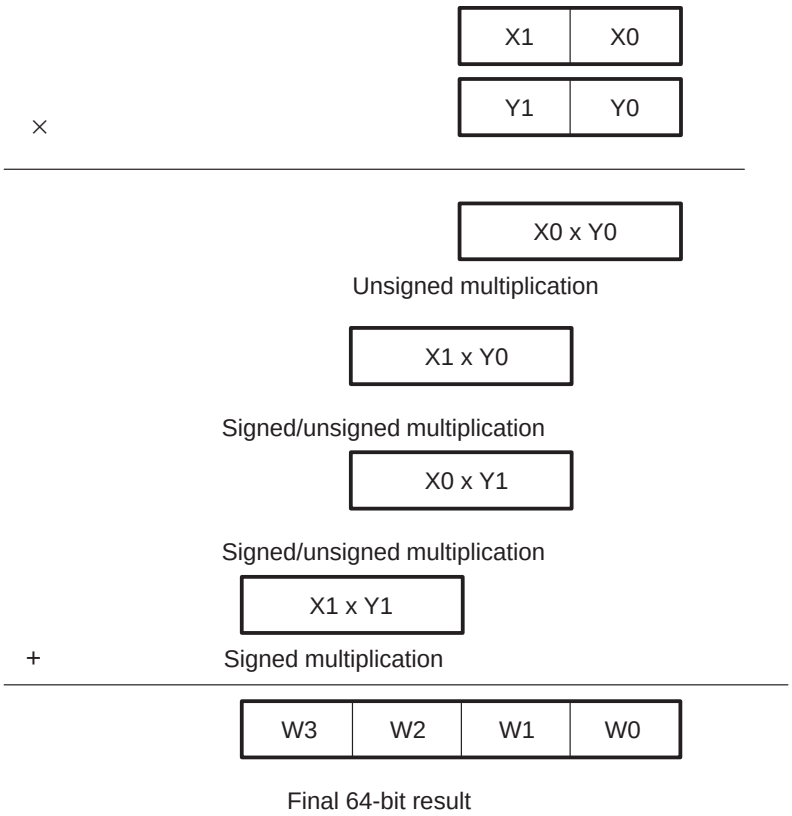
```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; 64 bit Subtraction
;
;        X3 X2 X1 X0
;      – Y3 Y2 Y1 Y0
;      –––––––––––––
;        W3 W2 W1 W0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
      DLD    @X3,A     ;A = X3 X2
      SUBB   @Y2,A     ;A = X3 X2 – 00 Y2 – (inv C)
      DST    A,@W1
      SUB    @Y3,16,A  ;A = X3 X2 – Y3 Y2 – (inv C)
      DST    A,@W3
      RET
```

6.4.2 Multiplication

The MPYU (unsigned multiply) and MACSU (signed/unsigned multiply and accumulate) instructions can also handle extended-precision calculations.

Figure 6–3 shows how two 32-bit numbers obtain a 64-bit product. The MPYU instruction multiplies two unsigned 16-bit numbers and places the 32-bit result in one of the accumulators in a single cycle. The MACSU instruction multiplies a signed 16-bit number by an unsigned 16-bit number and accumulates the result in a single cycle. Efficiency is gained by generating partial products of the 16-bit portions of a 32-bit (or larger) value instead of having to split the value into 15-bit (or smaller) parts.

Figure 6–3. 32-Bit Multiplication



The program in Example 6–7 shows that a multiply of two 32-bit integer numbers requires one multiply, three multiply/accumulates, and two shifts. The product is a 64-bit integer number. Note in particular, the use of MACSU, MPYU and LD instructions. The LD instruction can perform a right-shift in the accumulator by 16 bits in a single cycle.

Example 6–8 performs fractional multiplication. The operands are in Q31 format, while the product is in Q30 format.

Example 6–7. 32-Bit Integer Multiplication

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine multiplies two 32-bit signed integers
; resulting; in a 64-bit product. The operands are fetched
; from data memory and the result is written back to data
; memory.
; Data Storage:
;    X1,X0           32-bit operand
;    Y1,Y0           32-bit operand
;    W3,W2,W1,W0     64-bit product
; Entry Conditions:
;    SXM = 1, OVM = 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   STM   #X0,AR2          ;AR2 = X0 addr
   STM   #Y0,AR3          ;AR3 = Y0 addr
   LD    *AR2,T           ;T = X0
   MPYU  *AR3+,A          ;A = X0*Y0
   STL   A,@W0            ;save W0
   LD    A,-16,A          ;A = A >> 16
   MACSU *AR2+,*AR3-,A    ;A = X0*Y0>>16 + X0*Y1
   MACSU *AR3+,*AR2,A     ;A = X0*Y0>>16 + X0*Y1 + X1*Y0
   STL   A,@W1            ;save W1
   LD    A,-16,A          ;A = A >> 16
   MAC   *AR2,*AR3,A      ;A = (X0*Y1 + X1*Y0)>>16 + X1*Y1
   STL   A,@W2            ;save W2
   STH   A,@W3            ;save W3
```

Example 6–8. 32-Bit Fractional Multiplication

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine multiplies two Q31 signed integers
; resulting in a Q30 product. The operands are fetched
; from data memory and the result is written back to data
; memory.
; Data Storage:
;    X1,X0           Q31 operand
;    Y1,Y0           Q31 operand
;    W1,W0           Q30 product
; Entry Conditions:
;    SXM = 1, OVM = 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   STM   #X0,AR2          ;AR2 = X0 addr
   STM   #Y1,AR3          ;AR3 = Y1 addr
   LD    #0,A             ;clear A
   MACSU *AR2+,*AR3-,A    ;A = X0*Y1
   MACSU *AR3+,*AR2,A     ;A = X0*Y1 + X1*Y0
   LD    A,-16,A          ;A = A >> 16
   MAC   *AR2,*AR3,A      ;A = A + X1*Y1
   STL   A,@W0            ;save lower product
   STH   A,@W1            ;save upper product
```

6.5 Floating-Point Arithmetic

In fixed-point arithmetic, the binary point that separates the integer from the fractional part of the number is fixed at a certain location. For example, if a 32-bit number places the binary point after the most significant bit (which is also the sign bit), only fractional numbers (numbers with absolute values less than 1), can be represented. The fixed-point system, although simple to implement in hardware, imposes limitations in the dynamic range of the represented number. You can avoid this difficulty by using floating-point numbers.

A floating-point number consists of a mantissa, m, multiplied by a base, b, raised to an exponent, e, as follows:

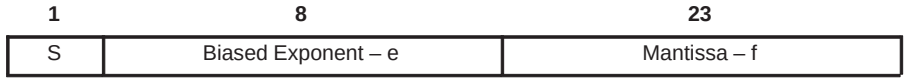
$m * b^e$

To implement floating-point arithmetic on the ’54x, operands must be converted to fixed-point numbers and then back to floating-point numbers. Fixed-point values are converted to floating-point values by normalizing the input data.

Floating-point numbers are generally represented by mantissa and exponent values. To multiply two numbers, add their mantissas, multiply the exponents, and normalize the resulting mantissa. For floating-point addition, shift the mantissa so that the exponents of the two operands match. Left-shift the lower-power operand by the difference between the two exponents. Add the exponents and normalize the result.

Figure 6–4 illustrates the IEEE standard format to represent floating-point numbers. This format uses sign-magnitude notation for the mantissa, and the exponent is biased by 127. In a 32-bit word representing a floating-point number, the first bit is the sign bit, represented by s. The next eight bits correspond to the exponent, which is expressed in an offset-by-127 format (the actual exponent is e–127). The following 23 bits represent the absolute value of the mantissa, with the most significant 1 implied. The binary point is placed after this most significant 1. The mantissa, then, has 24 bits.

Figure 6–4. IEEE Floating-Point Format



The values of the numbers represented in the IEEE floating-point format are as follows:

$(-1)^S * 2^{e-127} * (01.f)$	If 0 < e < 255
Special Cases:	
$(-1)^S * 0.0$	If e = 0, and f = 0 (zero)
$(-1)^S * 2–126 * (0.f)$	If e = 0 and f <> 0 (denormalized)
$(-1)^S *$ infinity	If e = 255 and f = 0 (infinity)
NaN (not a number)	If e = 255 and f <> 0

Example 6–9 through Example 6–11 illustrate how the ’54x performs floating-point addition, multiplication, and division.

Example 6–9. Add Two Floating-Point Numbers

```
*;*************************************************************************
*; FLOAT_ADD – add two floating point numbers
*; Copyright (c) 1993–1994 Texas Instruments Incorporated
*; NOTE: The ordering of the locals are placed to take advantage of long word
*; loads and stores which require the hi and low words to be at certain addresses.
*; Any future modifications which involve the stack must take this quirk into
*; account
*;*************************************************************************
*************************************************************************
;Operand 1 (OP1) and Operand (OP2) are each packed into sign, exponent, and the
;words of mantissa. If either exponent is zero special case processing is initiated.
;In the general case, the exponents are compared and the mantissa of the lower
;exponent is renormalized according to the number with the larger exponent. The
;mantissas are also converted to a two’s complement format to perform the actual
;addition. The result of the addition is then renormalized with the corresponding
;adjustment in the exponent. The resulting mantissa is converted back to its
;original sign–magnitude format and the result is repacked into the floating point
;representation.
*************************************************************************
*;*************************************************************************
*;       resource utilization:  B accumulator, T–register
*;       status bits affected: TC, C, SXM, OVM,
*;       entry requirements : CPL bit set
*;*************************************************************************
```

Example 6–9. Add Two Floating-Point Numbers (Continued)

```
; Floating Point Format – Single Precision
*-------------------------------------------------------------------------------
*| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*|  S | E7 | E6 | E5 | E4 | E3 | E2 | E1 | E0 | M22| M21| M20| M19| M18| M17| M16|
*-------------------------------------------------------------------------------
*-------------------------------------------------------------------------------
*| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*| M15| M14| M13| M12| M11| M10| M9 | M8 | M7 | M6 | M5 | M4 | M3 | M2 | M1 | M0 |
*-------------------------------------------------------------------------------
*; Single precision floating point format is a 32 bit format consisting of a 1 bit
sign field, an 8 bit exponent field, and a 23 bit mantissa field. The fields are
defined as follows
*;  Sign <S>          : 0 = positive values; 1 = negative value
*;  Exponent <E7–E0>  : offset binary format
*;              00 = special cases (i.e. zero)
*;                   01 = exponent value + 127 = –126
*;                   FE = exponent value + 127 = +127
*;                   FF = special cases (not implemented)
*;  Mantissa <M22–M0> : fractional magnitude format with implied 1
*;                                1.M22M21...M1M0
*;            Range             : –1.9999998 e+127 to –1.0000000 e–126
*;                                +1.0000000 e–126 to +1.9999998 e+127
*;                                (where e represents 2 to the power of)
*;                                –3.4028236 e+38  to –1.1754944 e–38
*;                                +1.1754944 e–38  to +3.4028236 e+38
*;                                (where e represents 10 to the power of)
*;*******************************************************************************
res_hm       .usect "flt_add",1      ; result high mantissa
res_lm       .usect "flt_add",1      ; result low mantissa
res_exp      .usect "flt_add",1      ; result exponent
res_sign     .usect "flt_add",1      ; result sign
op2_hm       .usect "flt_add",1      ; OP2 high mantissa
op2_lm       .usect "flt_add",1      ; OP2 low mantissa
op2_se       .usect "flt_add",1      ; OP2 sign and exponent
op1_se       .usect "flt_add",1      ; OP1 sign and exponent
op1_hm       .usect "flt_add",1      ; OP1 high mantissa
op1_lm       .usect "flt_add",1      ; OP1 low mantissa
op1_msw      .usect "flt_add",1      ; OP1 packed high word
op1_lsw      .usect "flt_add",1      ; OP1 packed low word
op2_msw      .usect "flt_add",1      ; OP2 packed high word
op2_lsw      .usect "flt_add",1      ; OP2 packed low word
err_no       .usect "flt_add",1      ;
        .mmregs
*****************************************************************
* Floating point number 12.0 can be represented as 1100 = 1.100 x 23 => sign =0
*   biased exponent = 127+3 = 130
*             130 = 10000010
*         Mantissa 10000000000000000000
* Thus 12.0 can be represented as 01000001010000000000000000000000= 4140h
*******************************************************************************
*
```

Example 6–9. Add Two Floating-Point Numbers (Continued)

```
K_OP1_HIGH    .set   4140h        ; floating point number 12.0
K_OP1_LOW     .set   0000h
K_OP2_HIGH    .set   4140h        ; floating point number 12.0
K_OP2_LOW     .set   0000h
       .mmregs
       .text
start_flt:
       RSBX   C16
       LD     #res_hm,DP          ; initialize the page pointer
       LD     #K_OP2_HIGH,A       ; load floating #2 – 12
       STL    A,op2_msw
       LD     #K_OP2_LOW,A
       STL    A,op2_lsw
       LD     #K_OP1_HIGH,A       ; load floating #1 –  12
       STL    A,op1_msw
       LD     #K_OP1_LOW,A
       STL    A,op1_lsw
*
*;*****************************************************************************
*;       CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;    Test OP1 for special case treatment of zero.
*;    Split the MSW of OP1 in the accumulator.
*;    Save the exponent on the stack [xxxx xxxx EEEE EEEE].
*;    Add the implied one to the mantissa value.
*;    Store the mantissa as a signed value
*;*****************************************************************************
*
       DLD    op1_msw,A           ; load the OP1 high word
       SFTA   A,8                 ; shift right by 8
       SFTA   A,–8
       BC     op1_zero,AEQ        ; If op1 is 0, jump to special case
       LD     A,B                 ; Copy OP1 to acc B
       RSBX   SXM                 ; Reset for right shifts used for masking
       SFTL   A,1                 ; Remove sign bit
       STH    A,–8,op1_se         ; Store exponent to stack
       SFTL   A,8                 ; Remove exponent
       SFTL   A,–9
       ADD    #080h,16,A          ; Add implied 1 to mantissa
       XC     1,BLT               ; Negate OP1 mantissa for negative values
       NEG    A
       SSBX   SXM                 ; Make sure OP2 is sign–extended
       DST    A,op1_hm            ; Store mantissa
*
*;*****************************************************************************
*;       CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;    Test OP1 for special case treatment of zero.
*;    Split the MSW of OP1 in the accumulator.
*;    Save the exponent on the stack [xxxx xxxx EEEE EEEE].
*;    Add the implied one to the mantissa value.
*;    Store the mantissa as a signed value
*;*****************************************************************************
*
```

Example 6–9. Add Two Floating-Point Numbers (Continued)

```
       DLD    op2_msw,A           ; Load acc with op2
       BC     op2_zero,AEQ        ; If op2 is 0, jump to special case
       LD     A,B                 ; Copy OP2 to acc B
       SFTL   A,1                 ; Remove sign bit
       STH    A,–8,op2_se         ; Store exponent to stack
       RSBX   SXM                 ; Reset for right shifts used for masking
       SFTL   A,8                 ; Remove exponent
       SFTL   A,–9
       ADD    #080h,16,A          ; Add implied 1 to mantissa
       XC     1,BLT               ; Negate OP2 mantissa for negative values
       NEG    A
       SSBX   SXM                 ; Set sign extension mode
       DST    A,op2_hm            ; Store mantissa
**;*****************************************************************************
*;       EXPONENT COMPARISON
*;  Compare exponents of OP1 and OP2 by subtracting: exp OP2 – exp OP1
*;  Branch to one of three blocks of processing
*;     Case 1:  exp OP1 is less than exp OP2
*;     Case 2:  exp OP1 is equal to exp OP2
*;     Case 3:  exp OP1 is greater than exp OP2
*;*****************************************************************************
*
       LD     op1_se,A            ; Load OP1 exponent
       LD     op2_se,B            ; Load OP2 exponent
*
       SUB    A,B                 ; Exp OP2 – exp OP1 –> B
       BC     op1_gt_op2,BLT      ; Process OP1 > OP2
       BC     op2_gt_op1,BGT      ; Process OP2 > OP2
*
*;*****************************************************************************
*;       exp OP1 = exp OP2
*;  Mantissas of OP1 and OP2 are normalized identically.
*;  Add mantissas:  mant OP1 + mant OP2
*;  If result is zero, special case processing must be executed.
*;  Load exponent for possible adjustment during normalization of result
*;*****************************************************************************
a_eq_b
       DLD    op1_hm,A            ; Load OP1 mantissa
       DADD   op2_hm,A            ; Add OP2 mantissa
       BC     res_zero,AEQ        ; If result is zero, process special case
       LD     op1_se,B            ; Load exponent in preparation for normalizing
*
*;*****************************************************************************
*;    normalize THE RESULT
*;    Take the absolute value of the result.
*;    Set up to normalize the result.
*;    The MSB may be in any of bits 24 through 0.
*;    Left shift by six bits; bit 24 moves to bit 30, etc.
*;    Normalize resulting mantissa with exponent adjustment.
*;*****************************************************************************
*
```

Example 6–9. Add Two Floating-Point Numbers (Continued)

```
normalize
       STH    A,res_sign       ; Save signed mantissa on stack
       ABS    A                ; Create magnitude value of mantissa
       SFTL   A,6              ; Pre–normalize adjustment of mantissa
       EXP    A                ; Get amount to adjust exp for normalization
       NOP
       NORM   A                ; Normalize the result
       ST     T,res_exp        ; Store exp adjustment value
       ADD    #1,B             ; Increment exp to account for implied carry
       SUB    res_exp,B        ; Adjust exponent to account for normalization
*
*;*****************************************************************************
*;       POST–NORMALIZATION ADJUSTMENT AND STORAGE
*;  Test result for underflow and overflow.
*;  Right shift mantissa by 7 bits.
*;  Mask implied 1
*;  Store mantissa on stack.
*;*****************************************************************************
*
normalized
       STL    B,res_exp        ; Save result exponent on stack
       BC     underflow,BLEQ   ; process underflow if occurs
       SUB    #0FFh,B          ; adjust to check for overflow
       BC     overflow,BGEQ    ; process overflow if occurs
       SFTL   A,–7             ; Shift right to place mantissa for splitting
       STL    A,res_lm         ; Store low mantissa
       AND    #07F00h,8,A      ; Eliminate implied one
       STH    A,res_hm         ; Save result mantissa on stack**
;*****************************************************************************
*;*****************************************************************************
*
*;       CONVERSION OF FLOATING POINT FORMAT – PACK
*;  Load sign.
*;  Pack exponent.
*;  Pack mantissa.
*;*****************************************************************************
*
       LD     res_sign,9,A     ; 0000 000S 0000 0000 0000 0000 0000 0000
       AND    #100h,16,A
       ADD    res_exp,16,A     ; 0000 000S EEEE EEEE 0000 0000 0000 0000
       SFTL   A,7              ; SEEE EEEE E000 0000 0000 0000 0000 0000
       DADD   res_hm,A         ; SEEE EEEE EMMM MMMM MMMM MMMM MMMM MMMM
*
*;*****************************************************************************
*;       CONTEXT RESTORE
*;  Pop local floating point variables.
*;  Restore contents of B accumulator, T Register
*;*****************************************************************************
*
return_value
       NOP
       NOP
       RET
*
```

Example 6–9. Add Two Floating-Point Numbers (Continued)

```
*;*************************************************************************
*;       exp OP1 > exp OP2
*;  Test if the difference of the exponents is larger than 24 (precision of the mantissa)
*;  Return OP1 as the result if OP2 is too small.
*;  Mantissa of OP2 must be right shifted to match normalization of OP1
*;  Add mantissas:  mant OP1 + mant op2
*;*************************************************************************
*
op1_gt_op2
       ABS    B                   ; If exp OP1 >= exp OP2 + 24 then return OP1
       SUB    #24,B
       BC     return_op1,BGEQ
       ADD    #23,B               ; Restore exponent difference value
       STL    B,res_sign          ; Store exponent difference to be used as RPC
       DLD    op2_hm,A            ; Load OP2 mantissa
       RPT    res_sign            ; Normalize OP2 to match OP1
       SFTA   A,-1
       BD     normalize           ; Delayed branch to normalize result
       LD     op1_se,B            ; Load exponent value to prep for normalization
       DADD   op1_hm,A            ; Add OP1 to OP2
*
*;*************************************************************************
*;       OP1 < OP2
*; Test if the difference of the exponents is larger than 24 (precision of the mantissa).
*; Return OP2 as the result if OP1 is too small.
*; Mantissa of OP1 must be right shifted to match normalization of OP2.
*; Add mantissas:  mant OP1 + mant OP2
*;*************************************************************************
op2_gt_op1
       SU     B  #24,B            ; If exp OP2 >= exp OP1 + 24 then return OP2
       BC     return_op2,BGEQ
       ADD    #23,B               ; Restore exponent difference value
       STL    B,res_sign          ; Store exponent difference to be used as RPC
       DLD    op1_hm,A            ; Load OP1 mantissa
       RPT    res_sign            ; Normalize OP1 to match OP2
       SFTA   A,-1   BD normalize ; Delayed branch to normalize result
       LD     op2_se,B            ; Load exponent value to prep for normalization
       DADD   op2_hm,A            ; Add OP2 to OP1
*;*************************************************************************
*;       OP1 << OP2  or OP1 = 0
*;*************************************************************************
*
return_op2
op1_zero
       BD     return_value
       DLD    op2_msw,A           ; Put OP2 as result into A
       NOP
*
*;*************************************************************************
*;       OP1 << OP2  or  OP1 = 0
*;*************************************************************************
*
```

Example 6–9. Add Two Floating-Point Numbers (Continued)

```
op2_zero
return_op1
       DLD    op1_hm,A            ; Load signed high mantissa of OP1
       BC     op1_pos,AGT         ; If mantissa is negative . . .
       NEG    A                   ; Negate it to make it a positive value
       ADDM   #100h,op1_se        ; Place the sign value back into op1_se
op1_pos
       SUB    #80h,16,A           ; Eliminate implied one from mantissa
       LD     op1_se,16,B         ; Put OP1 back together in acc A as a result
       BD     return_value
       SFTL   B,7
       ADD    B,A
*;*****************************************************************************
*;       overflow PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
*
overflow
       ST     #2,err_no           ; Load error no
       LD     res_sign,16,A       ; Pack sign of result
       AND    #8000,16,A          ; Mask to get sign
       OR     #0FFFFh,A           ; Result low mantissa = 0FFFFh
       BD     return_value        ; Branch delayed
       ADD    #07F7Fh,16,A        ; Result exponent = 0FEh
                                  ; Result high mant = 07Fh
*****************************************************************************
*;       underflow PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
*
underflow
       ST     #1,err_no           ; Load error no
       RET
res_zero
       BD     return_value        ; Branch delayed
       SUB    A,A                 ; For underflow result = 0
       NOP
```

Example 6–10. Multiply Two Floating-Point Numbers

```
*;*************************************************************************
*;
*;  Float_MUL – multiply two floating point numbers
*;  Copyright (c) 1993–1994 Texas Instruments Incorporated
*;*************************************************************************
*;*************************************************************************
;
;This routine multiplies two floating point  numbers. OP1 and OP2 are each unpacked
;into sign, exponent, and two words of mantissa. If either exponent is zero
;special case processing is initiated. The exponents are summed. If the result is
;less than zero underflow has occurred. If the result is zero, underflow may have
;occurred. If the result is equal to 254 overflow may have occurred.  If the result
;is greater than 254 overflow has occurred. Underflow processing returns a value
;of zero. Overflow processing returns the largest magnitude value along with the
;appropriate sign. If no special cases are detected, a 24x24–bit multiply is
;executed. The result of the exclusive OR of the sign bits, the sum of the
;exponents and the ;24 bit truncated mantissa are packed and returned
*;*************************************************************************
*;
*;       resource utilization:  B accumulator, T–register
*;       status bits affected: TC, C, SXM, OVM, C16
*;       entry requirements : CPL bit set
*;*************************************************************************
;
; Floating Point Format – Single Precision
*-------------------------------------------------------------------------------
*| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*|  S | E7 | E6 | E5 | E4 | E3 | E2 | E1 | E0 | M22| M21| M20| M19| M18| M17| M16|
*-------------------------------------------------------------------------------
*-------------------------------------------------------------------------------
*| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*| M15| M14| M13| M12| M11| M10| M9 | M8 | M7 | M6 | M5 | M4 | M3 | M2 | M1 | M0 |
*-------------------------------------------------------------------------------
*;  Single precision floating point format is a 32 bit format  consisting of a    *
*;  1 bit sign field, an 8 bit exponent field, and a 23 bit mantissa field. The  *
*;  fields are defined as follows.                                               *
*;             Sign <S>          : 0 = positive values; 1 = negative values
*;             Exponent <E7–E0>  : offset binary format
*;                                 00 = special cases (i.e. zero)
*;                                 01 = exponent value + 127 = –126
*;                            FE = exponent value + 127 = +127
*;                                 FF = special cases (not implemented)
*;  Mantissa <M22–M0> : fractional magnitude format with implied 1
*;                                 1.M22M21...M1M0
*;             Range             : –1.9999998 e+127 to –1.0000000 e–126
*;                                 +1.0000000 e–126 to +1.9999998 e+
*;                                 (where e represents 2 to the power of)
*;                                 –3.4028236 e+38  to –1.1754944 e–
*;                                 +1.1754944 e–38  to +3.4028236 e+38
*;                                 (where e represents 10 to the power of)
*;*************************************************************************
```

Example 6–10. Multiply Two Floating-Point Numbers (Continued)

```
res_hm       .usect "flt_add",1            ;result high mantissa
res_lm       .usect "flt_add",1            ;result low mantissa
res_exp      .usect "flt_add",1            ;result exponent
res_sign     .usect "flt_add",1            ; result sign
op2_hm       .usect "flt_add",1            ; OP2 high mantissa
op2_lm       .usect "flt_add",1            ; OP2 low mantissa
op2_se       .usect "flt_add",1            ; OP2 sign and exponent
op1_se       .usect "flt_add",1            ; OP1 sign and exponent
op1_hm       .usect "flt_add",1            ; OP1 high mantissa
op1_lm       .usect "flt_add",1            ; OP1 low mantissa
op1_msw      .usect "flt_add",1            ; OP1 packed high word
op1_lsw      .usect "flt_add",1            ; OP1 packed low word
op2_msw      .usect "flt_add",1            ; OP2 packed high word
op2_lsw      .usect "flt_add",1            ; OP2 packed low word
err_no       .usect "flt_add",1            ;
*****************************************************************
* Floating point number 12.0 can be represented as 1100 = 1.100 x 23 => sign =0
*                                                    biased exponent = 127+3 = 130
*                                                    130 = 10000010
*                                                    Mantissa 10000000000000000000000
*  Thus 12.0 can be represented as 01000001010000000000000000000000= 4140h
*********************************************************************************
*
K_OP1_HIGH          .set      4140h        ; floating point number 12.0
K_OP1_LOW           .set      0000h
K_OP2_HIGH          .set      4140h        ; floating point number 12.0
K_OP2_LOW           .set      0000h
          .mmregs
          .text
start_flt:
          RSBX   C16                      ; Insure long adds for later
          LD     #res_hm,DP               ; initialize the page pointer
          LD     #K_OP2_HIGH,A            ; load floating #2 – 12
          STL    A,op2_msw
          LD     #K_OP2_LOW,A
          STL    A,op2_lsw
          LD     #K_OP1_HIGH,A            ; load floating #1 –  12
          STL    A,op1_msw
          LD     #K_OP1_LOW,A
          STL    A,op1_lsw
*
*;*****************************************************************************
*;
*;       CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;  Test OP1 for special case treatment of zero.
*;  Split the MSW of A in the accumulator.
*;    Save the sign and exponent on the stack [xxxx xxxS EEEE EEEE].
*;    Add the implied one to the mantissa value
*;    Store entire mantissa with a long word store
*;*****************************************************************************
*
          DLD    op1_msw,A                ; OP1
          SFTA   A,8
          SFTA   A,–8
          BC     op_zero,AEQ              ; if op1 is 0, jump to special case
```

Example 6–10. Multiply Two Floating-Point Numbers (Continued)

```
          STH    A,–7,op1_se         ; store sign AND exponent to stack
          STL    A,op1_lm            ; store low mantissa
          AND    #07Fh,16,A          ; mask off sign & exp to get high mantissa
          ADD    #080h,16,A          ; ADD implied 1 to mantissa
          STH    A,op1_hm            ; store mantissa to stack
*;*****************************************************************************
*;       CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;  Test OP2 for special case treatment of zero.
*;  Split the MSW of A in the accumulator.
*;    Save the sign and exponent on the stack [xxxx xxxS EEEE EEEE].
*;    Add the implied one to the mantissa value.
*;    Store entire mantissa with a long word store
*;*****************************************************************************
          DLD    op2_msw,A           ; load acc a with OP2
          BC     op_zero,AEQ         ; if OP2 is 0, jump to special case
          STH    A,–7,op2_se         ; store sign and exponent to stack
          STL    A,op2_lm            ; store low mantissa
          AND    #07Fh,16,A          ; mask off sign & exp to get high mantissa
          ADD    #080h,16,A          ; add implied 1 to mantissa
          STH    A,op2_hm            ; store mantissa to stack
*;*****************************************************************************
*;       SIGN EVALUATION
*;  Exclusive OR sign bits of OP1 and OP2 to determine sign of result.
*;*****************************************************************************
          LD     op1_se,A            ; load sign and exp of op1 to acc
          XOR    op2_se,A            ; xor with op2 to get sign of result
          AND    #00100h,A           ; mask to get sign
          STL    A,res_sign          ; save sign of result to stack
*;*****************************************************************************
*;       EXPONENT SUMMATION
*;  Sum the exponents of OP1 and OP2 to determine the result exponent.  Since
*;  the exponents are biased (excess 127) the summation must be decremented
*;  by the bias value to avoid double biasing the result
*;  Branch to one of three blocks of processing
*;    Case 1:  exp OP1 + exp OP2 results in underflow (exp < 0)
*;    Case 2:  exp OP1 + exp OP2 results in overflow (exp >= 0FFh)
*;    Case 3:  exp OP1 + exp OP2 results are in range (exp >= 0 & exp < 0FFh)
*;      NOTE:  Cases when result exp = 0 may result in underflow unless there
*;             is a carry in the result that increments the exponent to 1.
*;             Cases when result exp = 0FEh may result in overflow if there
*;             is a carry in the result that increments the exponent to 0FFh.
*;*****************************************************************************
          LD     op1_se,A            ; Load OP1 sign and exponent
          AND    #00FFh,A            ; Mask OP1 exponent
          LD     op2_se,B            ; Load OP2 sign and exponent
          AND    #0FFh,B             ; Mask OP2 exponent
          SUB    #07Fh,B             ; Subtract offset (avoid double bias)
          ADD    B,A                 ; Add OP1 exponent
          STL    A,res_exp           ; Save result exponent on stack
          BC     underflow,ALT       ; branch to underflow handler if exp < 0
          SUB    #0FFh,A             ; test for overflow
          BC     overflow,AGT        ; branch to overflow is exp > 127
*;*****************************************************************************
```

Example 6–10. Multiply Two Floating-Point Numbers (Continued)

```
*;       MULTIPLICATION
*; Multiplication is implemented by parts.  Mantissa for OP1 is three bytes
*; identified as Q, R, and S
*; (Q represents OP1 high mantissa and R and S represent the two bytes of OP1 low
*; mantissa).  Mantissa for
*; OP2 is also 3  bytes identified as X, Y, and Z (X represents OP2 high mant and
*; Y and Z represent the two bytes
*; of OP2 low mantissa).  Then
*;                       0  Q  R  S      (mantissa of OP1)
*;                   x  0  X  Y  Z      (mantissa of OP2)
*;                     ===========
*;                       RS*YZ         <-- save only upper 16 bits of result
*;                       RS*0X
*;                       0Q*YZ
*;                       0Q*0X         <-- upper 16 bits are always zero
*;                     ===========
*;                       result        <-- result is always in the internal 32 bits
*;(which ends up in the accumulator) of the possible 64 bit product
*;*************************************************************************
;
          LD     op1_lm,T         ; load low mant of op1 to T register
          MPYU   op2_lm,A         ; RS * YZ
          MPYU   op2_hm,B         ; RS * 0X
          ADD    A,-16,B          ; B = (RS * YZ) + (RS * 0X)
          LD     op1_hm,T         ; load high mant of op1 to T register
          MPYU   op2_lm,A         ; A = 0Q * YZ
          ADD    B,A              ; A = (RS * YZ) + (RS * 0X) + (0Q * YZ)
          MPYU   op2_hm,B         ; B = 0Q * 0X
          STL    B,res_hm         ; get lower word of 0Q * 0X
          ADD    res_hm,16,A      ; A = final result
*;*************************************************************************
*;       POST-NORMALIZATION ADJUSTMENT AND STORAGE
*;  Set up to adjust the normalized result.
*;    The MSB may be in bit 31.  Test this case and increment the exponent
*;    and right shift mantissa 1 bit so result is in bits 30 through 7
*;  Right shift mantissa by 7 bits.
*;  Store low mantissa on stack.
*;  Mask implied 1 and store high mantissa on stack.
*;  Test result for underflow and overflow.
*************************************************************************
          ADD    #040h,A          ; Add rounding bit
          SFTA   A,8              ; sign extend result to check if MSB is in 31
          SFTA   A,-8
          RSBX   SXM              ; turn off sign extension for normalization
          LD     res_exp,B        ; load exponent of result
          BC     normalized,AGEQ  ; check if MSB is in 31
          SFTL   A,-1             ; Shift result so result is in bits 30:7
          ADD    #1,B             ; increment exponent
          STL    B,res_exp        ; save updated exponent normalized
          BC     underflow,BLEQ   ; check for underflow
          SUB    #0FFh,B          ; adjust to check for overflow
          BC     overflow,BGEQ    ; check for overflow
          SFTL   A,-7             ; shift to get 23 msb bits of mantissa result
          STL    A,res_lm         ; store low mantissa result
```

Example 6–10. Multiply Two Floating-Point Numbers (Continued)

```
          AND    #07F00h,8,A      ; remove implied one
          STH    A,res_hm         ; store the mantissa result
*;*****************************************************************************
*;
*;        CONVERSION OF FLOATING POINT FORMAT – PACK
*;  Load sign.
*;  Pack exponent.
*;  Pack mantissa.
*;*****************************************************************************
          LD     res_sign,16,A    ; 0000 000S 0000 0000 0000 0000 0000 0000
          ADD    res_exp,16,A     ; 0000 000S EEEE EEEE 0000 0000 0000 0000
          SFTL   A,7              ; SEEE EEEE E000 0000 0000 0000 0000 0000
          DADD   res_hm,A         ; SEEE EEEE EMMM MMMM MMMM MMMM MMMM MMMM
*;*****************************************************************************
*;
*;        CONTEXT RESTORE
*;*****************************************************************************
return_value
op_zero
          nop
          nop
          ret
*;*****************************************************************************
*;  overflow PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
overflow
          ST     #2,err_no        ; Load error no
          LD     res_sign,16,B    ; Load sign of result
          LD     #0FFFFh,A        ; Result low mantissa = 0FFFFh
          OR     B,7,A            ; Add sign bit
          BD     return_value     ; Branch delayed
          ADD    #07F7Fh,16,A     ; Result exponent = 0FEh
                                  ; Result high mant = 07Fh
*;*****************************************************************************
*;       UNDERFLOW PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
underflow
          ST     #1,err_no        ; Load error no
          BD     return_value     ; Branch delayed
          SUB    A,A              ; For underflow result = 0
          NOP
```

Example 6–11. Divide a Floating-Point Number by Another

```
*;***************************************************************************
*;  FLOAT_DIV – divide two floating point numbers
*;  Copyright (c) 1993–1994 Texas Instruments Incorporated
*;***************************************************************************
;Implementation: OP1 and OP2 are each unpacked into sign, exponent, and two words
;of mantissa.  If either exponent is zero special case processing is initiated.
;The difference of the exponents are taken. IF the result is less than zero underflow
;has occurred. If the result is zero, underflow may have occurred. If the result
;is equal to 254 overflow may have occurred.  If the result is greater than 254
;overflow has occurred.
; Underflow processing returns a value of zero. Overflow processing returns the
;largest magnitude value along with the appropriate sign.  If no special cases are
;detected, a 24x24–bit divide is ;executed.  The result of the exclusive OR of the
;sign bits, the difference of the exponents and the 24 bit truncated mantissa are
;packed and returned.
*;***************************************************************************
*;***************************************************************************
*;       resource utilization:  B accumulator , T register
*;       status bits affected: TC, C, SXM, OVM, C16
*;       entry requirements : CPL bit set
*;***************************************************************************
; Floating Point Format – Single Precision
*-------------------------------------------------------------------------------
*| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*|  S | E7 | E6 | E5 | E4 | E3 | E2 | E1 | E0 | M22| M21| M20| M19| M18| M17| M16|
*-------------------------------------------------------------------------------
*-------------------------------------------------------------------------------
*| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*| M15| M14| M13| M12| M11| M10| M9 | M8 | M7 | M6 | M5 | M4 | M3 | M2 | M1 | M0 |
*-------------------------------------------------------------------------------
*; Single precision floating point format is a 32 bit format  consisting of a 1
bit sign field, an 8 bit exponent  *
*; field, and a 23 bit mantissa field. The fields are defined as follows
*
*;            Sign <S>          : 0 = positive values; 1 = negative values
*;            Exponent <E7–E0>  : offset binary format
*;                                00 = special cases (i.e. zero)
*;                             01 = exponent value + 127 = –126
*;                                FE = exponent value + 127 = +127
*;                                FF = special cases (not implemented)
*;  Mantissa <M22–M0> : fractional magnitude format with implied 1
*;                                1.M22M21...M1M0
*;            Range             : –1.9999998 e+127 to –1.0000000 e–126
*;                                +1.0000000 e–126 to +1.9999998 e+127
*;                                (where e represents 2 to the power of)
*;                                –3.4028236 e+38  to –1.1754944 e–38
*;                                +1.1754944 e–38  to +3.4028236 e+
*;                                (where e represents 10 to the power of)
*;***************************************************************************
```

Example 6–11. Divide a Floating-Point Number by Another (Continued)

```
res_hm              .usect    "flt_div",1
res_lm              .usect    "flt_div",1
res_exp             .usect    "flt_div",1
res_sign            .usect    "flt_div",1
op2_hm              .usect    "flt_div",1
op2_lm              .usect    "flt_div",1
op2_se              .usect    "flt_div",1
op1_se              .usect    "flt_div",1
op1_hm              .usect    "flt_div",1
op1_lm              .usect    "flt_div",1
op1_msw             .usect    "flt_div",1
op1_lsw             .usect    "flt_div",1
op2_msw             .usect    "flt_div",1
op2_lsw             .usect    "flt_div",1
err_no              .usect    "flt_div",1
      .mmregs
*
*
K_divisor_high      .set      4140h
K_divisor_low       .set      0000h
K_dividend_high     .set      4140h
K_dividend_low      .set      0000h
                    .sect     "vectors"
      B             float_div
      NOP
      NOP
      .text
float_div:
      LD    #res_hm,DP              ; initialize the page pointer
      LD    #K_divisor_high,A       ; load floating #2 – 12
      STL   A,op2_msw
      LD    #K_divisor_low,A
      STL   A,op2_lsw
      LD    #K_dividend_high,A      ; load floating #1 –  12
      STL   A,op1_msw
      LD    #K_dividend_low,A
      STL   A,op1_lsw
*************************************************************
      RSBX      C16                 ; Insure long adds for later
*
*;*****************************************************************************
*;        CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;  Test OP1 for special case treatment of zero.
*;  Split the MSW of A in the accumulator.
*;    Save the sign and exponent on the stack [xxxx xxxS EEEE EEEE].
*;    Add the implied one to the mantissa value.
*;    Store entire mantissa with a long word store
*;*****************************************************************************
      DLD   op1_msw,A               ; load acc a with OP1
      SFTA  A,8
      SFTA  A,–8
      BC    op1_zero,AEQ            ; if op1 is 0, jump to special case
      STH   A,–7,op1_se             ; store sign and exponent to stack
```

Example 6–11. Divide a Floating-Point Number by Another (Continued)

```
       STL    A,op1_lm            ; store low mantissa
       AND    #07Fh,16,A          ; mask off sign & exp to get high mantissa
       ADD    #080h,16,A          ; ADD implied 1 to mantissa
       STH    A,op1_hm            ; store mantissa to stack
*
*;*****************************************************************************
*;        CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;  Test OP1 for special case treatment of zero.
*;  Split the MSW of A in the accumulator.
*;    Save the sign and exponent on the stack [xxxx xxxS EEEE EEEE].
*;    Add the implied one to the mantissa value.
*;    Store entire mantissa with a long word store
*;*****************************************************************************
*
       DLD    op2_msw,A           ; load acc a with OP2
       BC     op2_zero,AEQ        ; if OP2 is 0, divide by zero
       STH    A,–7,op2_se         ; store sign and exponent to stack
       STL    A,op2_lm            ; store low mantissa
       AND    #07Fh,16,A          ; mask off sign & exp to get high mantissa
       ADD    #080h,16,A          ; ADD implied 1 to mantissa
       STH    A,op2_hm            ; store mantissa to stack
*
*;*****************************************************************************
*;        SIGN EVALUATION
*;  Exclusive OR sign bits of OP1 and OP2 to determine sign of result.
*;*****************************************************************************
*      LD     op1_se,A            ; load sign and exp of op1 to acc
       XOR    op2_se,A            ; xor with op2 to get sign of result
       AND    #00100h,A           ; mask to get sign
       STL    A,res_sign          ; save sign of result to stack
*
*;*****************************************************************************
*;        EXPONENT SUMMATION
*;  Find difference between operand exponents to determine the result exponent. *
*   Since the subtraction process removes the bias it must be re–added in.      *
*
*;  Branch to one of three blocks of processing
*;    Case 1:  exp OP1 + exp OP2 results in underflow (exp < 0)
*;    Case 2:  exp OP1 + exp OP2 results in overflow (exp >= 0FFh)
*;    Case 3:  exp OP1 + exp OP2 results are in range (exp >= 0 & exp < 0FFh)
*;        NOTE:  Cases when result exp = 0 may result in underflow unless there *
*         is a carry in the result that increments         the exponent to 1.   *
*         Cases when result exp = 0FEh may result in overflow if there is a carry  *
*         in the result that increments the exponent to 0FFh.
*;*****************************************************************************
*
       LD     op1_se,A            ; Load OP1 sign and exponent
       AND    #0FFh,A             ; Mask OP1 exponent
*
       LD     op2_se,B            ; Load OP2 sign and exponent
       AND    #0FFh,B             ; Mask OP2 exponent
*
```

Example 6–11. Divide a Floating-Point Number by Another (Continued)

```
       ADD    #07Fh,A          ; Add offset (difference eliminates offset)
       SUB    B,A              ; Take difference between exponents
       STL    A,res_exp        ; Save result exponent on stack
*
       BC     underflow,ALT    ; branch to underflow handler if exp < 0
       SUB    #0FFh,A          ; test for overflow
       BC     overflow,AGT     ; branch to overflow is exp > 127
*
*;*************************************************************************
*;         DIVISION
*; Division is implemented by parts. The mantissas for both OP1 and OP2 are left shifted
*  in the  32 bit field to reduce the effect of secondary and tertiary contributions to
*  the final result. The left shifted results are identified as OP1'HI, OP1'LO, OP2'HI,
*  and OP2'LO where OP1'HI and OP2'HI have the xx most significant bits of the mantissas
*  and OP1'LO and OP2'LO contain the remaining bits *  of each mantissa. Let QHI and QLO
*  represent the two portions of the resultant mantissa. Then
*       QHI + QLO = (OPI'HI + OPI'LO)/(OP2'HI + OP2'LO) = (OPI'HI + OPI'LO)/OP2'HI * 1/(1 + OP2'LO/OP2'HI)
*;     Now let   X = OP2'LO/OP2'HI
*;     Then by Taylor's Series Expansion
*       1/(1 + x) = 1–x + x^2–x^3 + ........
*;     Since OP2'HI contains the first xx significant bits of the OP2 mantissa,*
X = OP2'LO/OP2'HI < 2–yy*;    Therefore the X2 term and all subsequent terms are less
than the least significant
*      bit of the 24–bit result and can be dropped. The result then becomes
*       QHI + QLO = (OPI'HI + OPI'LO)/(OP2'HI + OP2'LO) * (1–OP2'LO/OP2'HI)
                  = (QHI + QLO) * (1–OP2'LO/OP2'HI)
*;     where Q'HI and Q'LO represent the first approximation of the result. Also since
*      Q'LO and OP2'LO/OP2'HI are less significant the 24th bit of the result, this
*      product term can be dropped so
    *       QHI + QLO = (OPI'HI + OPI'LO)/(OP2'HI + OP2'LO) = (OPI'HI + OPI'LO)/OP2'HI * 1/(1 + OP2'LO/OP2'HI)
that
*;*************************************************************************
       DLD    op1_hm,A         ; Load dividend mantissa
       SFTL   A,6              ; Shift dividend in preparation for division
*
       DLD    op2_hm,B         ; Load divisor mantissa
       SFTL   B,7              ; Shift divisor in preparation for division
       DST    B,op2_hm         ; Save off divisor
*
       RPT    #14              ; QHI = OP1'HI/OP2'HI
       SUBC   op2_hm,A
       STL    A,res_hm         ; Save QHI
*
       SUBS   res_hm,A         ; Clear QHI from ACC
       RPT    #10              ; Q'LO = OP1'LO / OP2'HI
       SUBC   op2_hm,A
```

Example 6–11. Divide a Floating-Point Number by Another (Continued)

```
       STL    A,5,res_lm          ; Save Q'LO*
       LD     res_hm,T            ; T = Q'HI
       MPYU   op2_lm,A            ; Store Q'HI * OP2'LO in acc A
       SFTL   A,–1                ;*
       RPT    #11                 ; Calculate Q'HI * OP2'LO / OP2'HI
       SUBC   op2_hm,A            ;   (correction factor)
       SFTL   A,4                 ; Left shift to bring it to proper range
       AND    #0FFFFh,A           ; Mask off correction factor
*
       NEG A                      ; Subtract correction factor
       ADDS   res_lm,A            ; Add Q'LO
       ADD    res_hm,16,A         ; Add Q'HI
*
*;*****************************************************************************
*;        POST–NORMALIZATION ADJUSTMENT AND STORAGE
*;  Set up to adjust the normalized result. The MSB may be in bit 31. Test this
case and increment the exponent and right shift mantissa 1 bit so result is in
bits 30 through 7. Right shift mantissa by 7 bits. Store low mantissa on stack.
Mask implied 1 and store high mantissa on stack. Test result for underflow and
overflow.
*;*****************************************************************************
*
       LD     res_exp,B           ; Load result exponent
       EXP    A                   ; Get amount to adjust exp for normalizationNOP
       NORM   A                   ; Normalize the result
       ST     T,res_exp           ; Store the exponent adjustment value
       SUB    res_exp,B           ; Adjust exponent (add either zero or one)
       SFTL   A,–1                ; Pre–scale adjustment for rounding
       ADD    #1,B                ; Adjust exponent
       ADD    #020h,A             ; Add rounding bit
       EXP    A                   ; Normalize after rounding     NOP
       NORM   A                   ;
       ST     T,res_exp           ; Adjust exponent for normalization
       SUB    res_exp,B           ;
       STL    B,res_exp           ; Save exponent
       BC     underflow,BLEQ      ; process underflow if occurs
       SUB    #0FFh,B             ; adjust to check for overflow
       BC     overflow,BGEQ       ; process overflow if occurs
       SFTL   A,–7                ; Shift right to place mantissa for splitting
       STL    A,res_lm            ; Save result low mantissa
       AND    #07F00h,8,          ; Eliminate implied one
       STH    A,res_hm            ; Save result mantissa on stack
*
*;*****************************************************************************
*;        CONVERSION OF FLOATING POINT FORMAT – PACK
*;  Load sign.
*;  Pack exponent.
*;  Pack mantissa.
*;*****************************************************************************
*
```

Example 6–11. Divide a Floating-Point Number by Another (Continued)

```
       LD     res_sign,16,A       ; 0000 000S 0000 0000 0000 0000 0000 0000
       ADD    res_exp,16,A        ; 0000 000S EEEE EEEE 0000 0000 0000 0000
       SFTL   A,7                 ; SEEE EEEE E000 0000 0000 0000 0000 0000
       DADD   res_hm,A            ; SEEE EEEE EMMM MMMM MMMM MMMM MMMM MMMM
**;*****************************************************************************
*;        CONTEXT RESTORE
*;*****************************************************************************
return_value
op1_zero
   ret
*
*;*****************************************************************************
*;        OVERFLOW PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
overflow
       ST     #2,err_no           ; Load error no
       SAT    A                   ; Result exponent = 0FEh
       SUB    #081h,16,A          ; Result high mant = 07Fh
       BD     return_value        ; Branch delayed
       LD     res_sign,16,B       ; Load sign of result
       OR     B,7,A               ; Pack sign*
*;*****************************************************************************
*;
UNDERFLOW PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
*
underflow
       ST     #1,err_no           ; Load error no
       BD     return_value        ; Branch delayed
       sub    A,A                 ; For underflow result = 0
       nop
**;
*****************************************************************************
*; DIVIDE BY ZERO
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
op2_zero
       ST     #3,err_no           ; Load error no
       SAT    A                   ; Result exponent = FEh
                                  ; Result low mant = FFFFh
       LD     op1_se,16,B         ; Load sign and exponent of OP1
       AND    #100h,16,B          ; Mask to get sign of OP1
       OR     B,7,A               ; Pack sign
       BD     return_value        ; Branch delayed
       SUB    #081h,16,A          ; Result high mant = 7Fh
       NOP
```

6.6 Logical Operations

DSP-application systems perform many logical operations, including bit manipulation and packing and unpacking data. A digital modem uses a scrambler and a descrambler to perform bit manipulation. The input bit stream is in a packed format of 16 bits. Each word is unpacked into 16 words of 16-bit data, with the most significant bit (MSB) as the original input bit of each word. The unpack buffer contains either 8000h or 0000h, depending upon the bit in the original input-packed 16-bit word. The following polynomial generates a scrambled output, where the $\oplus$ sign represents modulus 2 additions from the bitwise exclusive OR of the data values:

$$\text{Scrambler output} = 1 \oplus x^{-18} \oplus x^{-23}$$

The same polynomial sequence in the descrambler section reproduces the original 16-bit input sequence. The output of the descrambler is a 16-bit word in packed format.

Example 6–12. Pack/Unpack Data in the Scrambler/Descrambler of a Digital Modem

```
; TEXAS INSTRUMENTS INCORPORATED
          .mmregs
                        .asg         AR1,UNPACK_BFFR
                        .asg         AR3,SCRAM_DATA_18
                        .asg         AR4,SCRAM_DATA_23
                        .asg         AR2,DE_SCRAM_DATA_18
                        .asg         AR5,DE_SCRAM_DATA_23
d_scram_bffr            .usect       ”scrm_dat”,30
d_de_scram_bffr         .usect       ”dscrm_dt”,30
d_unpack_buffer         .usect       ”scrm_var”,100
d_input_bit             .usect       ”scrm_var”,1
d_pack_out              .usect       ”scrm_var”,1
d_asm_count             .usect       ”scrm_var”,1
K_BFFR_SIZE             .set         24
K_16                    .set         16
                        .def         d_input_bit
                        .def         d_asm_count
; Functional Description
;  This routine illustrates the pack and unpack of a data stream and
;  also bit manipulation. A digital scrambler and descrambler does the
;  bit manipulation and the input to the scrambler is in unpacked format
;  and the output of the descrambler is in packed 16–bit word.
;  scrambler_output = 1+x^–18+x^–23
;  additions are modulus 2 additions or bitwise exclusive OR of data
;  values. The same polynomial is used to generate the descrambler
;  output.
       .sect     ”scramblr”
scrambler_init:
       STM    #d_unpack_buffer,UNPACK_BFFR
       STM    #d_scram_bffr,SCRAM_DATA_23
       RPTZ   A,#K_BFFR_SIZE
       STL    A,*SCRAM_DATA_23+
       STM    #d_scram_bffr+K_BFFR_SIZE–1,SCRAM_DATA_23
```

Example 6–12. Pack/Unpack Data in the Scrambler/Descrambler of a Digital Modem (Continued)

```
       STM    #d_scram_bffr+17,SCRAM_DATA_18
       STM    #d_de_scram_bffr+K_BFFR_SIZE–1,DE_SCRAM_DATA_23
       STM    #d_de_scram_bffr+17,DE_SCRAM_DATA_18
       LD     #d_input_bit,Dp
       ST     #–K_16+1,d_asm_count
scramler_task:
;  the unpack data buffer has either 8000h or 0000h since the bit stream
;  is either 1 or 0
unpack_data:
       STM    #K_16–1,BRC
       RPTB   end_loop–1                       ; unpack the data into 16–bit
                                               ; word
       PORTR  1h,d_input_bit                   ; read the serial bit stream
       LD     d_input_bit,15,A                 ; mask thelower 15 bits
                                               ; the MSB is the serial bit
                                               ; stream
       STL    A,*UNPACK_BFFR                   ; store the 16 bit word
unpack_16_words
scrambler:
       LD     *SCRAM_DATA_18–%,A
       XOR    *SCRAM_DATA_23,A                 ; A = x^–18+x^–23
       XOR    *UNPACK_BFFR,A                   ; A = A+x^0
       STL    A,*SCRAM_DATA_23–%               ; newest sample, for next
                                               ; cycle it will be x(n–1)
       STL    A,*UNPACK_BFFR                   ; store the scrambled data
scramble_word
descrambler:
       LD     *DE_SCRAM_DATA_18–%,A
       XOR    *DE_SCRAM_DATA_23,A              ; A = x^–18+x^–23
       XOR    *UNPACK_BFFR,A                   ; A = A+x^0
       STL    A,*DE_SCRAM_DATA_23–%            ; newest sample, for next
                                               ; cycle it will be x(n–1)
       STL    A,*UNPACK_BFFR                   ; store the scrambled data
de_scramble_word
; ASM field shifts the descrambler output MSB into proper bit position
;
pack_data
       RSBX   SXM                              ; reset the SXM bit
       LD     d_asm_count,ASM
       LD     *UNPACK_BFFR+,A
       LD     A,ASM,A
       OR     d_pack_out,A                     ; start pack the data
       STL    A,  d_pack_out
       ADDM   #1,d_asm_count
pack_word
       SSBX   SXM                              ; enable SXM mode
end_loop
       NOP                                     ; dummy instructions nothing
                                               ; with the code
       NOP
       .end
```

Chapter 7

Application-Specific Examples

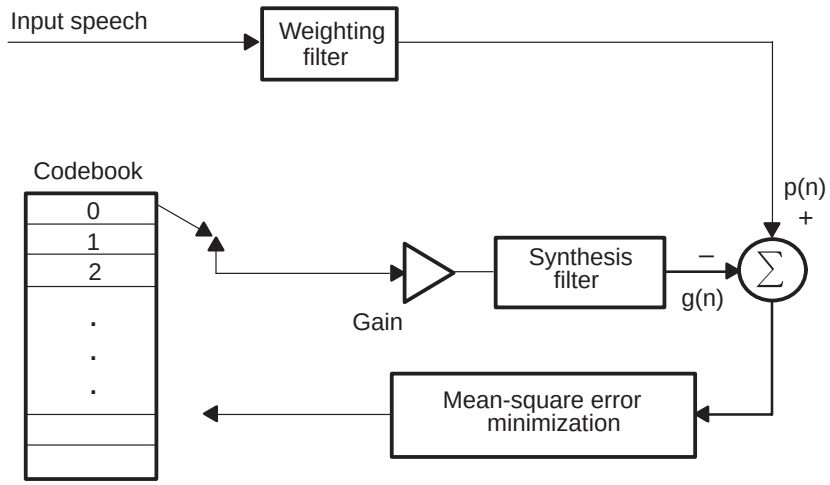
This chapter shows examples of typical applications for the ’54x. Since this DSP is widely used for speech coding and telecommunications, the applications show how some aspects of these functions are implemented.

Topic **Page**

7.1 Codebook Search for Excitation Signal in Speech Coding

A code-excited linear predictive (CELP) speech coder is widely used for applications requiring speech coding with a bit rate under 16K bps. The speech coder uses a vector quantization technique from codebooks to an excitation signal. This excitation signal is applied to a linear predictive-coding (LPC) synthesis filter. To obtain optimum code vectors from the codebooks, a codebook search is performed, which minimizes the mean-square error generated from weighted input speech and from the zero-input response of a synthesis filter. Figure 7–1 shows a block diagram of a CELP-based speech coder.

Figure 7–1. CELP-Based Speech Coder



To locate an optimum code vector, the codebook search uses Equation 7–1 to minimize the mean-square error.

Equation 7–1. Optimum Code Vector Localization

$$E_i = \sum_{i=0}^{N-1} \{p(n) - \gamma_i g_i\ (n)\}^2 \qquad N : \text{Subframe}$$

The variable p(n) is the weighted input speech, $g_i(n)$ is the zero-input response of the synthesis filter, and γ_i is the gain of the codebook.

The cross-correlation (c_i) of p(n) and $g_i(n)$ is represented by Equation 7–2. The energy (G_i) of $g_i(n)$ is represented by Equation 7–3.

Equation 7–2. Cross Correlation Variable (c_i)

$$c_i = \sum_{i=0}^{N-1} g_i * p(n)$$

Equation 7–3. Energy Variable (G_i)

$$G_i = \sum_{i=0}^{N-1} g_i{}^2$$

Equation 7–1 is minimized by maximizing c_i^2 / G_i. Therefore, assuming that a code vector with i = opt is optimal, Equation 7–4 is always met for any i. The codebook search routine evaluates this equation for each code vector and finds the optimum one.

Equation 7–4. Optimal Code Vector Condition

$$\left(\frac{c_i^2}{G_i}\right) \leq \left(\frac{c_{opt}^2}{G_{opt}}\right)$$

Example 7–1 shows the implementation algorithm for codebook search on ’54x. The square (SQUR), multiply (MPYA), and conditional store (SRCCD, STRCD, SACCD) instructions are used to minimize the execution cycles. AR5 points to c_i and AR2 points to G_i. AR3 points to the locations of G_{opt} and c_{opt}^2. The value of i(opt) is stored at the location addressed by AR4.

Example 7–1. Codebook Search

```
            .title"CODEBOOK SEARCH"
            .mmregs
            .text
SEARCH:
            STM    #C,AR5           ;Set C(i) address
            STM    #G,AR2           ;Set G(i) address
            STM    #OPT,AR3         ;Set OPT address
            STM    #IOPT,AR4        ;Set IOPT address
            ST     #0,*AR4          ;Initialize lag
            ST     #1,*AR3+         ;Initialize Gopt
            ST     #0,*AR3–         ;Initialize C2opt
            STM    #N–1,BRC
            RPTB   Srh_End–1
            SQUR   *AR5+,A          ;A = C(i) * C(i)
            MPYA   *AR3+            ;B = C(i)^2 * Gopt
            MAS    *AR2+,*AR3–,B    ;B = C(i)^2 * Gopt –
                                    ;G(i) * C2opt,T = G(i)
            SRCCD  *AR4,BGEQ        ;if(B >= 0) then
                                    ;iopt = BRC
            STRCD  *AR3+,BGEQ       ;if(B >= 0) then
                                    ;Gopt = T
            SACCD  A,*AR3–,BGEQ     ;if(B >= 0) then
                                    ;C2opt = A NOP
            NOP                     ;To save current BCR
Srh_End:                            ;*AR4 –> optimal index
            RET
            .end
```

7.2 Viterbi Algorithm for Channel Decoding

Convolutional encoding with the Viterbi decoding algorithm is widely used in telecommunication systems for error control coding. The Viterbi algorithm requires a computationally intensive routine with many add-compare-select (ACS) iterations. The '54x can perform fast ACS operations because of dedicated hardware and instructions that support the Viterbi algorithm on chip. This implementation allows the channel decoder and the equalizer in communication systems to be used efficiently.

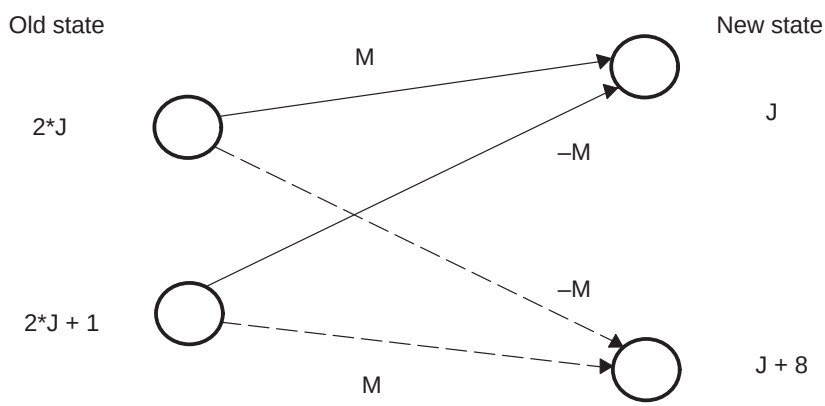
In the global system for mobile communications (GSM) cellular radio, the polynomials in Equation 7–5 are used for convolutional encoding.

Equation 7–5. Polynomials for Convolutional Encoding

$$G1(D) = 1 + D^3 + D^4 \qquad G2(D) = 1 + D + D^3 + D^4$$

This convolutional encoding can be represented in a trellis diagram, which forms a butterfly structure as shown in Figure 7–2. The trellis diagram illustrates all possible transformations of convolutional encoding from one state to another, along with their corresponding path states. There are 16 states, or eight butterflies, in every symbol time interval. Two branches are input to each state. Decoding the convolutional code involves finding the optimal path by iteratively selecting possible paths in each state through a predetermined number of symbol time intervals. Two path metrics are calculated by adding branch metrics to two old-state path metrics and the path metric (J) for the new state is selected from these two path metrics.

Figure 7–2. Butterfly Structure of the Trellis Diagram



Equation 7–6 defines a branch metric.

Equation 7–6. Branch Metric

$$M = SD(2*i) * B(J,0) + SD(2*i+1) * B(J,1)$$

SD(2*i) is the first symbol that represents a soft-decision input and SD(2*i+1) is the second symbol. B(J,0) and B(J,1) correspond to the code generated by the convolutional encoder as shown in Table 7–1.

Table 7–1. Code Generated by the Convolutional Encoder

J	B(J,0)	B(J,1)
0	1	1
1	–1	–1
2	1	1
3	–1	–1
4	1	–1
5	–1	1
6	1	–1
7	–1	1

The ’54x can compute a butterfly quickly by setting the ALU to dual 16-bit mode. To determine the new path metric (J), two possible path metrics from 2*J and 2*J+1 are calculated in parallel with branch metrics (M and –M) using the DADST instruction. The path metrics are compared by the CMPS instruction.

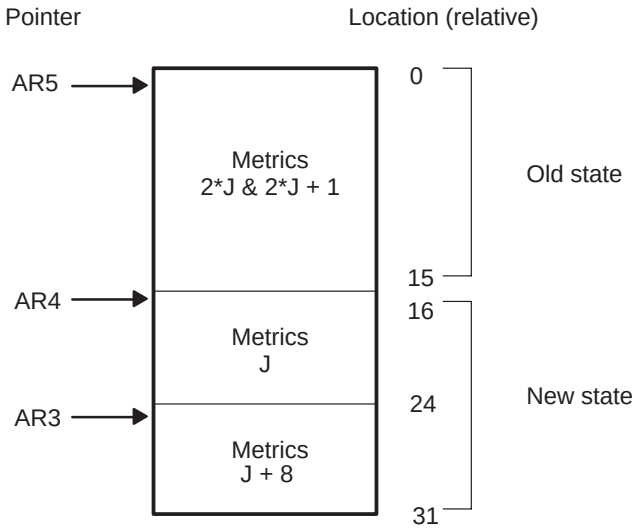
To calculate the new path metric (J+8), the DSADT instruction calculates two possible path metrics using branch metrics and old path metrics stored in the upper half and lower half of the accumulator. The CMPS instruction determines the new path metric.

The CMPS instruction compares the upper word and the lower word of the accumulator and stores the larger value in memory. The 16-bit transition register (TRN) is updated with every comparison so you can track the selected path metric. The TRN contents must be stored in memory locations after processing each symbol time interval. The back-track routine uses the information in memory locations to find the optimal path.

Example 7–2 shows the Viterbi butterfly macro. A branch metric value is stored in T before calling the macro. During every butterfly cycle, two macros prevent T from receiving opposite sign values of the branch metrics. Figure 7–3 illustrates pointer management and the storage scheme for the path metrics used in Example 7–2.

In one symbol time interval, eight butterflies are calculated for the next 16 new states. This operation repeats over a number of symbol time intervals. At the end of the sequence of time intervals, the back-track routine is performed to find the optimal path out of the 16 paths calculated. This path represents the bit sequence to be decoded.

Figure 7–3. Pointer Management and Storage Scheme for Path Metrics



Example 7–2. Viterbi Operator for Channel Coding

```
VITRBF    .MACRO            ;
          DADST  *AR5,A     ;A = OLD_M(2*J)+T//OLD_(2*J+1)-T
          DSADT  *AR5+,B    ;B = OLD_M(2*J)-T//OLD_(2*J+1)+T
          CMPS   A,*AR4+    ;NEW_M(J) = MAX(A_HIGH,A_LOW)
                            ;TRN<<1, TRN(0,0) = TC
          CMPS   B,*AR3+    ;NEW_M(J+8) = MAX(B_HIGH,B_LOW)
                            ;TRN<<1, TRN(0,) = TC
          .ENDM
VITRBR    .MACRO            ;
          DSADT  *AR5,A     ;A = OLD_M(2*J)-T//OLD_(2*J+1)+T
          DADST  *AR5+,B    ;B = OLD_M(2*J)+T//OLD_(2*J+1)-T
          CMPS   A,*AR4+    ;NEW_M(J) = MAX(A_HIGH,A_LOW)
                            ;TRN<<1, TRN(0,0) = TC
          CMPS   B,*AR3+    ;NEW_M(J+8) = MAX(B_HIGH,B_LOW)
                            ;TRN<<1, TRN(0,) = TC
          .ENDM
```

Chapter 8

Bootloader

The bootloader lets you load and execute programs received from a host processor, EPROMs, or other standard memory devices. The ’54x devices provide different ways to download the code to accommodate various system requirements. Some applications use a serial interface. If the code exists in external ROM, a parallel interface is appropriate. This chapter uses the ’542 as a reference platform for HPI bootloader option platform and ’541 for other bootloader options available on ’54x.

Topic **Page**

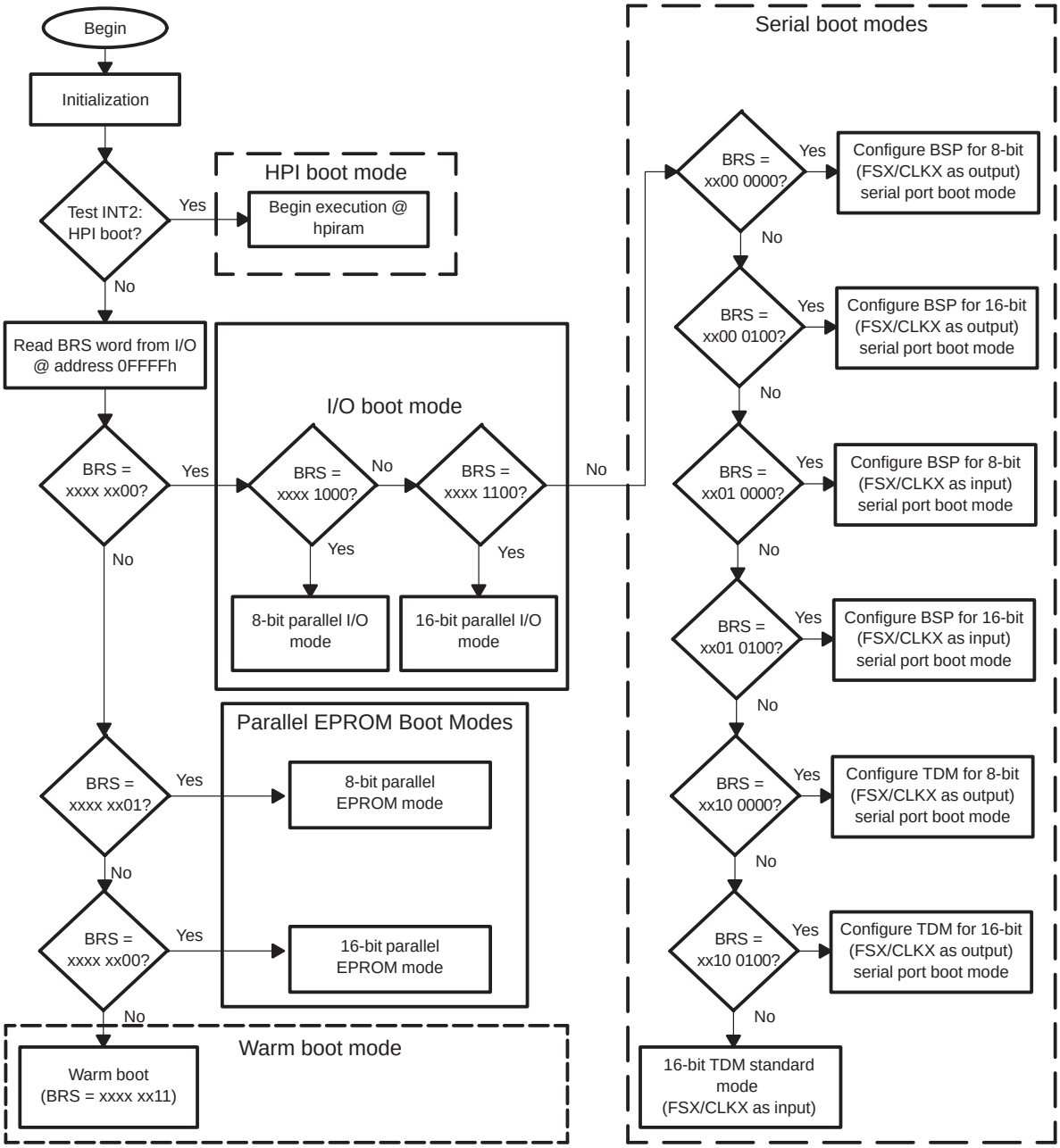
8.1 Boot Mode Selection

Execution begins at location FF80h of the on-chip ROM if the MP/$\overline{MC}$ pin of the ’54x is sampled low during a hardware reset. This location contains a branch instruction to start the bootloader program that is factory-programmed in ROM. This program sets up the CPU status registers before initiating the bootload. Interrupts are globally disabled (INTM = 1) and internal dual-access and single-access RAMs are mapped into program/data space (OVLY = 1). Seven wait states are initialized for all the program and data spaces. The size of the external memory bank is set to 4K words. It uses one latency cycle when switching between program and data space.

The boot routine reads the I/O port address 0FFFFh by driving the I/O strobe ($\overline{IS}$) signal low. The lower eight bits of the word read from I/O port address 0FFFFh specify the mode of transfer. The boot routine selection (BRS) word determines the boot mode. The BRS word uses a source (SRC) field when in parallel EPROM mode and an entry address (ADDR) field when using a warm boot (see Section 8.6, *Warm Boot* on page 8-12). The six least significant bits and the configuration of CLKX and FSX pins determine whether to use the 8- or 16-bit bootload serial boot option. The BRS word also determines the 8- or 16-bit parallel I/O mode.

The host port interface (HPI) uses interrupt 2 to bootload (INT2). If INT2 is not latched, the boot routine skips HPI boot mode (see Section 8.2, *Host Port Interface (HPI) Bootloading Sequence* on page 8-4). It reads the lower eight bits from the I/O address, 0FFFFh, to determine the boot mode. Figure 8–1 illustrates the boot mode selection process.

Figure 8–1. Boot Mode Selection Process



8.2 Host Port Interface (HPI) Boot Loading Sequence

The HPI is an 8-bit parallel port that interfaces a host processor to the ’542. The host processor and the ’542 exchange information via on-chip, shared memory. The host interrupts the CPU by writing to the HPI control register (HPIC). The CPU interrupts the host by asserting the host interrupt ($\overline{\text{HINT}}$) signal. The host can acknowledge or clear this signal. The signal determines whether the HPI boot option is selected by asserting $\overline{\text{HINT}}$ low. This signal is tied to the external interrupt $\overline{\text{INT2}}$ input pin if HPI boot mode is selected. (Instead of tying $\overline{\text{HINT}}$ to $\overline{\text{INT2}}$, you can send a valid interrupt to the $\overline{\text{INT2}}$ input pin within 30 CLOCKOUT cycles after the ’542 fetches the reset vector.)

Asserting $\overline{\text{HINT}}$ low sets the corresponding interrupt flag register (IFR) bit. The bootloader waits for 20 CLKOUT cycles after asserting $\overline{\text{HINT}}$ and reads bit 2 of IFR. If the bit is set (indicating that $\overline{\text{INT2}}$ is recognized), the bootloader transfers control to the start address of the on-chip HPI RAM (1000h in program space) and executes code from that point. If bit 2 of the IFR is not set, the boot routine skips HPI boot mode and reads BRS from the I/O address, 0FFFFh, in I/O space. The lower eight bits of this word specify the mode of transfer. The bootloader ignores the rest of the bits.

If HPI boot mode is selected, the host must download the code to on-chip HPI RAM before the HPI brings the device out of reset. The bootloader keeps HPI in shared-access mode (SMODE = 1) during the entire operation. Once $\overline{\text{HINT}}$ is asserted low by the bootloader, it stays low until a host controller clears it by writing to HPIC.

8.3 16-Bit/8-Bit Parallel Boot

The parallel boot option is used when the code is stored in EPROMs (8 or 16 bits wide). The code is transferred from data to program memory. The six most significant bits (MSBs) of the source address are specified by the SRC field of the BRS word, as shown in Figure 8–2.

Figure 8–2. 16-Bit EPROM Address Defined by SRC Field

15 – 10	9	8	7	6	5	4	3	2	1	0
SRC	0	0	0	0	0	0	0	0	0	0

Note: SRC = Source address

If 16-bit parallel mode is selected, data is read in 16-bit words from the source address and incremented by 1 after every read operation. The destination address and the length of the code are specified by the first two 16-bit words. The length is defined as:

Length = number of 16-bit words to be transferred – 1

The number of 16-bit words specified by length does not include the first two words read (destination and length parameters), starting from the source address. This is shown in Figure 8–3. The code is transferred from data memory (source address) to program memory (destination address). At least a 10-cycle delay occurs between a read from EPROM and a write to the destination address. This ensures that if the destination is external memory, like fast SRAM, there is enough time to turn off the source memory (EPROM) before the write operation is performed. After the code is transferred to program memory, the ’541 branches to the destination address. This occurs for both for 16-bit and 8-bit parallel boot options.

Figure 8–3. Data Read for a 16-Bit Parallel Boot

15 – 0
Destination $_{16}$
Length $_{16}$ = N –1
Code Word (1) $_{16}$
.
.
Code word (N) $_{16}$

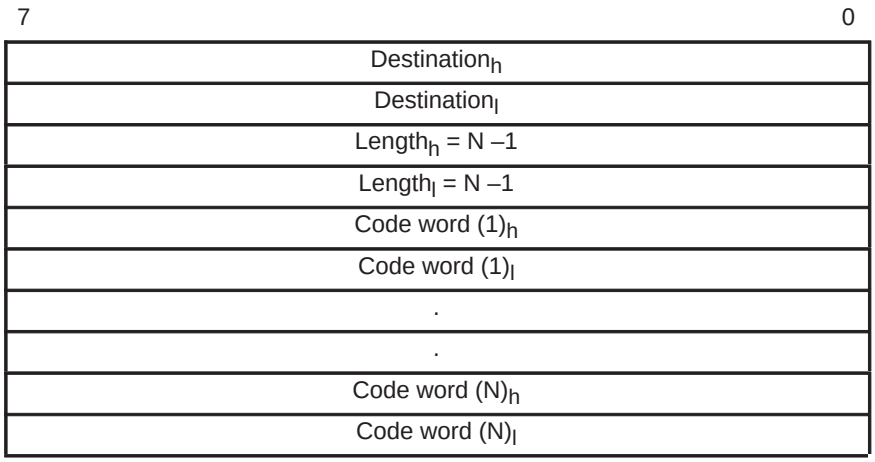
Notes: 1) Destination $_{16}$ = 16-bit destination
2) Length $_{16}$ = 16-bit word that specifies the length of the code (N) that follows it
3) Code word (N) $_{16}$ = N 16-bit words to be transferred

If the 8-bit parallel boot option is selected (see Figure 8–4), two consecutive memory locations (starting at the source address) are read to make one 16-bit word. The high-order byte must be followed by the low-order byte. Data is read from the lower eight data lines, ignoring the upper byte on the data bus. The destination address is a 16-bit word that constitutes the address in program space where the boot code is transferred. The length is defined as:

Length = number of 16-bit words to be transferred – 1

= (number of bytes to be transferred / 2) –1

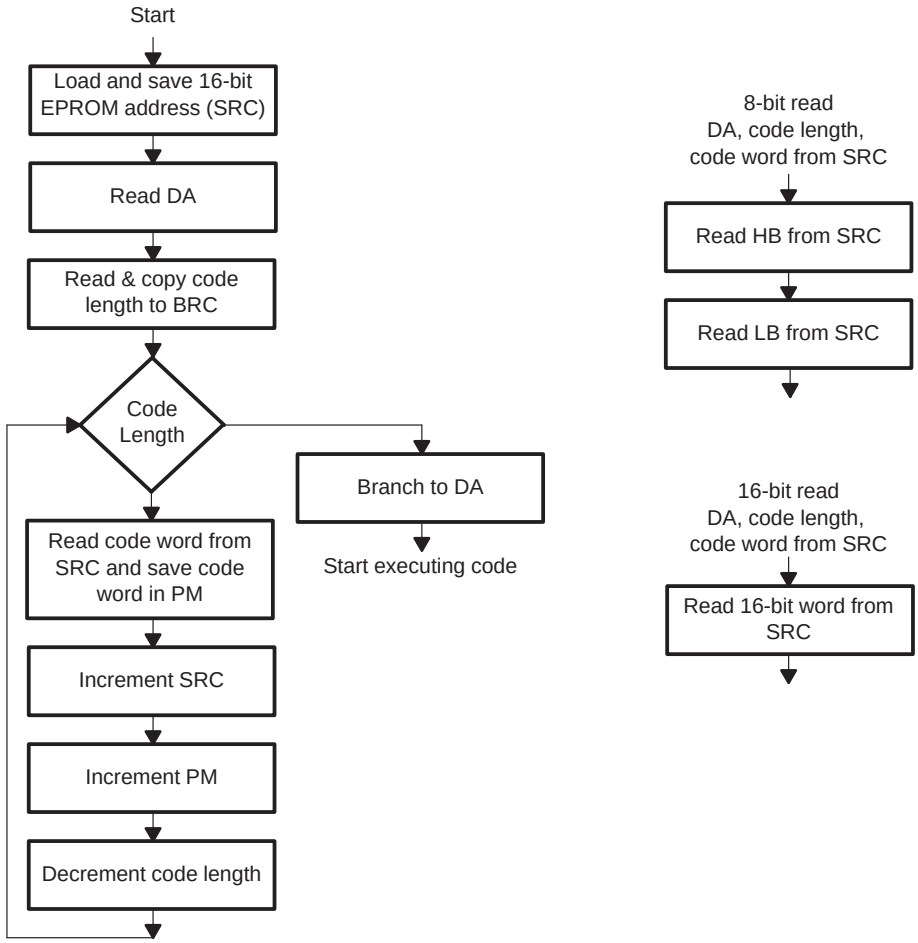
Figure 8–4. Data Read During 8-Bit Parallel Boot



Notes:
1) Destination$_h$ and Destination$_l$ represent high and low bytes of destination address
2) Length$_h$ and Length$_l$ represent high and low bytes of a 16-bit word that specifies the length N of the code that follows it.
3) N_h and N_l bytes constitute N words to be transferred.

Figure 8–5 shows the parallel boot sequence, both for the 16- and 8-bit options.

Figure 8–5. 8-Bit/16-Bit Parallel Boot



8.4 I/O Boot

The I/O boot mode provides asynchronous transfer code from I/O address 0h to internal/external program memory. Each word can be 16 or 8 bits long. The ’541 communicates with external devices using the $\overline{\text{BIO}}$ and XF handshake lines. The handshake protocol shown in Figure 8–6 is required to successfully transfer words from I/O address 0h.

Figure 8–6. Handshake Protocol

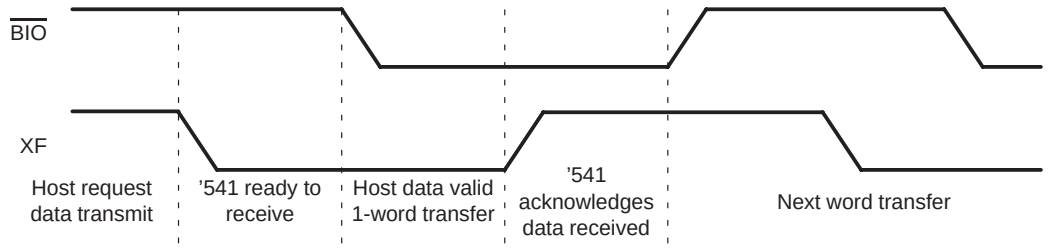
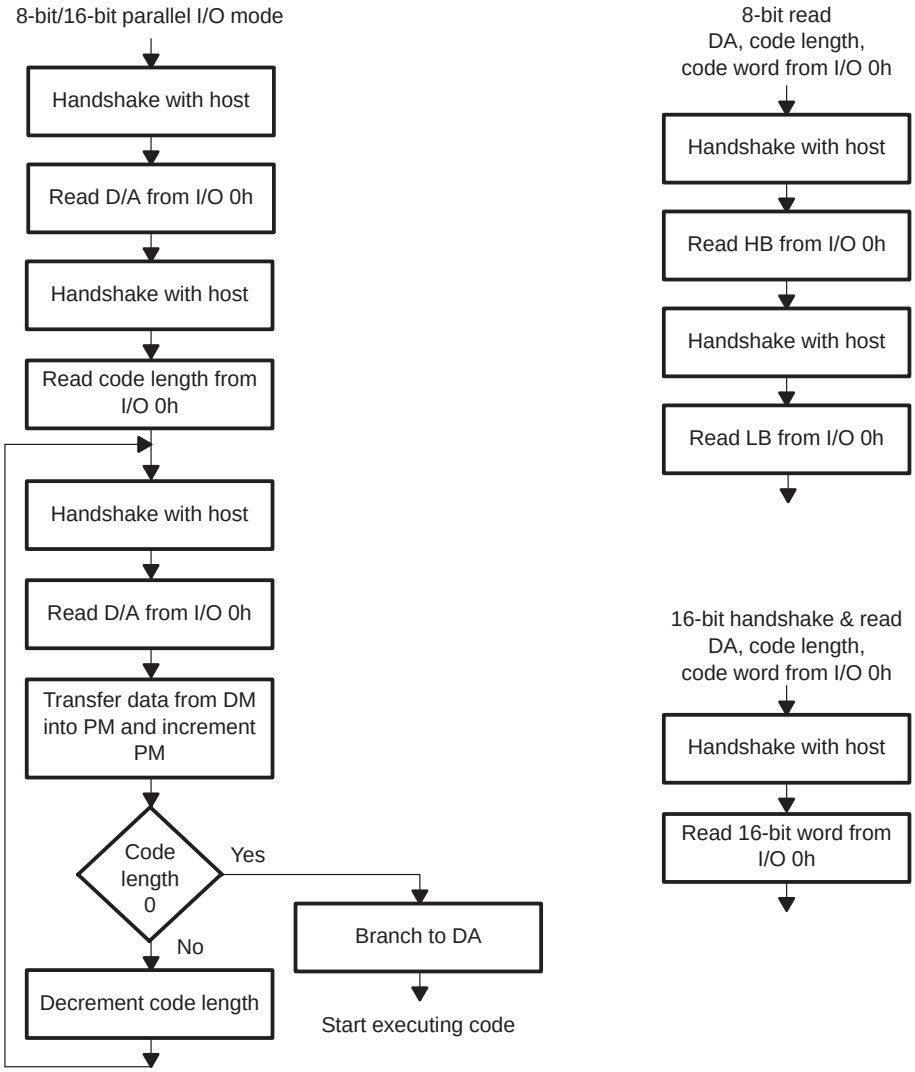


Figure 8–6 shows a data transfer initiated by the host, which drives the $\overline{\text{BIO}}$ pin low. When $\overline{\text{BIO}}$ goes low, the ’541 inputs data from I/O address 0h, drives the XF pin high to indicate to the host that the data has been received, and writes the input data to the destination address. The ’541 then waits for the $\overline{\text{BIO}}$ pin to go high before driving the XF pin low. The low status of the XF line can be polled by the host for the next data transfer.

If 8-bit transfer mode is selected, the lower eight data lines are read from I/O address, 0h. The upper byte on the data bus is ignored. The ’541 reads two 8-bit words to form a 16-bit word. The low byte of each 16-bit word must follow the high byte. Figure 8–7 shows the I/O boot sequence, both for the 16- and 8-bit options.

For both 8- and 16-bit I/O, the first two 16-bit words received by the ’541 must be the destination and length of the code, respectively. A minimum delay of 10 clock cycles occurs between the rising edge of XF and the write to the destination address. This allows the processor sufficient time to turn off its data buffers before the ’541 initiates the write operation if the destination is external memory. The ’541 accesses the external bus only when XF is high.

Figure 8–7. 8-Bit/16-Bit I/O Boot Mode



8.5 Standard Serial Boot

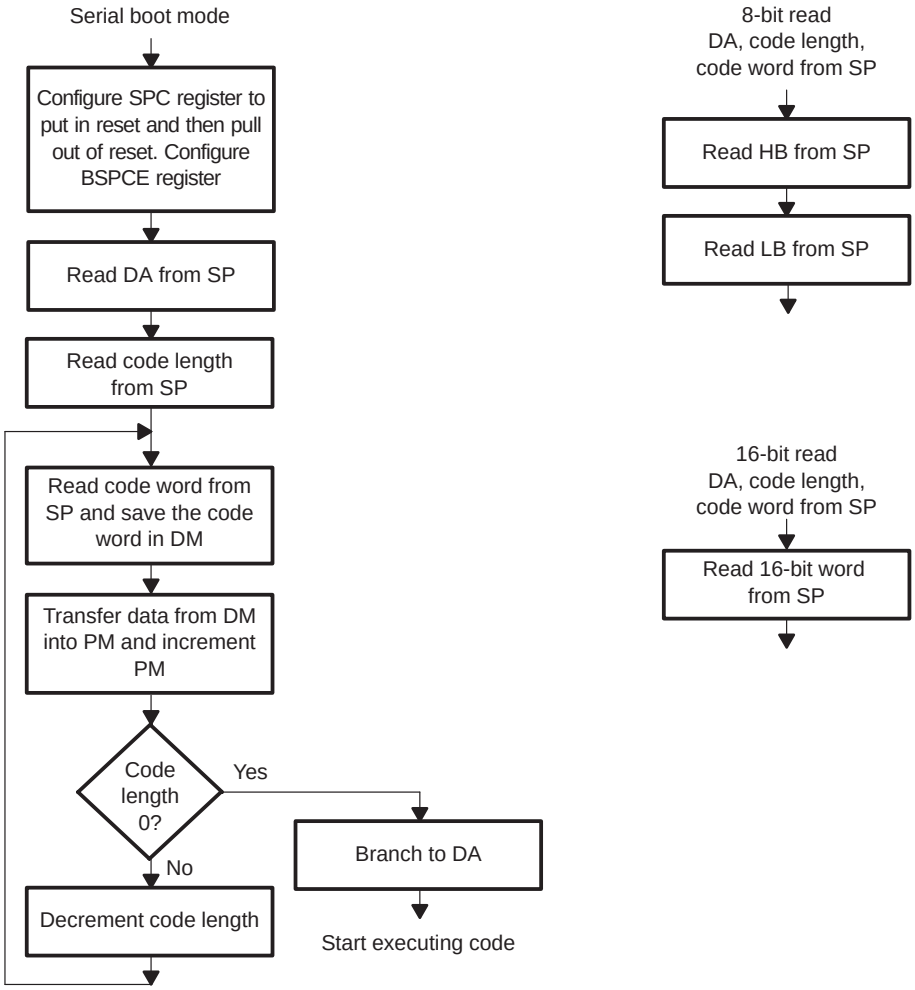
The ’541 serial boot option can use either the buffered serial port (BSP) or time-division multiplexed (TDM) serial port in standard mode during booting. Eight modes are available for the serial boot option (see Figure 8–1, *Boot Mode Selection Process*, on page 8-3). The word length (8- or 16-bit) and the configuration of the CLKX/FSX pins determines the correct mode to use. For 8-bit operation, set the following bits:

- Receive reset signal, $\overline{\text{RRST}}$ (to take serial port 0 out of reset)
- Transmit mode bit, TXM
- Clock mode bit, MCM (so that CLKX is driven by an on-chip clock source)
- Frame sync mode bit, FSM (frame sync pulses must be supplied externally on the FSR pin)
- Format bit, FO (so that data is transferred in an 8-bit length)

This translates to a value of 3Ch to the SPC register, which puts the serial port in reset. The $\overline{\text{RRST}}$ and $\overline{\text{XRST}}$ of SPC register are set to 1 to take the serial port out of reset for configuring CLKX/FSX as output pins in 8-bit mode. Then a value of FCh is written to the SPC register for 16-bit mode, FO = 0. This writes 38h to the SPC register to put the serial port in reset. It also writes a 1 to both $\overline{\text{RRST}}$ and $\overline{\text{XRST}}$ to pull the serial port register out of reset and configure CLKX/FSX as output pins, and a value of F8h is written to the SPC register.

To drive the CLKX and FSX pins as inputs, the MCM and TXM bits are disabled and the serial port is configured for 8- and 16-bit mode options. The external flag, XF, sends a signal that the ’541 is ready to respond to and receive from the serial port. The XF flag is set high at reset and is driven low to initiate reception. No frame sync pulses can appear on FSR before XF goes low. For the buffered serial port, the BSP control extension register (BSPCE) is initialized to set the CLKX to 1/4 of the CLKOUT signal. Figure 8–8 shows the serial boot sequence.

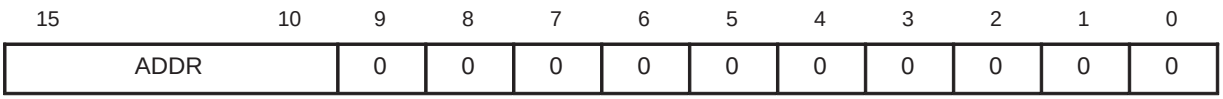
Figure 8–8. Serial Boot Mode



8.6 Warm Boot

The ’541 transfers control to the entry address if the warm boot option is specified. The warm boot option can be used after a warm device reset or if the program has already been transferred to internal or external memory by other means, such as a direct memory access (DMA). For a warm boot, the six MSBs at the entry point of the code are specified by the SRC or ADDR fields of the BRS word, as shown in Figure 8–9.

Figure 8–9. Warm Boot Address Specified in BRS Word



Note: ADDR = 6-bit page address

Example 8–1. Warm Boot Option

```
*************************************************************
*   FILENAME : BOOTC542.ASM
*
*   This code segment sets up and executes bootloader code based upon
*   data saved in data memory
*************************************************************
        .title     ″bootc542″
*********************************************************
*     symbol definitions
*********************************************************
        .mmregs
        .mnolist
        .def      boot
        .def      endboot
        .def      bootend
        .def      bsprcv_isr
        .def      tdmrcv_isr
        .def      dest
        .def      src
        .def      lngth
        .def      s8word
        .def      hbyte
        .def      state
*               .ref      boota           ; reserved for ROM Code customer
boota           .set      0184h           ; Arbitrary ref (for USR bootcode)
*    Conditional Assembly Flags
C542            .set      1
*
pa0             .set      0h          ; port address 0h for i/o boot load
brs             .set      60h         ; boot routine select (configuration word)
bootmode        .set      61h         ; boot mode extracted from brs
```

Example 8–1. Warm Boot Option (Continued)

```
s8word          .set    62h       ; concatenator for 8–bit serial load
hbyte           .set    63h       ; high byte of 8–bit serial word
p8word          .set    64h       ; concatenator for 8–bit memory load
src             .set    65h       ; source address
dest            .set    66h       ; destination address (dmov from above)
lngth           .set    67h       ; code length
temp            .set    68h       ; temporary register
state           .set    69h       ; serial I/O state vector
nmintv          .set    6ah       ; non–maskable interrupt vector
*         Bit equates
b0              .set    00h
b4              .set    04h
b8              .set    08h
bc              .set    0ch
b10             .set    010h
b14             .set    014h
b20             .set    020h
b24             .set    024h
b30             .set    030h
b34             .set    034h
hpiram          .set    01000h
int2msk         .set    004h      ; INT2_ bit position on IFR
*********************************************************
*     main program starts here
*********************************************************
       .sect     ”bootload”
boot
   ssbx   intm                    ; disable all interrupts
   ld     #0, dp
   stm    #boota, @nmintv
   orm    #03b00h, @st1           ; xf=1, mh=1, intm=1, ovm=1, sxm=1
   orm    #020h, @pmst            ; ovly=1
   stm    #07fffh, swwsr          ; 7 wait states for P_,D_, and I_ spaces
   stm    #0f800h, bscr           ; full bank switching
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*    HPI boot, simply branch to HPI RAM
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   .if    C542
   stm    #01010b, hpic           ; Send HINT_ low
   rpt    #18                     ; wait 20 clockout cycles
   nop
   bitf   @ifr, #int2msk          ; Check if INT2_ flag is set
                                  ; ThisTEST MUST BE >= 30 cycles from boot?
   stm    #int2msk, ifr           ; Clear INT2_ bit in ifr if INT2_ latched
   bcd    endboot, tc             ; If yes, branch to HPI RAM
   st     #hpiram, @dest
   .endif
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*    Read Configuration Byte
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   portr  #0ffffh, @brs           ; brs <–– boot load configuration word
   ld     @brs, 8, a              ; get boot value in acc AL
   and    #0fc00h, a              ; throw away 2 LSBs
```

Example 8–1. Warm Boot Option (Continued)

```
   stl    a, @src                ; save as source address
   ldu    @brs, a                ; determine bootload method
   and    #3, a                  ; if 2 LSBs == 00
   bc     ser_io, aeq            ;  use serial or parallel I/O
   sub    #2, a                  ; if 2 LSBs == 01
   bc     par08, alt             ;  load from 8–bit memory
                                 ; if 2 LSBs == 10
   bc     par16, aeq             ;  load from 16–bit memory
                                 ; else 2 LSBs == 11
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*       Warm–boot, simply branch to source address      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
warmboot
   delay  @src                   ; dest <–– src
endboot
   ld     @dest, a               ; branch to destination address
   bacc   a
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*       Bootload from 16–bit memory             *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
par16
   mvdk   @src, ar1              ; ar1 points at source memory (Data)
   ld     *ar1+, a               ; load accumulator A with destination
   stl    a, @dest               ; save to scratchpad
   stlm   a, ar2                 ; current destination in block repeat
   ld     *ar1+, a               ; get the length
   stlm   a, brc                 ; update block repeat counter register
   nop                           ; brc latency
   rptb   xfr16–1
   mvdk   *ar1+, ar3             ; read object data
   ldm    ar2, a                 ; get previous destination address
   add    #1, a                  ; these instructions also
   stlm   a, ar2                 ; serve the purpose of inserting
   sub    #1, a                  ; 10 cycles b/w read & write
   writa  @ar3                   ; write object data to destination
xfr16
   b      endboot
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*       Bootload from 8–bit memory, MS byte first       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
par08
   mvdk   @src, ar1              ; ar1 points at source memory (Data)
   ld     *ar1+, 8, a            ; load accumulator A with destination
   mvdk   *ar1+, ar3             ; ar3   <–– junkbyte.low byte
   andm   #0ffh, @ar3            ; ar3   <–– low byte
   or     @ar3, a                ; acc A <–– high byte.low byte
   stl    a, @dest               ; save to scratchpad  for endboot
   stlm   a, ar2                 ; ar2 points at destination
   ld     *ar1+, 8, a            ; get number of 16–bit words
   mvdk   *ar1+, ar3             ; ar3   <–– junkbyte.low byte
   andm   #0ffh, @ar3            ; ar3   <–– low byte
   or     @ar3, a                ; acc A <–– high byte.low byte
```

Example 8–1. Warm Boot Option (Continued)

```
   stlm   a, brc              ; update block repeat counter register
   nop                        ; brc update latency
   rptb   eloop4–1
   ld     *ar1+, 8, a         ; acc A <–– high byte
   mvdk   *ar1+, ar3          ; ar3   <–– junkbyte.low byte
   andm   #0ffh, @ar3         ; ar3   <–– low byte
   or     @ar3, a             ; acc A <–– high byte.low byte
   stl    a, @p8word
   ldm    ar2, a              ; acc A <–– destination address
   nop                        ; 10 cycles b/w read & write
   writa  @p8word             ; write object data to destination
   add    #1, a
   stlm   a, ar2              ; update destination address
eloop4
       b      endboot
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
ser_io
   ld     @brs, a
   and    #0fh, a             ; clear except lower 4 bits
   stl    a, @bootmode        ; save only boot mode
   cmpm   @bootmode, #b8      ; test for io boot 8
   bc     pasync08, tc        ; if set, perform parallel I/O bootload–8
   cmpm   @bootmode, #bc      ; test bit #3 of configuration word
   bc     pasync16, tc        ; if set, perform parallel I/O bootload–16
* * * * * * * * * * * * * * * * * * * * * * *
*       Bootload from serial port       *
* * * * * * * * * * * * * * * * * * * * * * *
ser
   ld     @brs, a
   and    #3fh, a             ; clear except lower 6 bits
   stl    a, @bootmode        ; save only boot mode
   cmpm   @bootmode, #b0      ; test bit #0 of bootmode word
   bcd    bsp08int, tc        ; if set, then 8–bit serial with int BCLKX, BFSX
   andm   #0ff01h, @spc       ; clear bits 1–7
   cmpm   @bootmode, #b4      ; test bit #2 of bootmode word
   bc     bsp16int, tc        ; if set, then 16–bit serial with int BCLKX, BFSX
   cmpm   @bootmode, #b10     ; test bit #4 of bootmode word
   bc     bsp08ext, tc        ; if set, then 8–bit serial with ext BCLKX, BFSX
   cmpm   @bootmode, #b14        ; test bit #4&2 of bootmode word
   bc     bsp16ext, tc        ; if set, then 16–bit serial with ext BCLKX, BFSX
   cmpm   @bootmode, #b20     ; test bit #5 of bootmode word
   bcd    tdm08int, tc        ; if set, then 8–bit serial with int TCLKX, TFSX
   andm   #0ff00h, @tspc      ; clear bits 0–7
   cmpm   @bootmode, #b24     ; test bit #5&2 of bootmode word
   bc     tdm16int, tc        ; if set, then 16–bit serial with int TCLKX, TFSX
   cmpm   @bootmode, #b30     ; test bit #5&4 of bootmode word
   bc     tdm08ext, tc        ; if set, then 8–bit serial with ext TCLKX, TFSX
   cmpm   @bootmode, #b34     ; test bit #5&4&2 of bootmode word
   bc     tdm16ext, tc        ; if set, then 16–bit serial with ext TCLKX, TFSX
   b      bootend
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*        Bootload from Buffered Serial Port (BSP)       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Example 8–1. Warm Boot Option (Continued)

```
bsp16ext
   orm    #0008h, @spc        ; configure sport and put in reset
   stm    #0003h, spce        ;CLKKV=3,FSP=CLKP=FE=FIG=PCM=BXE=HLTX=BRE=HLTR=0
   bd     bspself1
   st     #bspadd16, @state   ; 16–bir service routine addr
bsp08ext
   orm    #000ch, @spc        ; configure sport and put in reset
   stm    #0003h, spce        ; CLKKV=3,FSP=CLKP=FE=FIG=PCM=BXE=HLTX=BRE=HLTR=0
   bd     bspself1
   st     #bspadd8_1, @state  ; 8–bit service routine addr
bsp16int
   orm    #0038h, @spc        ; configure sport and put in reset
   stm    #0003h, spce        ; CLKKV=3,FSP=CLKP=FE=FIG=PCM=BXE=HLTX=BRE=HLTR=0
   bd     bspself1
   s      #bspadd16, @state   ; 16–bit service routine addr
bsp08int
   orm    #003ch, @spc        ; configure sport and put in reset
   stm    #0003h, spce        ; CLKKV=3,FSP=CLKP=FE=FIG=PCM=BXE=HLTX=BRE=HLTR=0
   st     #bspadd8_1, @state  ; 8–bit service routine addr
bspself1
   orm    #0080h, @spc        ; take sport out of reset
   rsbx   xf                  ; signal ready–to–receive
*  Poll for receive data ready
bspin
   rsbx   tc                  ; clear flag
bspinn
   bcd    bspinn, ntc         ; begin receive data routine
   bitf   @spc, #0400h        ; if rrdy = 1
bsprcv_isr
   ld     @state, a           ; vector to the sport receive routine
   bacc   a
*  Load destination address (16–bit serial mode)
bspadd16
   mvdk   @drr, ar1           ; get destination addr into ar1
   mvkd   ar1, @dest          ; save destination addr
   bd     bspin
   st     #bsplen16, @state   ; next service routine
*  Load end address (16–bit serial mode)
bsplen16
   ldm    drr, a              ; get length in words
   add    @dest, a            ; add destination address
   stlm   a, ar0              ; save end address
   bd     bspin
   st     #bspisr16, @state    ; next service routine
*  Bootload 16–bit serial data
bspisr16
   ldu    @ar1, a             ; get the destination address
   cmpr   eq, ar1             ; check for RAM full condition
   bcd    bspin, ntc          ; spin if transfer not complete
   writa  @drr                ; write object word at destination addr
   mar    *ar1+               ; increment destination addr
   b      endboot
```

Example 8–1. Warm Boot Option (Continued)

```
*  Load destination address (8–bit serial mode)
bspadd8_1
   ld    @drr, 8, a          ; acc A <–– junkbyte.high byte
   and   #0ff00h, a          ; acc A <–– high.byte
   stl   a, @hbyte           ; save high byte
   bd    bspin
   st    #bspadd8_2, @state  ; next service routine
bspadd8_2
   ldu   @drr, a             ; acc A <–– junkbyte.low byte
   and   #0ffh, a            ; acc A <–– low byte
   or    @hbyte, a           ; acc A <–– high byte.low byte
   stlm  a, ar1              ; save destination address
   stl   a, @dest
   bd    bspin
   st    #bsplen8_1, @state   ; next service routine
*  Load end address (8–bit serial mode)
bsplen8_1
   ld    @drr, 8, a          ; acc A <–– high byte
   stl   a, @hbyte           ; save high byte
   bd    bspin
   st    #bsplen8_2, @state  ; next service routine
bsplen8_2
   ldu   @drr, a             ; acc A <–– junkbyte.low byte
   and   #0ffh, a            ; acc A <–– low byte
   or    @hbyte, a           ; acc A <–– high byte.low byte
   add   @dest, a            ; add destination address
   stlm  a, ar0              ; save end address
   bd bspin
   st    #bspisr8_1, @state  ; next service routine
*  Bootload 8–bit serial data
bspisr8_1
   ld    @drr, 8, a          ; acc A <–– high byte
   stl   a, @hbyte           ; save high byte
   bd    bspin
   st    #bspisr8_2, @state  ; next service routine
bspisr8_2
   ldu   @drr, a             ; acc A <–– junkbyte.low byte
   and   #0ffh, a            ; acc A <–– low byte
   or    @hbyte, a           ; acc A <–– high byte.low byte
   stl   a, @s8word          ; save 16–bit word
   ldm   ar1, a              ; get destination addr
   cmpr  eq, ar1             ; check for RAM full condition
   bcd   endboot, tc         ; exit if RAM full
   writa @s8word             ; copy word to pmem[<ar1>]
   mar   *ar1+
   bd    bspin
   st    #bspisr8_1, @state  ; next service routine
```

Example 8–1. Warm Boot Option (Continued)

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  Bootload from TDM Serial Port (TDM)                 *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
tdm16ext
   orm    #0008h, @tspc       ; configure sport and put in reset
   bd     tdmself1
   st     #tdmadd16, @state   ; 16–bir service routine addr
tdm08ext
   orm    #000ch, @tspc       ; configure sport and put in reset
   bd     tdmself1
   st     #tdmadd8_1, @state  ; 8–bit service routine addr
tdm16int
   orm    #0038h, @tspc       ; configure sport and put in reset
   bd     tdmself1
   st     #tdmadd16, @state   ; 16–bit service routine addr
tdm08int
   orm    #003ch, @tspc       ; configure sport and put in reset
   st     #tdmadd8_1, @state  ; 8–bit service routine addr
tdmself1
   orm    #0080h, @tspc       ; take sport out of reset
   rsbx   xf                  ; signal ready–to–receive
*  Poll for receive data ready
tdmspin
   rsbx   tc                  ; clear flag
tdmspinn
   bcd    tdmspinn, ntc       ; begin receive data routine
   bitf   @tspc, #0400h       ; if rrdy = 1
tdmrcv_isr
   ld     @state, a           ; vector to the sport receive routine
   bacc   a
*  Load destination address (16–bit serial mode)
tdmadd16
   mvdk   @trcv, ar1          ; get destination addr into ar1
   mvkd   ar1, @dest          ; save destination addr
   bd     tdmspin
   st     #tdmlen16, @state   ; next service routine
*  Load end address (16–bit serial mode)
tdmlen16
   ldm    trcv, a             ; get length in words
   add    @dest, a            ; add destination address
   stlm   a, ar0              ; save end address
   bd     tdmspin
   st     #tdmisr16, @state   ; next service routine
*  Bootload 16–bit serial data
tdmisr16
   ldu    @ar1, a             ; get the destination address
   cmpr   eq, ar1             ; check for RAM full condition
   bcd    tdmspin, ntc        ; spin if transfer not complete
   writa  @trcv               ; write object word at destination addr
   mar    *ar1+               ; increment destination addr
   b      endboot
```

Example 8–1. Warm Boot Option (Continued)

```
*  Load destination address (8–bit serial mode)
tdmadd8_1
   ld    @trcv, 8, a         ; acc A <–– junkbyte.high byte
   and   #0ff00h, a          ; acc A <–– high.byte
   stl   a, @hbyte           ; save high byte
   bd    tdmspin
   st    #tdmadd8_2, @state  ; next service routine
tdmadd8_2
   ldu   @trcv, a            ; acc A <–– junkbyte.low byte
   and   #0ffh, a            ; acc A <–– low byte
   or    @hbyte, a           ; acc A <–– high byte.low byte
   stlm  a, ar1              ; save destination address
   stl   a, @dest  bd tdmspin
   st    #tdmlen8_1, @state  ; next service routine
*  Load end address (8–bit serial mode)
tdmlen8_1
   ld    @trcv, 8, a         ; acc A <–– high byte
   stl   a, @hbyte           ; save high byte
   bd    tdmspin
   st    #tdmlen8_2, @state  ; next service routine
tdmlen8_2
   ldu   @trcv, a            ; acc A <–– junkbyte.low byte
   and   #0ffh, a            ; acc A <–– low byte
   or    @hbyte, ad          ; acc A <–– high byte.low byte
   add   @dest, a            ; add destination address
   stlm  a, ar0              ; save end address
   bd    tdmspin
   st    #tdmisr8_1, @state  ; next service routine* Bootload 8–bit serial data
tdmisr8_1
   ld    @trcv, 8, a         ;acc A <–– high byte
   stl   a, @hbyte           ; save high byte
   bd    tdmspin
   st    #tdmisr8_2, @state  ; next service routine
tdmisr8_2
   ldu   @trcv, a            ; acc A <–– junkbyte.low byte
   and   #0ffh, a            ; acc A <–– low byte
   or    @hbyte, a           ; acc A <–– high byte.low byte
   stl   a, @s8word          ; save 16–bit word
   ldm   ar1, a              ; get destination addr
   cmpr  eq, ar1             ; check for RAM full condition
   bcd   endboot, tc         ; exit if RAM full
   writa @s8word             ; copy word to pmem[<ar1>]
   mar   *ar1+
   bd    tdmspin
   st    #tdmisr8_1, @state  ; next service routine
```

Example 8–1. Warm Boot Option (Continued)

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  Bootload from parallel I/O port (pa0)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  Bootload from I/O port (16–bit parallel)
pasync16
   call   handshake
   portr  pa0, @dest          ; read word from port to destination
   call   handshake
   portr  pa0, @lngth         ; read word from port to length
   mvdk   @lngth, ar1         ; ar1   <–– code length
   ldu    @dest, a            ; acc A <–– destination address
loop16
   call   handshake
   portr  pa0, @temp          ; read word from port to temp
   ssbx   xf                  ; acknowledge word as soon as it’s read
   rpt    #8
   nop                        ; 10 cycles delay between xf and write
   writa  @temp               ; write word to destination
   add    #1, a               ; increment destination address
   banz   loop16, *ar1–       ; loop if ar1 is not zero
   ldu    @dest, a            ; branch to destination addess
   bacc   a
*  Bootload from I/O port (8–bit parallel), MS byte first pasync08
*  get destination address from 1st two byte
   call   handshake
   portr  pa0, @hbyte
   ld     @hbyte, 8, a        ; read high byte from port
   stl    a, @hbyte           ; save high byte
   call   handshake
   portr  pa0, @dest
   ldu    @dest, a            ; read low byte from port
   and    #0ffh, a            ; clear upper byte
   or     @hbyte, a           ; combine high and low byte
   stl    a, @dest            ; save destination address
*  get code length from 2nd two byte
   call   handshake
   portr  pa0, @hbyte
   ld     @hbyte, 8, a        ; read high byte from port
   stl    a, @hbyte           ; save high byte
   call   handshake
   portr  pa0, @lngth
   ldu    @lngth, a           ; read low byte from port
   and    #0ffh, a            ; clear upper byte
   or     @hbyte, a           ; combine high and low byte
   stl    a, @lngth           ; save code length
   stlm   a, ar1              ; ar1   <–– code length
   ldu    @dest, a
   ld     a, b                ; acc B <–– destination address
```

Example 8–1. Warm Boot Option (Continued)

```
loop08
   call   handshake
   portr  pa0, @hbyte
   ld     @hbyte, 8, a           ; read high byte from port
   stl    a, @hbyte              ; save high byte
   call   handshake
   portr  pa0, @temp
   ssbx   xf                     ; acknowledge byte as soon as it's read
   ldu    @temp, a               ; read low byte from port
   and    #0ffh, a               ; clear upper byte
   or     @hbyte, a              ; combine high and low byte
   stl    a, @temp               ; save code word
   ld     b, a                   ; acc A <-- destination address
   nop
   nop                           ; 10 cycles delay between xf and write
   writa  @temp                  ; write code word to program memory
   add    #1, a                  ; increment destination address
   ld     a, b                   ; save new destination address
   banz   loop08, *ar1–          ; loop if ar1 not zero
   ldu    @dest, a               ; branch to destination address
   bacc   a
*  Handshake with BIO signal using XF
handshake
   ssbx   xf                     ; acknowledge previous data word
biohigh
   bc     biohigh,bio            ; wait till host sends request
   rsbx   xf                     ; indicate ready to receive new data
biolow
   rc     bio                    ; wait till new data ready
   b      biolow
bootend
   .end
```

Chapter 9

Host–Target Communication

This chapter describes the communication interface between the ’54x EVM and its host. The system passes data between the target and host, while maintaining real-time operation. The system can be driven by interrupts, polled, or a mixture of the two.

Topic **Page**

9.1 Communication Channels

The host communicates to the ’54x EVM via 38 16-bit I/O locations. Each I/O location is defined by an offset to an I/O page 0 address, shown in Table 9–1. There are two independent communication channels, A and B, through which the host and target can communicate. The status/control registers on both the host and target provide system-level control and status information.

Table 9–1. ’54x EVM Host-Interface Register Usage

I/O Offset From Base Address	Register	Size	Register Type
0x0800	Channel A	16	Read/write
0x0804	Channel B	16	Read/write
0x0808	Status/control	16	Read/write

The host writes to channel A (offset 0x0800) and overwrites the current value. An interrupt 1 ($\overline{\text{INT1}}$) signal is generated to the target, which sets the channel A transmit status bit, AXST, to 1 in the host control register (HCR) and sets the channel A receive status bit, ARST, to 1 in the target control register (TCR). The host reads from the same location, which clears the ARST bit in HCR and clears AXST in TCR. Channel B is a 64-word deep, bidirectional FIFO register that transfers both data and commands. Host write to 0x0804 is buffered by the FIFO. Data is ignored if the FIFO is full. Figure 9–1 shows the HCR and Table 9–2 describes the bits.

Figure 9–1. Host Control Register (HCR) Diagram

15	14	13	12	11	10	9–8	7	6	5	4	3–2	1	0
RESET	MP/MC	BTIE	ATIE	HBIO	BRST2	BRST	REV1	REV0	XF	BIO	BXST	ARST	AXST

Table 9–2. Host Control Register (HCR) Bit Summary

<table>
<tr><th>Bit</th><th>Name</th><th>Description</th></tr>
<tr><td>15</td><td>RESET</td><td>Software reset. If RESET = 1, the target processor and emulation logic are reset but the host/target communication flags are not reset.</td></tr>
<tr><td>14</td><td>MP/MC</td><td>Microprocessor/microcomputer mode select. The EVM powers up in microcomputer mode.</td></tr>
<tr><td>13</td><td>BTIE</td><td>Channel B target interrupt enable. If BTIE = 1, channel A receive conditions generate a host interrupt.</td></tr>
<tr><td>12</td><td>ATIE</td><td>Channel A target interrupt enable. If ATIE = 1, channel A receive conditions generate a host interrupt.</td></tr>
<tr><td>11</td><td>HBIO</td><td>Host BIO input to target processor</td></tr>
<tr><td>10</td><td>BRST2</td><td>Channel B receive status bit 2. If BRST2 =1, the target has written to channel B, forcing an interrupt. The BRST2 flag is cleared when the host reads its channel B.</td></tr>
<tr><td>9–8</td><td>BRST</td><td>Channel B receive status:
<table>
<tr><th colspan="2">BRST</th><th></th></tr>
<tr><th>Bit 9</th><th>Bit 8</th><th>Channel B Receive Status</th></tr>
<tr><td>0</td><td>0</td><td>Buffer empty</td></tr>
<tr><td>0</td><td>1</td><td>Buffer less than half full</td></tr>
<tr><td>1</td><td>0</td><td>Buffer half or more than full</td></tr>
<tr><td>1</td><td>1</td><td>Buffer full</td></tr>
</table></td></tr>
<tr><td>7</td><td>REV1</td><td>Card revision status bit 1</td></tr>
<tr><td>6</td><td>REV0</td><td>Card revision status bit 0</td></tr>
<tr><td>5</td><td>XF</td><td>External flag from target processor (status)</td></tr>
<tr><td>4</td><td>BIO</td><td>BIO input to target processor</td></tr>
</table>

Table 9–2. Host Control Register (HCR) Bit Summary (Continued)

<table>
<tr><th>Bit</th><th>Name</th><th>Description</th></tr>
<tr><td>3–2</td><td>BXST</td><td>Channel B transmit status:
<table>
<tr><th colspan="2">BXST</th><th rowspan="2">Channel B Transmit Status</th></tr>
<tr><th>Bit 3</th><th>Bit 2</th></tr>
<tr><td>0</td><td>0</td><td>Buffer empty</td></tr>
<tr><td>0</td><td>1</td><td>Buffer less than half full</td></tr>
<tr><td>1</td><td>0</td><td>Buffer half or more than full</td></tr>
<tr><td>1</td><td>1</td><td>Buffer full</td></tr>
</table></td></tr>
<tr><td>1</td><td>ARST</td><td>Channel A receive status. If ARST =1, the target has written to its channel A register. The ARST flag is cleared when the host reads channel A.</td></tr>
<tr><td>0</td><td>AXST</td><td>Channel A transmit status. If AXST = 1, host has written to its channel A register. The AXST flag is cleared when the target reads channel A.</td></tr>
</table>

The EVM supports two communication channels, configured as six I/O ports for host/target communication and 16 I/O ports for user expansion. Channel A is a single 16-bit bidirectional register mapped into two I/O port locations. Channel B is a single, bidirectional, 64-deep, FIFO buffer that is mapped into two I/O port locations. A status I/O port provides target control and general-purpose control, status, and discrete-bit I/O. Figure 9–3 shows the TCR and Table 9–3 describes the bits.

Figure 9–2. ’54x EVM Port Usage

Port Address	Name	Usage
0x0010	Channel A	Communications
0x0012	Channel B	Communications
0x0014	Status	Target status/control

Figure 9–3. Target Control Register (TCR) Diagram

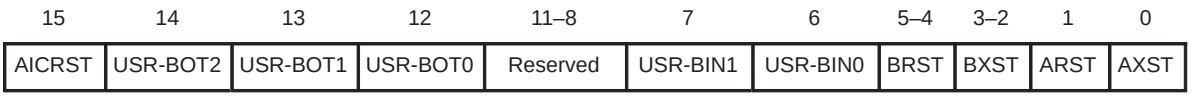


Table 9–3. Target Control Register (TCR) Bit Summary

Bit	Name	Description
15	AICRST	If AICRST = 0, the analog interface circuit is reset
14	USR-BOT2	User discrete output bit 2
13	USR-BOT1	User discrete output bit 1
12	USR-BOT0	User discrete output bit 0
11–8	–	Reserved
7	USR-BIN1	User discrete input bit 1
6	USR-BIN0	User discrete input bit 0
5–4	BRST	Channel B receive status:

BRST		
Bit 5	**Bit 4**	**Channel B Receive Status**
0	0	Buffer empty
0	1	Buffer less than half full
1	0	Buffer half or more than full
1	1	Buffer full

3–2	BXST	Channel B transmit status:

BXST		
Bit 3	**Bit 2**	**Channel B Transmit Status**
0	0	Buffer empty
0	1	Buffer less than half full
1	0	Buffer half or more than full
1	1	Buffer full

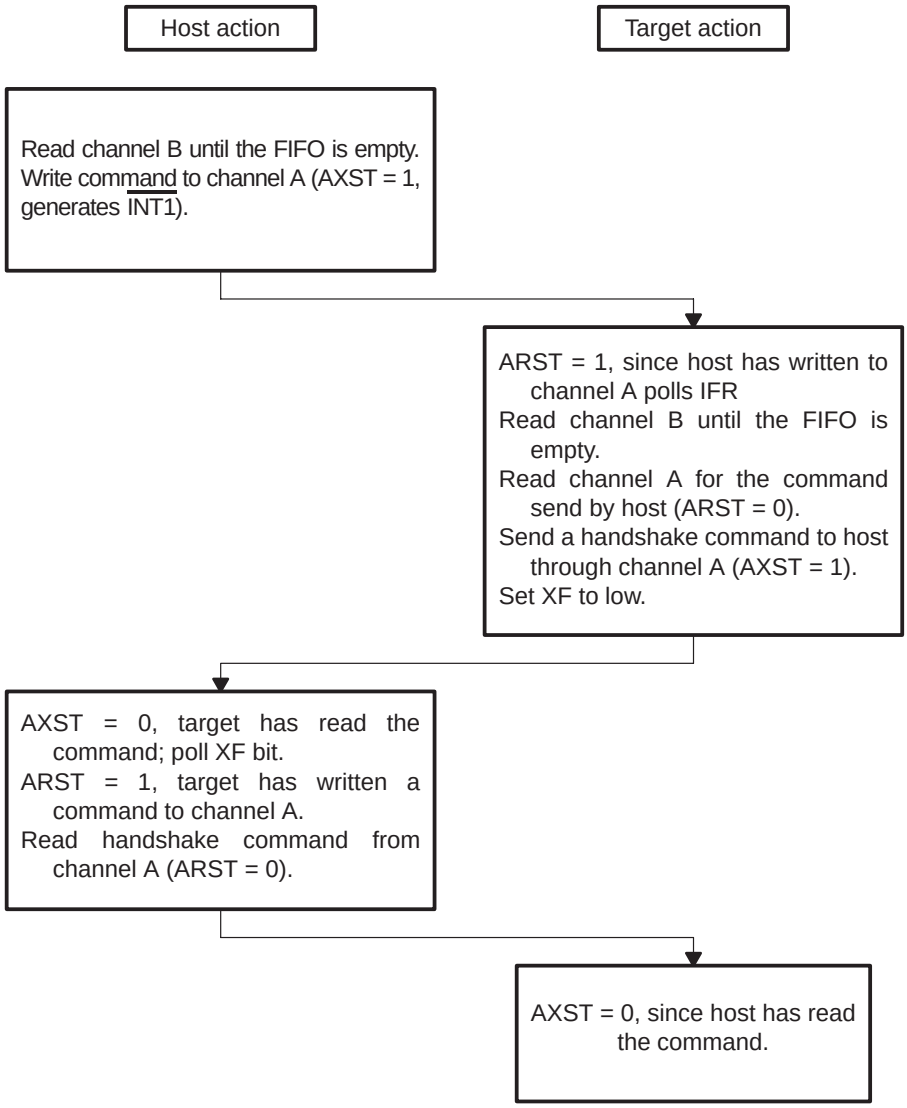
1	ARST	Channel A receive status. If ARST =1, the host has written to its channel A register. The ARST flag is cleared when the target reads channel A.
0	AXST	Channel A transmit status. If AXST = 1, target has written to its channel A register. The AXST flag is cleared when the host reads channel A.

Note: For further register and I/O information, see the *TMS320C54x Evaluation Module Technical Reference*.

9.2 Handshake and Data Transfer

Example 9–1 through Example 9–4 show how to communicate between the host and the target through both channels, A and B. The communication involves two steps: a handshake and a data transfer. Channel A sends commands between the host and the target and channel B uses the FIFO buffer to transfer data, either 64 or 32 words at a time. A buffer of 256 samples is transferred from target to the host. Data is sent 32 words at a time from the target, except for the first FIFO, where the first 64 words are sent to the host. The data is transferred from the target to the FIFO whenever an $\overline{\text{INT1}}$ signal occurs. This is generated when the host writes to channel A. The XF line on the host control sets up a handshake between the host and the target. Figure 9–4 illustrates the sequence of events in a handshake and Figure 9–5 illustrates the sequence of events in a data transfer.

Figure 9–4. Handshake Protocol



Example 9–1. Handshake — Target Action

```
*************************************************************************
* This file includes the TCR register configuration of EVM
*************************************************************************
K_AIC_RST           .set      0b << 15            ; if AICRST=0, aic is reset
K_USR_BOT           .set      000b << 12          ; User discrete output bits
                                                  ; 0,1,2
K_RESRV             .set      0000b << 8          ; Reserved bits
K_USR_BIN           .set      00b << 6            ; User discrete input bits 0,1
K_RCV_BRST          .set      00b << 4            ; Channel B receive status regs
                                                  ; buffer half or more K_XMT_BXST
                    .set      11b << 2            ; Ch B trasnmit status register
                                                  ; buffer half or more K_RCV_ARST
                    .set      0b << 1             ; Ch A receive register
K_XMT_AXST          .set      0b << 1             ; Ch A transmit register
K_TCR_HIGH          .set      K_AIC_RST|K_USR_BOT|K_RESRV
K_TCR_LOW           .set      K_USR_BIN|K_RCV_BRST|K_XMT_BXST|K_RCV_ARST|K_XMT_AXST
K_TCR               .set      K_TCR_HIGH|K_TCR_LOW
*************************************************************************
* this includes I/O address of CH_A, CH_B and different commands that's been
* passed between host and the target
*************************************************************************
K_0                 .set      0h                  ; constant 0
K_FIFO_FULL         .set      0xFF                ; Full FIFO command written by
                                                  ; target
K_FIFO_EMPTY        .set      0xEE                ; Empty FIFO command
                                                  ; written by host
K_AXST_CLEAR        .set      0xAE                ; Clear AXST empty command
                                                  ; written by the target
K_HANDSHAKE_CMD     .set      0xAB                ; handshake CMD written by host
K_CHB               .set      12h                 ; Use Channel B as I/O interface
                                                  ; to 54x EVM for sending data
K_CHA               .set      10h                 ; Use Channel A as I/O interface
                                                  ; to 54x EVM for send command
                                                  ; to host
K_TRGCR_ADDR        .set      14h                 ; Target status control register
                                                  ; I/O address location
K_AXST          .set          1h                  ; 0h
K_ARST          .set          2h                  ; used to check the control bits
K_BXST          .set          0Ch                 ; check if K_FIFO_SIZE
K_FIFO_SIZE     .set          64                  ; its a 64 FIFO
K_FRAME_SIZE    .set          256                 ; Frame size
K_HOST_FLAG     .set          1                   ; if 0, then host interface
                                                  ; is disabled
; TEXAS INSTRUMENTS INCORPORATED
          .mmregs
          .include     "target.inc"
          .include     "init_54x.inc"
          .include     "interrpt.inc"
          .ref      FIFO_DP
          .ref      d_command_reg
          .ref      d_command_value
          .def      evm_handshake
;
```

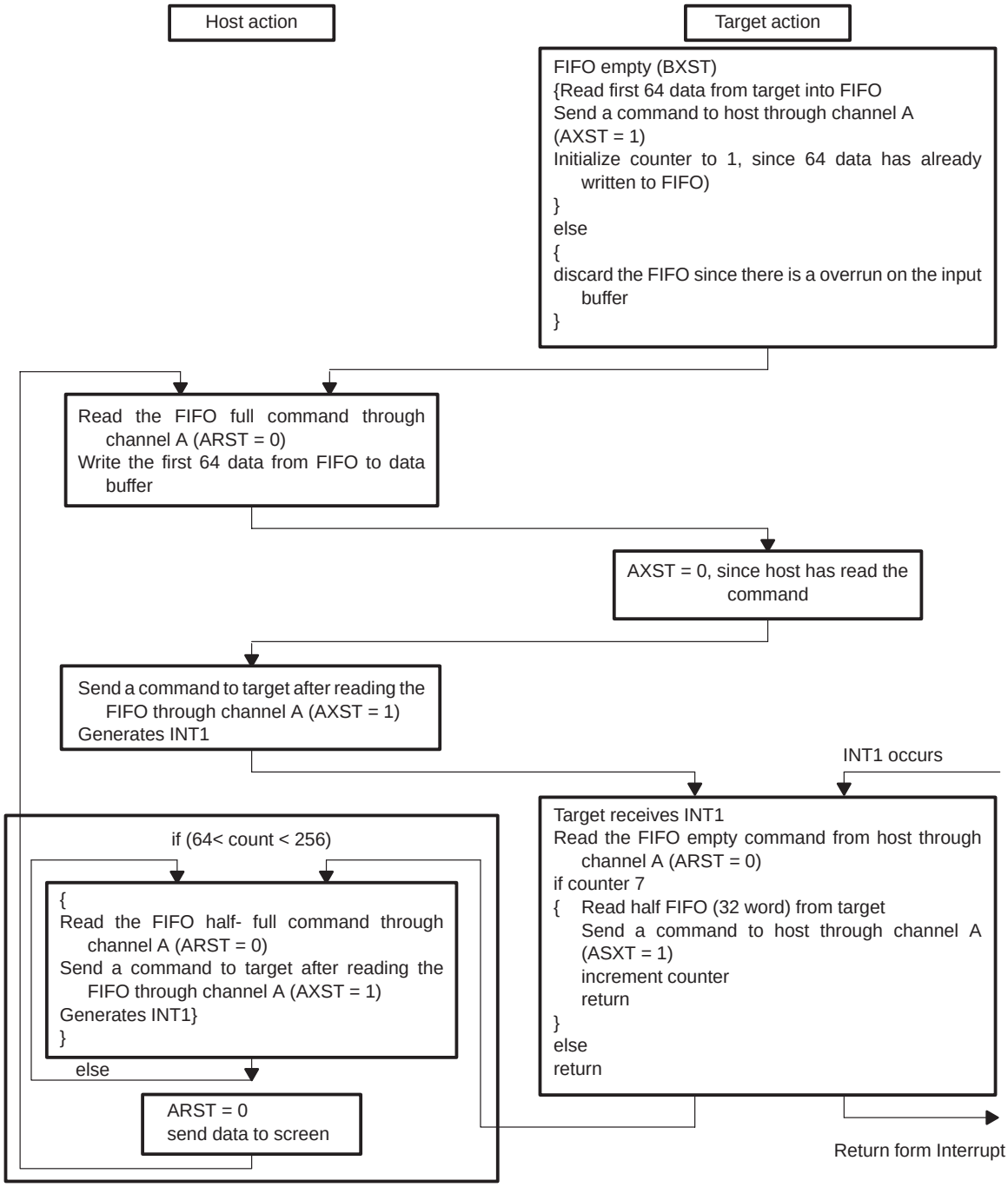
Example 9–1. Handshake — Target Action (Continued)

```
;  Functional Description
;
;   This initiates the handshake between the host(PC) and the target (DSP).
;   The host writes a command to CH A. This generates an INT1 on the target.
;   The AXST bit on HCR is set to 1.The INT1 bit in IFR is polled if it is set
;   then it is cleared to clear pending interrupts. The FIFO is cleared
;   by reading from the FIFO. The command from host is read thru CH A and ARST
;   on TCR is cleared. Another command from target is written to CH A,
;   which sets AXST. Also sets XF low. The host polls XF line.
;   The host reads CH A which clears ARST on host side and AXST on target side.
;
;-------------------------------------------------------------------------
          .sect        ”handshke”
evm_handshake:
          LD     #0,DP
          BITF   IFR,02h                       ; Poll for INT1
          BC     evm_handshake,NTC             ; ARST = 1
          STM    #K_INT1,IFR                   ; clear the pending interrupt
          LD     #FIFO_DP,DP
          RPT    #K_FIFO_SIZE–1
          PORTR  K_CHB,d_command_reg           ; assures that FIFO is empty to
          PORTR  K_CHA,d_command_value         ; ARST = 0
target_handshake_command:                      ; read the command from
                                               ; to acknowledge INT1
          PORTR  K_TRGCR_ADDR,d_command_reg    ; while (port14 & ARST)
          BITF   d_command_reg,K_ARST          ; check FIFO empty
          BC     target_handshake_command,TC   ; branch occurs
          LD     #K_HANDSHAKE_CMD,A            ; indicate of FIFO empty
          SUB    d_command_value,A
bad_handshake_command
          BC     bad_handshake_command,ANEQ    ; read the command send by hosts
          ST     #K_AXST_CLEAR,d_command_reg   ; send to a command to clear AXST
          PORTW  d_command_reg, K_CHA          ; write command to command reg A
                                               ; AXST = 1
          RSBX   XF                            ; XF = 0
          RET
          .end
```

Example 9–2. Handshake — Host Action

```
/*_____________________________________________________________________________
This function initializes the data buffer and reads the FIFO so that FIFO
 is empty when the real data transfers starts
-----------------------------------------------------------------------------*/
void initialize_slave(void)
{
          int j;
          for (j=0;j < 64; j++)
          dataa[j] = inport(BDAT_REG);           /* read data from data reg.    */
          for (j=0;j <256; j++)
          dataa[j] = 0;
          outport(CONT_REG, inport(CONT_REG) & 0xf7ff);
       }
/*-----------------------------------------------------------------------------
This initiates the handshake between the target and host. The host writes a command
to target which sets the AXST flag to 1. The INT1 is generated whenever
host writes to CH A. On the target side, INT1 is polled and reads the CH A.
This clears ARST on target side. A command is written to Ch A on target after
emptying the FIFO that sets AXSt =1. Later sets XF to go low. On host XF is polled
and then reads CH A that clears ARST to 0 and AXST to 0 on the target side
*/
 int receive_clear_AXST(void)
                                /* RECEIVE COMMAND FROM EVM    */
       {
          command = 0xAB;
          outport (ADAT_REG,command);
                   while(inport(CONT_REG) & AXST);      /* write command to evm */
          while((inport(CONT_REG) & XF));
          while(!(inport(CONT_REG) & ARST));      /* wait for evm to send command*/
          reply = inport(ADAT_REG);                  /* read command into reply */
          while ((reply & 0xAE) !=0xAE);
          return(reply);                            /* return command for process'g*/
    }
/*-----------------------------------------------------------------------------*/
```

Figure 9–5. Data Transfer Protocol



Example 9–3. Data Transfer — Target Action

```
; TEXAS INSTRUMENTS INCORPORATED
              .mmregs
              .include     ”target.inc”
              .ref      d_output_addr
FIFO_DP                        .usect  ”fifo_var”,0
d_command_reg                  .usect  ”fifo_var”,1
d_command_value                .usect  ”fifo_var”,1
d_fifo_count                   .usect  ”fifo_var”,1
d_fifo_ptr                     .usect  ”fifo_var”,1
              .def      fifo_host_transfer
              .def      FIFO_DP
              .def      d_command_reg
              .def      d_command_value
              .def      d_fifo_ptr
              .def      d_fifo_count
;  Functional Description
;             This routine transfers a FIFO(64) of data to host thru CH B.
;             In the process, after transferring data from DSP to FIFO sends a command
;             to host thru CH A.The host acknowledges and sends a command to target (DSP)
;             thru CH A.
;             The host transfer can be disabled by setting the K_HOST_FLAG =0
;-------------------------------------------------------------------------------
              .asg      AR7,OUTBUF_P
              .asg      AR7,SV_RSTRE_AR7
              .sect     ”fifo_fil”
fifo_host_transfer:
              LD        #FIFO_DP,DP
              .if       K_HOST_FLAG =1
              PORTR     K_TRGCR_ADDR,d_command_reg      ; while (port14 & BXST)
              BITF      d_command_reg,K_BXST
              BC        fifo_discard,TC                 ; FIFO discard
              MVDK      d_output_addr,OUTBUF_P          ; load PING/PONG bffr address
              RPT       #K_FIFO_SIZE–1                  ; write first set of 64 data
                                                        ; to FIFO
              PORTW     *OUTBUF_P+,K_CHB                ; Fill FIFO
              ST        #K_FIFO_FULL,d_command_value
              PORTW     d_command_value, K_CHA          ; writecommand to comnd reg A
              ST        #1,d_fifo_count                 ; start counting for transfers
              MVKD      OUTBUF_P,d_fifo_ptr             ; save the fifo_ptr
fifo_discard
       .endif
       RET
       .end
```

Example 9–4. Data Transfer — Host Action

```
      command_FIFO = receive_command_FIFO_FULL();
      for (fifo_size=0; fifo_size < 64; fifo_size++)
      dataa[fifo_size+count] = inport(BDAT_REG);
      send_command_new_FIFO(command);
      for (count=64; count< 256; count++)
      {
      command_FIFO = receive_command_FIFO_FULL(); /* command from target*/
      for (fifo_size=0; fifo_size < 32; fifo_size++)
      dataa[fifo_size+count] = inport(BDAT_REG);  /* read 32 word fifo*/
      count = count+31;
      send_command_new_FIFO(command);            /* send command to target*/ }
      int receive_command_FIFO_FULL(void)
                           /* RECEIVE COMMAND FROM EVM    */
         {
   while(!(inport(CONT_REG) & ARST));         /* wait for evm to send command*/
   reply = inport(ADAT_REG);                  /* read command into reply     */
   while ((reply & 0xFF) !=0xFF);
   return(reply);                             /* return command for process'g*/
         }
/*_____________________________________________________________________________*/
/* This function sends a command to target for a new set of data from FIFO*/
void send_command_new_FIFO(command)
             unsigned int command;
             {
                      command = 0xEE;
                      outport (ADAT_REG,command);
                      while(inport(CONT_REG) & AXST);
             }
```

Chapter 10

Application Code Examples

This chapter contains complete code examples for routines that are excerpted in previous chapters of this book. These routines have been developed using a ’54x EVM platform. These programs demonstrate applications that use a host interface and run in real time. You may download them to use in developing your own applications.

Topic **Page**

10.1 Running the Applications

The host communicates to the ’54x EVM through 16-bit I/O locations. Each I/O location is defined by an offset to an I/O page 0 address. The offset used for these applications is 0x240 + 0x800 for channel A, 0x240 + 0x804 for channel B, and 0x240 + 0x808 for the target/status control register, where 0x240 is the base address of the host. Check your PC system documentation to make sure that I/O space 0x240 does not conflict with other I/O devices. If the EVM is mapped to other than space 0x240, the base addresses of the control and status registers must be modified in the file host.h.

The ’54x assembler assembles code and generates object files. The linker command file links the object files and generates a file named main.out, using common object file format (COFF). You must load main.out into the EVM debugger command window with the LOAD command and compile the host software using a Borland C compiler. This generates the file, master.exe. that contains graphic routines that display data transferred from the target to the host.

To run the target application, load main.out into the EVM debugger. To start the program on the target side, press the F5 function key. If you halt the program with the escape key or use the halt command in the debugger window, the program remains in the handshake loop waiting for the host to send a command. Press the F5 function key to continue. To run the host application, execute master.exe at the DOS prompt or a window command line. When master.exe is executed by the host, it displays the message:

```
Graphics: No error. Press any key to halt.
```

When you press this key, the graphics window opens and displays data for the task the target has initiated. The default task is an oscilloscope routine. To change to a different task, go to the debugger window, halt the program, and in the command window, type:

```
e *present_command = x
```

where x = 1, 2, 3, 4, 5 or 6, and present_command has one of the following values:

1 = oscilloscope

2 = Low-pass finite impulse response (FIR) filter using MAC instruction

3 = Low-pass infinite impulse response (IIR) filter using biquad sections

4 = Lowpass FIR filtering using FIRS instruction

5 = System identification using adaptive filtering with least mean squares (LMS) instruction

6 = 256-point real fast Fourier transform (FFT)

You can view the output of the present task in the graphics window.

To exit the host application, press F3. Communication is lost if at any time the target code is reloaded or reset in the command window while the host executable is running in the background. This means that if you attempt to reset or reload the code in the debugger window and you press F5, the computer locks up. This occurs because there is no handshake between the host and the target. To unlock, reload and run the code (press F5) on the target side. On the host side, quit the window and rerun the executable.

The adaptive filter can be tested in two steps. The initial step size d_mu = 0 in the first step. If the present task is changed at the debugger window with the command e *present_command = 5, runs with d_mu = 0. Thus, the system is not identified since the coefficients of the adaptive filter are not updated. In the second step the step size can be changed by typing e *d_mu = 0x1000 at the command window. In this case, the system is identified and the filter coefficients are adapted using the LMS algorithm. In both cases the error signal can be observed both on host and also from the output of the 'AC01.

10.2 Application Code

Table 10–1 lists programs appropriate for running on a target system and tells you where to look for them in this chapter.

Table 10–1. Target Files

Title	File Name	Page
Vector Table Initialization	vectors.asm	10-6
Main Program That Calls Different Functions	main.asm	10-16
Memory Allocation for Entire Application	memory.asm	10-10
Processor Initialization	init_54x.asm	10-22
Initialization of Variables, Pointers, and Buffers	prcs_int.asm	10-29
Initialization of Serial Port 1	init_ser.asm	10-33
’AC01 Initialization	init_aic.asm	10-38
’AC01 Register Configuration	aic_cfg.asm	10-42
Receive Interrupt Service Routine	rcv_int1.asm	10-46
Task Scheduling	task.asm	10-51
Echo the Input Signal	echo.asm	10-56
Low-Pass FIR Filtering Using MAC Instruction	fir.asm	10-59
Low-Pass Biquad IIR Filter	iir.asm	10-69
Low-Pass Symmetric FIR Filtering Using FIRS Instruction	sym_fir.asm	10-64
Adaptive Filtering Using LMS Instruction	adapt.asm	10-74
256-Point Real FFT Initialization	rfft.asm	10-84
Bit Reversal Routine	bit_rev.asm	10-87
256-Point Real FFT Routine	fft.asm	10-91
Unpack 256-Point Real FFT Output	unpack.asm	10-97
Compute the Power Spectrum of the Complex Output of the 256-Point Real FFT	power.asm	10-103

Table 10–2 lists programs appropriate for running on a host system and tells you where to look for them in this chapter.

Table 10–2. Communication Interface Files

Title	File Name	Page
Handshake Between Host and Target	hand_shk.asm	10-25
Interrupt 1 Service Routine	hst_int1.asm	10-111
Data Transfer from FIFO	fifo.asm	10-106
Main Function Call on Host Side	master.c	10-118
Function Calls on Host Side	host.c	10-116
Display the Data on the Screen	view2.c	10-123
Graphic Drivers Routine	graphic2.c	10-121

Example 10–28 on page 10-124 shows the linker command that links all object files together to produce a single executable COFF object module. This file establishes the memory configuration for the entire application, using the ’541’s memory map. Example 10–29 on page 10-127 shows the configuration of the memory map for a ’541 device used by an EVM debugger.

Example 10–1. Vector Table Initialization

```
;  TEXAS INSTRUMENTS INCORPORATED
;  DSP Data Communication System Development / ASP
;
;  Archives:   PVCS
;  Filename:   vectors.asm
;  Version:    1.0
;  Status:     draft            ( )
;               proposal         (X)
;               accepted         ( )        dd–mm–yy/?acceptor.
;
;    AUTHOR     Padma P. Mallela
;
;               Application Specific Products
;               Data Communication System Development
;               12203 SW Freeway, MS 701
;               Stafford, TX 77477;{
;               IPR statements description (can be collected).
;}
;    (C)        Copyright 1996. Texas Instruments.
;               All rights reserved.
;
;{
;               Change history:
;
;     VERSION    DATE    /         AUTHORS             COMMENT
;      1.0       July–24–96  /     P.Mallela           original created
;
;}
;{
;     1. ABSTRACT
;
;    1.1 Function Type
;        a. Core Routine
;        b. Subroutine
;
;    1.2 Functional Description
;        This file contains vector table of 541
;
;
;    1.3 Specification/Design Reference (optional)
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:   TMS320C54X ASSEMBLER
;        Version:    1.02 (PC)
;        Activation: asm500 –s vectors.asm
;
;    1.6 Notes and Special Considerations
;{
;           This code is written for 541 device. The code is tested on
;           C54x EVM
```

Example 10–1. Vector Table Initialization (Continued)

```
;}
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;           –
;    2.2 Local Compiler Flags
;           –
;    2.3 Local Constants
;           –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include   ”init_54x.inc”
         .include   ”main.inc”
;    3.2 External Data
         .ref       SYSTEM_STACK
;    3.3 Import Functions
         .ref       main_start
         .ref       receive_int1
         .ref       host_command_int1
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;           –
;    4.2 Global Static Data
;           –
;    4.3 Dynamic Data
;           –
;    4.4 Temporary Data
;           –
;    4.5 Export Functions
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;============================================================================
;    5.1 reset
;
;    5.2 Functional Description
;        This function initializes the vector table of 541 device
;----------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
;
;        Reentrancy:  No
;        Recursive:   No
;
```

Example 10–1. Vector Table Initialization (Continued)

```
;    5.4 Inputs
;
;    5.5 Outputs
;
;    5.6 Global
;
;    5.7 Special considerations for data structure
;          –
;    5.8 Entry and Exit conditions
;
;    |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN
;    |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |
;in  |0 | 0 | 1 | 0 | 0  | 0 |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU
;    |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |
;out |0 | 0 | 1 | 0 | 0  | 0 |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;              Execution time: ?cycles
;              Call rate:  not applicable for this application
;
;=============================================================================
;HeaderEnd
;    5.10 Code
                .sect        ”vectors”
reset:          BD main_start                              ;RESET vector
                STM          #SYSTEM_STACK,SP
nmi:            RETE
                NOP
                NOP
                NOP                                        ;NMI
; software interrupts
sint17                  .space  4*16
sint18                  .space  4*16
sint19                  .space  4*16
sint20                  .space  4*16
sint21                  .space  4*16
sint22                  .space  4*16
sint23                  .space  4*16
sint24                  .space  4*16
sint25                  .space  4*16
sint26                  .space  4*16
sint27                  .space  4*16
sint28                  .space  4*16
sint29                  .space  4*16
sint30                  .space  4*16
int0:       RETE
            NOP
            NOP                             ; INT0
            NOP
```

Example 10–1. Vector Table Initialization (Continued)

```
int1:       BD          host_command_int1    ; Host interrupt
            PSHM        ST0
            PSHM        ST1                  ; INT1
int2:       RETE
            NOP
            NOP
            NOP
tint:       RETE
            NOP
            NOP                               ; TIMER
            NOP
rint0:      RETE                              ; Serial Port Receive
            NOP                               ; Interrupt 0
            NOP
            NOP
xint0:      RETE                              ; Serial Port Transmit
            NOP                               ; Interrupt 0
            NOP
            NOP
rint1:      BD           receive_int1         ; Serial Port Receive
            PSHM         ST0                  ;Interrupt 1
            PSHM         ST1
xint1:      RETE                              ; Serial Port Transmit
            NOP                               ; Interrupt 1
            NOP
            NOP
int3:       RETE
            NOP
            NOP                               ;INT3
            NOP
           .end
*--------------------------------------------------------------------------------
```

Example 10–2. Memory Allocation for Entire Application

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:  PVCS
; Filename:  memory.asm
; Version:   1.0
; Status :   draft      ( )
;            proposa    (X)
;            accepted   ( ) dd–mm–yy/?acceptor.
;
; AUTHOR     Padma P. Mallela
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;           VERSION   DATE     /    AUTHORS            COMMENT
;            1.0   July–24–96 /    P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains main function
;
;
;    1.3 Specification/Design Reference (optional)
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s memory.asm
;
;    1.6 Notes and Special Considerations
;{
; This code is written for 541 device. The code is tested on C54x EVM
;
```

Example 10–2. Memory Allocation for Entire Application (Continued)

```
;}
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;           –
;    2.2 Local Compiler Flags
;           –
;    2.3 Local Constants
;           –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
             .mmregs
             .include ”defines.inc”
             .include ”main.inc”
;    3.2 External Data
;
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;           –
;    4.2 Global Static Data
;           –
;    4.3 Dynamic Data
;           –
;    4.4 Temporary Data
;           –
;    4.5 Export Functions
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 main_start
;
;    5.2 Functional Description
;        Memory configuration of the application
;-------------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
;
;           Reentrancy:            No
;           Recursive :            No
;
;    5.4 Inputs
```

Example 10–2. Memory Allocation for Entire Application (Continued)

```
;    5.5 Outputs
;
;    5.6 Global
;
;    5.7 Special considerations for data structure
;
;    5.8 Entry and Exit conditions
;
;    |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;    |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in  |U | 1 |NU |NU |NU  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;    |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;              Execution time: ?cycles
;              Call rate:    not applicable for this application
;
;===========================================================================
;HeaderEnd
;    5.10 Code
STACK          .usect   ”stack”,K_STACK_SIZE
SYSTEM_STACK   .set     K_STACK_SIZE+STACK

input_data     .usect   ”inpt_buf”,K_FRAME_SIZE*2 ; input data array
output_data    .usect   ”outdata”,K_FRAME_SIZE*2  ; output data array
; this section of variables are used in receive_int1 routine and related routines
RCV_INT1_DP    .usect   ”rcv_vars”,0
d_rcv_in_ptr   .usect   ”rcv_vars”,1              ; save/restore input bffr ptr
d_xmt_out_ptr  .usect   ”rcv_vars”,1              ; save/restore output bffr ptr
d_frame_flag   .usect   ”rcv_vars”,1
d_index_count  .usect   ”rcv_vars”,1
; System Coefficients
scoff          .sect    ”coeffh”
               .include ”impulse.h”
; RAM location for the System coefficient
hcoff          .usect   ”bufferh”, H_FILT_SIZE
wcoff          .usect   ”bufferw”, ADPT_FILT_SIZE ;
; RAM location for the input data
xh             .usect   ”bufferx”, H_FILT_SIZE    ; input data for system
xw             .usect   ”bufferp”, ADPT_FILT_SIZE ; input data for adaptive filter
; RAM location for filter outputs, residual error
; and temporary location for the new input sample.
ADAPT_DP       .usect   ”adpt_var”,0
d_primary      .usect   ”adpt_var”,1
d_output       .usect   ”adpt_var”,1
d_error        .usect   ”adpt_var”,1
d_mu           .usect   ”adpt_var”,1
d_mu_e         .usect   ”adpt_var”,1
d_new_x        .usect   ”adpt_var”,1
d_adapt_count  .usect   ”adpt_var”,1
```

Example 10–2. Memory Allocation for Entire Application (Continued)

```
                                                ; the 16 tap FIR coefficients
COFF_FIR_START .sect    ”coff_fir”                   ; filter coefficients
    .word  6Fh
    .word  0F3h
    .word  269h
    .word  50Dh
    .word  8A9h
    .word  0C99h
    .word  0FF8h
    .word  11EBh
    .word  11EBh
    .word  0FF8h
    .word  0C99h
    .word  8A9h
    .word  50Dh
    .word  269h
    .word  0F3h
    .word  6Fh
COFF_FIR_END
; circular buffers for coefficients and data buffers
fir_coff_table  .usect    ”fir_coff”, 20
d_data_buffer   .usect    ”fir_bfr”, 40
; variables used in FIR routine
FIR_DP          .usect    ”fir_vars”,0
d_filin         .usect    ”fir_vars”,1
d_filout        .usect    ”fir_vars”,1
; variables used in IIR routine
IIR_DP          .usect    ”iir_vars”,0
d_iir_d        . usect    ”iir_vars”,3*2
d_iir_y         .usect    ”iir_vars”,1
                .sect     ”iir_coff”
iir_table_start
*
* second–order section # 01
*
    .word  –26778                ;A2
    .word  29529                 ;A1/2
    .word  19381                 ;B2
    .word  –23184                ;B1
    .word  –19381                ;B0
*
* second–order section # 02
*
    .word  –30497                ;A2
    .word  31131                 ;A1/2
    .word  11363                 ;B2
    .word  –20735                ;B1
    .word  11363                 ;B0
iir_table_end
iir_coff_table .usect   ”coff_iir”,16
; symmetric FIR filter coeffs
```

Example 10–2. Memory Allocation for Entire Application (Continued)

```
FIR_COFF       .sect      ”sym_fir”              ; filter coefficients
    .word  6Fh
    .word  0F3h
    .word  269h
    .word  50Dh
    .word  8A9h
    .word  0C99h
    .word  0FF8h
    .word  11EBh
; circular buffers used in symmetric filter routine
d_datax_buffer   .usect  ”cir_bfr”,20
d_datay_buffer   .usect  ”cir_bfr1”,20
    .include ”ref_tsk.inc”
task_list        .sect   ”task_tbl”             ; calls the tasks itself
    .word  do_nothing
    .word  echo_task               ; Echo routine
    .word  fir_task                ; FIR routine
    .word  iir_task
    .word  sym_fir_task
    .word  adapt_task
    .word  rfft_task
task_init_list   .sect   ”task_int”             ; has the initialization of tasks
    .word  do_nothing
    .word  no_echo_init_task       ; there is no init in this case
    .word  fir_init
    .word  iir_init
    .word  sym_fir_init
    .word  adapt_init
    .word  do_nothing
; variables used in task handling routine
TASK_VAR_DP      .usect  ”tsk_vars”,0
present_command  .usect  ”tsk_vars”,1
last_command     .usect  ”tsk_vars”,1
d_task_addr      .usect  ”tsk_vars”,1
d_task_init_addr .usect  ”tsk_vars”,1
d_buffer_count   .usect  ”tsk_vars”,1
d_output_addr    .usect  ”tsk_vars”,1
d_input_addr     .usect  ”tsk_vars”,1
; Set start addresses of buffers
fft_data         .usect  ”fft_bffr”, 4*K_FFT_SIZE  ; fft data processing buffer
; Copy twiddle tables
                 .sect   ”sin_tbl”
sine_table       .copy   twiddle1                  ; sine table
sine             .usect  ”twid_sin”,K_FFT_SIZE
                 .sect   ”cos_tbl”
cos_table  .copy   twiddle2                        ; cosine table
cosine           .usect  ”twid_cos”,K_FFT_SIZE
; Define variables for indexing input data and twiddle tables
FFT_DP           .usect  ”fft_vars”,0
d_grps_cnt       .usect  ”fft_vars”,1              ; (# groups in current stage)–1
d_twid_idx       .usect  ”fft_vars”,1
; index of twiddle tables
d_data_idx       .usect  ”fft_vars”,1              ; index of input data table
```

Example 10–2. Memory Allocation for Entire Application (Continued)

```
;variables used for host interface
FIFO_DP          .usect  ″fifo_var″,0
d_command_reg    .usect  ″fifo_var″,1
d_command_value  .usect  ″fifo_var″,1
d_fifo_count     .usect  ″fifo_var″,1
d_fifo_ptr       .usect  ″fifo_var″,1
    .end
; Filename : defines.inc
; this include file defines all the variables, buffers and pointers used for the entire
; application
.def          STACK,SYSTEM_STACK
.def          input_data,output_data
.def          scoff,hcoff,wcoff,xh,xw
.def          ADAPT_DP,d_primary,d_output,d_error
.def          d_mu,d_mue,d_new_x,d_adapt_count
.def          fir_coff_table,d_data_buffer
.def          FIR_DP,d_filin,d_filout
.def          COFF_FIR_START,COFF_FIR_END
.def          IIR_DP,d_iir_d,iir_y
.def          iir_coff_table
.def          COFF_FIR_START,COFF_FIR_END
.def.         d_datax_buffer,d_datay_buffer
.def          FIR_COFF
.def          TASK_VAR_DP,present_command,last_command
.def          d_task_addr,d_task_init_addr,d_buffer_count,d_output_addr
.def          RCV_INTL_DP
.def          d_rcv_in_ptr,d_xmt_out_ptr
.def          d_frame_flag,d_index_count
.def          fft_data,sine, cosine
.def          FFT_DP,d_grps_cnt, d_twid_idx, d_data_idx
.def          cos_table,sine_table
.def          FIFO_DP
.def          d_command_reg
.def          d_fifo_count
.def          d_fifo_ptr
; Filename:   ref_tsk.inc
; this includes all the task scheduling table referenced labels
.ref          do_nothing,echo_task,fir_task,iir_task
.ref          sym_fir_task,fir_init,iir_init,sym_fir_init
.ref          no_echo_init_task,fir_init,iir_init,sym_fir_init
.ref          adapt_init
.def          task_init_list,task_list
```

Example 10–3. Main Program That Calls Different Functions

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    main.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR       Padma P. Mallela
;
;              Application Specific Products
;              Data Communication System Development
;              12203 SW Freeway, MS 701
;              Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /     AUTHORS           COMMENT
;          1.0     July–24–96  /    P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1  Function Type
;         a.Core Routine
;         b.Subroutine
;
;    1.2  Functional Description
;         This file contains main function
;
;
;    1.3  Specification/Design Reference (optional)
;
;    1.4  Module Test Document Reference
;         Not done
;
;    1.5  Compilation Information
;         Compiler:       TMS320C54X  ASSEMBLER
;         Version:        1.02 (PC)
;         Activation:     asm500 –s main.asm
;
;    1.6 Notes and Special Considerations
;{
; This code is written for 541 device. The code is tested on C54x EVM
```

Example 10–3. Main Program That Calls Different Functions (Continued)

```
;
;}
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include    ”init_54x.inc”
         .include    ”main.inc”
;    3.2 External Data
         .ref        d_frame_flag
         .ref        RCV_INT1_DP
;    3.3 Import Functions
         .ref        aic_init,serial_init,init_54,init_bffr_ptr_var
         .ref        task_handler,evm_handshake,fifo_host_transfer
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def        main_start
;}
;    5. SUBROUTINE CODE
;    HeaderBegin
;============================================================================
;
;----------------------------------------------------------------------------
;    5.1 main_start
;
;    5.2 Functional Description
;        This is the main function that calls other functions.
;----------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
```

Example 10–3. Main Program That Calls Different Functions (Continued)

```
;            BD      main_start
;            PSHM    ST0
;            PSHM    ST1
;
;        Reentrancy:    No
;        Recursive :    No
;
;    5.4 Inputs
;
;    5.5 Outputs
;
;    5.6 Global
;
;
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
;
;=============================================================================
;HeaderEnd
;    5.10 Code
    .sect ”main_prg”
*****************************************************************************
* The code initializes the 541 device, handshake between Target (DSP)
* and the host (PC). Zeros all buffers, variables and init. pointers
* Initializes serial port, programs AC01 registers for selecting sampling
* rate, gains etc..
 ****************************************************************************
main_start:
    CALL     init_54                    ; initialize ST0,ST1 PMST and
                                        ; other registers
    .if      K_HOST_FLAG = 1
    CALL     evm_handshake              ; EVM host handshake
    .endif
    CALL     init_bffr_ptr_var          ; init tables,vars,bffrs,ptr
    CALL     serial_init                ; initialize serial_port 1
    CALLD    aic_init                   ; Configures AC01
    LD       #0,DP
    NOP
```

Example 10–3. Main Program That Calls Different Functions (Continued)

```
*************************************************************************
* After enabling interrupts from the above, the real processing starts here.
* After collecting 256 samples from AC01 a flag(d_frame_flag is set).
* Handles the task initiated by the user and transfers the data to the
* host. Keeps the sequence forever !!!!!
*************************************************************************
start_loop
    LD       #RCV_INT1_DP,DP              ; restore the DP loop:
    BITF     d_frame_flag,1               ; if 256 samples are received
    BC       loop,NTC                     ; if not just loop back
    CALL     task_handler                 ; handles task scheduling
    CALL     fifo_host_transfer           ; EVM HOST interface
    B        loop
    .end
* Includes all the constants – main.inc
K_0                    .set    0              ; constant
K_FIR_INDEX            .set    1              ; index count
K_FIR_BFFR             .set    16             ; FIR buffer size
K_neg1                 .set    –1h            ; index count
K_BIQUAD               .set    2              ; there are 2 bi–quad sections
K_IIR_SIZE             .set    10             ; each bi–quad has 5 coeffs
K_STACK_SIZE           .set    200            ; stack size
K_FRAME_SIZE           .set    256            ; PING/PONG buffer size
K_FRAME_FLAG           .set    1              ; set after 256 collected
H_FILT_SIZE            .set    128            ; H(z) filter size
ADPT_FILT_SIZE         .set    128            ; W(z) filter size
K_mu                   .set    0h             ; initial step constant
K_HOST_FLAG            .set    1              ; Enable EVM_HOST interface
K_DEFAULT_AC01         .set    1h             ; default AC01 init
*  This include file sets the FFT size for the 'C54x Real FFT code
*  Note that the Real FFT size (i.e. the number of points in the
*  original real input sequence) is 2N; whereas the FFT size is
*  the number of complex points formed by packing the real inputs,
*  which is N.  For example, for a 256–pt Real FFT, K_FFT_SIZE
*  should be set to 128 and K_LOGN should be set to 7.
K_FFT_SIZE             .set    128            ; # of complex points (=N)
K_LOGN                 .set    7              ; # of stages (=logN/log2)
K_ZERO_BK              .set    0
K_TWID_TBL_SIZE        .set    128            ; Twiddle table size
K_DATA_IDX_1           .set    2              ; Data index for Stage 1
K_DATA_IDX_2           .set    4              ; Data index for Stage 2
K_DATA_IDX_3           .set    8              ; Data index for Stage 3
K_FLY_COUNT_3          .set    4              ; Butterfly counter for Stage 3
K_TWID_IDX_3           .set    32             ; Twiddle index for Stage 3
```

Example 10–3. Main Program That Calls Different Functions (Continued)

```
*******************************************************************************
* FILENAME: INIT54x.INC
* This include file contains all the initial values of ST0, ST1, PMST, SWWSR, BSCR
registers
* ST0 Register Organization
*  ------------------------------------------------
*  | 15    13 | 12 | 11 | 10 |  9  | 8           0  |
*  |----------|----|----|----|-----|------------------|
*  |   ARP    | TC | C  | OVA| OVB |       DP         |
*  ------------------------------------------------
*
*******************************************************************************
K_ARP         .set    000b<<13              ; ARP can be addressed from 00b –111b
                                            ; reset value
K_TC          .set    1b<<12                ; TC = 1 at reset
K_C           .set    1b<<11                ; C = 1 at reset
K_OVA         .set    1b<<10                ; OVA = 0 at reset, Set OVA
K_OVB         .set    1b<< 9                ; OVB = 0 at reset, Set OVB
K_DP          .set    000000000b<<0         ; DP is cleared to 0 at reset
K_ST0         .set    K_ARP|K_TC|K_C|K_OVA|K_OVB|K_DP
*****************************************************************
*ST1 Register Organization
*
*  ---------------------------------------------------------------
*  | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 |  6 |  5 | 4    0 |
*  |---------------------------------------------------------------|
*  |BRAF| CPL| XF | HM |INTM|  0 |OVM|SXM|C16|FRCT|CMPT|  ASM   |
*  ---------------------------------------------------------------
*
*******************************************************************************
K_BRAF        .set    0b << 15              ; BRAF = 0 at reset
K_CPL         .set    0b << 14              ; CPL = 0 at reset
K_XF          .set    1b << 13              ; XF = 1 at reset
K_HM          .set    0b << 12              ; HM = 0 at reset
K_INTM        .set    1b << 11              ; INTM
K_ST1_RESR    .set    0b << 10              ; reserved
K_OVM         .set    1b << 9               ; OVM = 0 at reset
K_SXM         .set    1b << 8               ; SXM = 1 at reset
K_C16         .set    0b << 07              ; C16 = 0 at reset
K_FRCT        .set    1b << 06              ; FRCT = 0 at reset, Set FRCT
K_CMPT        .set    0b << 05              ; CMPT = 0 at reset
K_ASM         .set    00000b << 00          ; ASM = 0 at reset
K_ST1_HIGH    .set    K_BRAF|K_CPL|K_XF|K_HM|K_INTM|K_ST1_RESR|K_OVM|K_SXM
K_ST1_LOW     .set    K_C16|K_FRCT|K_CMPT|K_ASM
K_ST1         .set    K_ST1_HIGH|K_ST1_LOW
```

Example 10–3. Main Program That Calls Different Functions (Continued)

```
*******************************************************************************
*PMST Register Organization
*  ------------------------------------------------
*  | 15    7 |   6  | 5  | 4  | 3  |   2  | 1      0  |
*  |---------|------|----|----|----|------|-----------
*  |   IPTR  |MP/MC |OVLY|AVIS|DROM|CLKOFF| Reserved  |
*  ------------------------------------------------
K_IPTR        .set   111111111b <<  07       ; 111111111b at reset
K_MP_MC       .set   1b << 06                ; 1 at reset
K_OVLY        .set   Ob << 05                ; OVLY = 0 at reset
K_AVIS        .set   Ob << 04                ; AVIS = 0 at reset
K_DROM        .set   Ob << 03                ; DROM = 0 at reset
K_CLKOFF      .set   Ob << 02                ; CLKOFF = 0 at reset
K–PMST_RESR   .set   OOb << 0                ; reserved
                                             ; for 548  bit 0 = SMUL
                                             ; saturation on multiply
                                             ; bit 1 = SST = saturation on store
K_PMST        .set   K_IPTR|K_MP_MC|K_OVLY|K_AVIS|K_DROM|K_CLKOFF|K_PMST_RESR
*******************************************************************************
*SWWSR Register Organization
*  ------------------------------------------------------------
*  |      15     |14    12|11     9|8    6| 5   3  |  2   0  |
*  ------------------------------------------------------------
*  |  Reserved  |  I/O  |  Data   |  Data |Program|  Program |
*  ------------------------------------------------------------
*******************************************************************************
K_SWWSR_IO    .set                  2000h                  ; set the I/O space
*******************************************************************************
*Bank Switching Control Register (BSCR)Organization
*  ------------------------------------------------
*  |  15     12 |  11     | 10         2 |  1 |  0   |
*  ------------------------------------------------
*  |  BNKCMP    |  PS–DS  |  Reserved    | BH | EXIO |
*  ------------------------------------------------
*******************************************************************************
K_BNKCMP      .set    0000b << 12            ; bank size = 64K
K_PS_DS       .set    0b << 11
K_BSCR_RESR   .set    000000000b <<2         ; reserved space
K_BH          .set    0b << 1                ; BH = 0 at reset
K_EXIO        .set    0b << 0                ; EXIO = 0 at reset
K_BSCR        .set    K_BNKCMP|K_PS_DS|K_BSCR_RESR|K_BH|K_EXIO
```

Example 10–4. Processor Initialization

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    init_54x.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR       Padma P. Mallela
;
;              Application Specific Products
;              Data Communication System Development
;              12203 SW Freeway, MS 701
;              Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /     AUTHORS           COMMENT
;          1.0     July–29–96  /    P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains initialization of the processor
;
;
;    1.3 Specification/Design Reference (optional)
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s init_54x.asm
;
;    1.6 Notes and Special Considerations
;{
;    This code is written for 541 device. The code is tested on C54x EVM
;
```

Example 10–4. Processor Initialization (Continued)

```
;}
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
        .mmregs
        .include        ”init_54x.inc”
;    3.2 External Data
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
        .def    init_54
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 init_54
;
;    5.2 Functional Description
;        Initializes the processor from a reset state
;-------------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
;                CALL    init_54
;
;        Reentrancy:     No
;        Recursive :     No
;
;    5.4 Inputs
```

Example 10–4. Processor Initialization (Continued)

```
;
;    5.5 Outputs
;
;    5.6 Global
;
;
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |NU|NU |NU |NU |NU  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|NU|NU |NU |NU |NU  |NU |NU |NU |NU |NU |UM |UM |NU |NU |NU|NU|NU|NU |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:    not applicable for this application
;
;=============================================================================
;        HeaderEnd
;    5.10 Code
    .sect       ”main_prg”
    init_54:
; Init.the s/w wait state reg.for 2 wait states for I/O operations
    STM           #K_SWWSR_IO, SWWSR          ; 2 wait states for I/O operations
; wait states for Bank Switch
    STM           #K_BSCR, BSCR               ; 0 wait states for BANK SWITCH
; initialize the status and control registers
    STM           #K_ST0, ST0
    STM           #K_ST1, ST1
    RETD
    STM           #K_PMST,PMST
    .end
```

Example 10–5. Handshake Between Host and Target

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    hand_shk.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR       Padma P. Mallela
;
;              Application Specific Products
;              Data Communication System Development
;              12203 SW Freeway, MS 701
;              Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
; (C) Copyright 1996. Texas Instruments. All rights reserved.
;{
; Change history:
;
;        VERSION       DATE     /     AUTHORS           COMMENT
;          1.0     July–26–96   /    P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains one subroutine:
;        1) evm_handshake
;    1.3 Specification/Design Reference (optional)
;        called by main.asm depending upon if K_HOST_FLAG is set
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s hand_shk.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
;{
```

Example 10–5. Handshake Between Host and Target (Continued)

```
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include        ”target.inc”
         .include        ”init_54x.inc”
         .include        ”interrpt.inc”
;    3.2 External Data
         .ref            FIFO_DP
         .ref            d_command_reg
         .ref            d_command_value
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
        .def    evm_handshake
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;=============================================================================
;
;-----------------------------------------------------------------------------
;    5.1 evm_handshake
;
```

Example 10–5. Handshake Between Host and Target (Continued)

```
;    5.2 Functional Description
;
;        This initiates the handshake between the host(PC) and the target (DSP).
;        The host writes a command to CH A. This generates an INT1 on the target.
;        The AXST bit on HCR is set to 1.The bit in IFR is polled if it is set
;        then it is cleared to clear pending interrupts. The FIFO is cleared
;        by reading from the FIFO. The command from host is read thru CH A and
;        ARST on TCR is cleared. Another command from target is written to CH A,
;        which sets AXST. Also sets XF low. The host polls XF line. The host reads
;        CH A which clears ARST on host side and AXST on target side.
;
;-------------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
;        CALL    evm_handshake
;
;        Reentrancy:     No
;        Recursive :     No
;
;    5.4 Inputs
;            NONE
;    5.5 Outputs
;            NONE
;
;    5.6 Global
;        Data structure:     d_command_reg
;        Data Format:        16–bit variable
;        Modified:           Yes
;        Description:        command from host is read thru CH A
;
;        Data structure:     d_command_value
;        Data Format:        16–bit variable
;        Modified:           Yes
;        Description:        holds the command value

;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |UM|NU|NU|NU |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
;
;===============================================================================
```

Example 10–5. Handshake Between Host and Target (Continued)

```
;HeaderEnd
;    5.10 Code
     .sect    ”handshke”
evm_handshake:
     LD       #0,DP
     BITF     IFR,02h                      ; Poll for INT1
     BC       evm_handshake,NTC            ; ARST = 1
     STM      #K_INT1,IFR                  ; clear the pending interrupt
     LD       #FIFO_DP,DP
     RPT      #K_FIFO_SIZE–1
     PORTR    K_CHB,d_command_reg          ; assures that FIFO is empty to
     PORTR    K_CHA,d_command_value        ; ARST = 0
target_handshake_command:                  ; read the command from HOST
                                           ;to acknowledge INT1
     PORTR    K_TRGCR_ADDR,d_command_reg   ; while (port14 & ARST)
     BITF     d_command_reg,K_ARST         ; check FIFO empty
     BC       target_handshake_command,TC  ; branch occurs
     LD       #K_HANDSHAKE_CMD,A           ; indicate of FIFO empty
     SUB      d_command_value,A
bad_handshake_command
     BC       bad_handshake_command,ANEQ   ; read the command send by hosts
     ST       #K_AXST_CLEAR,d_command_reg  ; send to a command to clear AXST
     PORTW    d_command_reg, K_CHA         ; write command to command reg A
                                           ; AXST = 1
     RSBX     XF                           ; XF = 0
     RET
    .end
```

Example 10–6. Initialization of Variables, Pointers, and Buffers

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    prcs_int.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR       Padma P. Mallela
;
;              Application Specific Products
;              Data Communication System Development
;              12203 SW Freeway, MS 701
;              Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
; (C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /     AUTHORS           COMMENT
;          1.0     July–29–96  /    P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains initialization of buffers,pointers and variables
;
;
;    1.3 Specification/Design Reference (optional)
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s prcs_int.asm
;
;    1.6 Notes and Special Considerations
;{
; This code is written for 541 device. The code is tested on C54x EVM
;
```

Example 10–6. Initialization of Variables, Pointers, and Buffers (Continued)

```
;}
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include   ”main.inc”
;    3.2 External Data
         .ref       input_data,output_data
         .ref       present_command
         .ref       d_rcv_in_ptr,d_xmt_out_ptr
         .ref       RCV_INT1_DP
         .ref       d_buffer_count
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def    init_bffr_ptr_var
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 init_bffr_ptr
;
;    5.2 Functional Description
;        This routine initializes all the buffers, pointers and variables
;-------------------------------------------------------------------------------
;
```

Example 10–6. Initialization of Variables, Pointers, and Buffers (Continued)

```
;    5.3 Activation
;        Activation example:
;        CALL    init_bffr_ptr_var
;        Reentrancy:     No
;        Recursive :     No
;
;    5.4 Inputs
;        NONE
;    5.5 Outputs
;        NONE
;    5.6 Global
;
;        Data structure:     AR2
;        Data Format:        16–bit input buffer pointer
;        Modified:           Yes
;        Description:        initialize to the starting address
;
;        Data structure:     AR3
;        Data Format:        16–bit output buffer pointer
;        Modified:           Yes
;        Description:        initialize to the starting address
;
;        Data structure:     present_command
;        Data Format:        16–bit variable
;        Modified:           Yes
;        Description:        holds the present command
;
;        Data structure:     input_data
;        Data Format:        16–bit array
;        Modified:           Yes
;        Description:        address of the input data buffer
;
;        Data structure:     output_data
;        Data Format:        16–bit array
;        Modified:           Yes
;        Description:        address of the output data buffer
;
;        Data structure:     d_rcv_in_ptr
;        Data Format:        16–bit var
;        Modified:           Yes
;        Description:        holds the staritng address of input bffr
;
;        Data structure:     d_xmt_out_ptr
;        Data Format:        16–bit variable
;        Modified:           Yes
;        Description:        holds the starting address of output bffr
;
;    5.7 Special considerations for data structure
;        –
```

Example 10–6. Initialization of Variables, Pointers, and Buffers (Continued)

```
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |0 | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|0 |NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|2 | 1 | 1 |NU | 1  |NU |NU |UM |UM |UM |NU |NU |NU |NU |UM|NU|0 |NU |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
; ;=============================================================================
;HeaderEnd
;  5.10 Code
          .asg      AR1,ZRPAD_P                 ; zero pad pointer
          .asg      AR2,GETFRM_IN_P             ; get frame input data pointer
          .asg      AR3,GETFRM_OUT_P            ; get frame output data pointer
          .asg      AR2,COFF_P
          .sect     ”zeropad”
init_bffr_ptr_var:
          STM       #RCV_INT1_DP,AR2            ; init all vars to 0
          RPTZ      A,K_FRAME_SIZE/2–1          ; this may need mods if all vars
          STL       A, *AR2+                    ; are not in 1 page
          STM       #input_data,GETFRM_IN_P     ; input buffer ptr
          STM       #output_data,GETFRM_OUT_P   ; output buffer ptr
          LD        #RCV_INT1_DP,DP
          MVKD      GETFRM_IN_P,d_rcv_in_ptr    ; holds present in. bffr ptr
          MVKD      GETFRM_OUT_P,d_xmt_out_ptr  ; holds present out bffr ptr
          ST        #3,present_command          ; initialize present command
          ST        #K_0, d_buffercount         ; reset the buffer count
          STM       #input_data,ZRPAD_P
          RPTZ      A,2*K_FRAME_SIZE–1          ; zeropad both bottom 256 in-
put_data
          STL       A, *ZRPAD_P+                ; and fft_data buffers
          STM       #output_data,ZRPAD_P
          RPTZ      A,2*K_FRAME_SIZE–1
          STL       A, *ZRPAD_P+
          RET
```

Example 10–7. Initialization of Serial Port 1

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    init_ser.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR     Padma P. Mallela/Ramesh A Iyer
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
; (C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /    AUTHORS            COMMENT
;          1.0     July–29–96  /   P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains the initialization of the serial port 1
;
;    1.3 Specification/Design Reference (optional)
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s init_ser.asm
;
;    1.6 Notes and Special Considerations
;{
;    This code is written for 541 device. The code is tested on C54x EVM ;
;}
```

Example 10–7. Initialization of Serial Port 1 (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include    ”interrpt.inc”
         .include    ”init_ser.inc”
;    3.2 External Data
;        NONE
;    3.3 Import Functions
;        NONE
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
         AIC_VAR_DP      .usect  ”aic_vars”,0
         aic_in_rst      .usect  ”aic_vars”,1
         aic_out_of_rst  .usect  ”aic_vars”,1
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def    serial_init
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 serial_init
;
;    5.2 Functional Description
;        This routine initializes the serial port 1 of 541. The serial port is put
;        in reset by writting 0’s to RRST and XRST bits and pulled out of reset by
;        writting 1’s to both RRST and XRST bits. This routine also puts the AC01
;        in reset and after 12 cycles the AC01 is pulled out of reset. The serial
;        port initialization is done during the 12 cylce latency of the AC01 init.
;
;-------------------------------------------------------------------------------
```

Example 10–7. Initialization of Serial Port 1 (Continued)

```
;
;    5.3 Activation
;        Activation example:
;        CALL    serial_init
;
;        Reentrancy:    No
;        Recursive :    No
;
;    5.4 Inputs
;
;    5.5 Outputs
;
;    5.6 Global
;
;        Data structure: aic_in_rst
;        Data Format:    16-bit variable
;        Modified:       Yes
;        Description:    holds the value to put AC01 in reset state
;
;        Data structure: aic_out_of_reset
;        Data Format:    16-bit variable
;        Modified:       Yes
;        Description:    holds the value to put AC01 out of reset state
;
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |0 | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
;
;=============================================================================
;HeaderEnd
;    5.10 Code
            .sect        ”ser_cnfg”
serial_init:
          LD        #AIC_VAR_DP,DP               ; initialize DP for aic_reset
          ST        #K_0, aic_in_rst             ; bit 15 = 0 of TCR resets AIC
          PORTW     aic_in_rst,K_TRGCR_ADDR      ;write to address 14h (TCR)
*****************************************************************************
*We need at least 12 cycles to pull the AIC out of reset.
*****************************************************************************
```

Example 10–7. Initialization of Serial Port 1 (Continued)

```
          STM       #K_SERIAL_RST, SPC1           ;reset the serial port with
                                                  ;0000 0000 0000 1000
          STM       #K_SERIAL_OUT_RST, SPC1       ;bring ser.port out of reset
                                                  ;0000 0000 1100 1000
          RSBX      INTM
          LD        #0,DP
          ORM       #(K_RINT1|K_INT1),IMR         ; Enable RINT1,INT1
                                                  ; 0000 0000 0100 0010
          LD        #AIC_VAR_DP,DP                ; restore DP
          STM       #(K_RINT1),IFR                ; clear RINT1
          STM       #K_0,DXR1                     ; 0000 0000 0100 0000
; Pull the AC01 out of reset – the AC01 requires that it be held in reset for
; 1 MCLK, which is equivalent to 96.45ns (based on an MCLK of 10.368MHz)
          ST        #K_8000, aic_out_of_rst       ; bit 15 = 1 brings AIC from
reset
          RETD
          PORTW aic_out_of_rst, K_TRGCR_ADDR      ; AIC out of reset
          .end
*************************************************************************
* FILENAME: “INIT_SER.INC“
*  ------------------------------------------------------
*  | 15  | 14  |   13   |   12    | 11   |  10  | 9  |  8  |
*  |-----|-----|--------|---------|------|------|----|-----|
*  |FREE | SOFT| RSRFULL| XSREMPTY| XRDY | RRDY | IN1| IN0 |
*  ------------------------------------------------------
*  ------------------------------------------
*  | 7   | 6   |   5 | 4  |  3  |  2 | 1   |  0  |
*  |-----|-----|-----|----|-----|----|-----|-----|
*  |RRST | XRST| TXM | MCM| FSM | FO | DLB | RES |
*  ------------------------------------------
* This include file includes the SPC1 register configuration
*************************************************************************
;Bit      Name         Function
;0     Reserved     Always read as 0
;1     DLB          Digital loop back: 0 –> Disabled, 1_.Enabled
;2     FO           Format bit: 0 –> data transfered as 8 bit bytes, 1 –> 16 bit
                    words
;3     FSM          Frame sync pulse: 0 –> serial port in continuous mode, 1 –> FSM
                    is required
;4     MCM          Clock mode bit: 0 –> CLKX obtained from CLKX pin 1 –> CLKX
                    obtained from CLKX
;5     TXM          Transmit mode bit: 0 –> Frame sync pulses generated externally
                    and supplied on FSX pin, 1 –> Internally generated frame sync
                    pulses out on FSX pin
;6     XRST         Transmit reset bit: 0 –> reset the serial port, 1 –> bring
                    serial port out of reset
;7     RRST         Receive reset bit: 0 –> reset the serial port, 1 –> bring
                    serial port out of reset
;8     IN0          Read–only bit reflecting the state of the CLKR pin
;9     IN1          Read–only bit reflecting the state of the CLKX pin
;10    RRDY         Transition from 0 to 1 indicates data is ready to be read
;11    XRDY         Transition from 0 to 1 indicates data is ready to be sent
```

Example 10–7. Initialization of Serial Port 1 (Continued)

```
;12    XSREMPTY     Transmit shift register empty  ( Read–only) 0 –> tramsitter
                    has experienced underflow
;13    RSRFUL       Receive shift register full flag (Read–only): 0 –> Receiver
                    has experienced overrun
;14    SOFT         Soft bit – 0 –> immdeiate stop, 1–> stop after word completion
;15    FREE         Free run bit: 0 –> behaviour depends on SOFT bit, 1–> free run
                    regardless of SOFT bit
; The system has the following configuration:
;      Uses 16–bit data => FO = 0
;      Operates in burst mode => FSM = 1
;      CLKX is derived from CLKX pin => MCM = 0
;      Frame sync pulses are generated externally by the AIC => TXM = 0
; Therefore, to reset the serial port, the SPC field would have
;      0000 0000 0000 1000
; To pull the serial port out of reset, the SPC field would have
;      0000 0000 1100 1000
K_0                 .set     00000000b << 8    ; bits 15–8 to 0 at reset
K_RRST              .set     0b << 7           ; First write to SPC1 is 0
                                               ; second write is 1
K_XRST              .set     0b << 6           ; First write to SPC1 is 0
                                               ; second write is 1
K_TXM               .set     0b << 5
K_MCM               .set     0b << 4
K_FSM               .set     1b << 3           ; Frame Sync mode
K_ZERO              .set     000b << 0
K_SERIAL_RST        .set     K_0|K_RRST|K_XRST|K_TXM|K_MCM|K_FSM|K_ZERO
                                               ; first write to SPC1 register
K_RRST1           .set    1b << 7             ; second write to SPC1
K_XRST1           .set    1b << 6             ; second write to SPC1
K_SERIAL_OUT_RST  .set    K_0|K_RRST1|K_XRST1|K_TXM|K_MCM|K_FSM|K_ZERO

K_TRGCR_ADDR      .set    14h                 ; Target/Status I/O address

K_0               .set    0h
K_8000            .set    8000h               ; set bit 15 to pull AIC out
                                              ; of reset
* FILENAME: INTERRUPT.INC
*  ------------------------------------------------------------------------
*  | 15     9 |  8  |  7   |   6  |  5   |  4   | 3   | 2   | 1   |  0  |
*  |----------|-----|------|------|------|------|-----|-----|-----|-----|
*  | Reserved | INT3| XINT1| RINT1| XINT0| RINT0| TINT| INT2| INT1| INT0|
*  ------------------------------------------------------------------------
* This file includes the IMR and IFR configuration
*****************************************************************************
K_IMR_RESR        .set    0000000b << 9       ; reserved space
K_INT3            .set    1b << 8             ; disable INT3
K_XINT1           .set    1b << 7             ; disable transmit interrupt 1
K_RINT1           .set    1b << 6             ; enable receive interrupt 1
K_XINT0           .set    1b << 5             ; disable transmit interrupt 0
K_RINT0           .set    1b << 4             ; disable receive interrupt
K_TINT            .set    1b << 3             ; disbale timer interrupt
K_INT2            .set    1b << 2             ; disable INT2
K_INT1            .set    1b << 1             ; disbale INT1
K_INT0            .set    1b << 1             ; enable INT0
```

Example 10–8. ’AC01 Initialization

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    init_aic.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR       Padma P. Mallela/Ramesh A. Iyer
;
;              Application Specific Products
;              Data Communication System Development
;              12203 SW Freeway, MS 701
;              Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /    AUTHORS           COMMENT
;          1.0     July–29–96/     P.Mallela      original created
;
;}
;{
;    1.  ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains the initialization of AC01
;
;
;    1.3 Specification/Design Reference (optional)
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s init_aic.asm
;
;    1.6 Notes and Special Considerations
;{
;        This code is written for 541 device. The code is tested on C54x EVM ;
;}
```

Example 10–8. ’AC01 Initialization (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include        ”interrpt.inc”
;    3.2 External Data
;        NONE
;    3.3 Import Functions
         .ref    wrt_cnfg                        ; initializes AC01
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def    aic_init
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;===============================================================================
;
;
;-------------------------------------------------------------------------------
;    5.1 aic_init
;
;    5.2 Functional Description
;        This routine disables IMR and clears any pending interrupts before
;        initializing AC01. The wrt_cnfg function configures the AC01
;--------------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
;                CALL    aic_init
;
;        Reentrancy:     No
;        Recursive :     No
```

Example 10–8. ’AC01 Initialization (Continued)

```
;
;    5.4 Inputs
;
;    5.5 Outputs
;
;    5.6 Global
;
;
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |  |0  | 1 | 1 |NU  | 1 |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|  |0  | 1 | 1 |NU  | 1 |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
;
;===========================================================================
;        HeaderEnd
;    5.10 Code
         .sect      ”aic_cnfg”
aic_init:
         CALLD      wrt_cnfg                        ; initialize AC01
         ANDM       #(~K_RINT1|K_INT1),IMR          ; disable receive_int1,INT1
         ORM        #(K_RINT1|K_INT1),IMR           ; enable the RINT1, INT1
         RETD
         STM        (K_RINT1),IFR                   ; service any pending interrupt
        .end
*****************************************************************
* This file includes the AC01 registers initialization
* All registers have 2 control bits that initiates serial communication
* There are 2 communication modes – primary and secondary communications
* During primary communication the control bits D00 and D01 are 11 to request
* for a secondary communication. In the secondary serial communications the
* control bits D15 and D14 perform same control function as primary.
* The R/W~ bit at reset is set to 0 placing the device in write mode.
*****************************************************************
K_NOP_ADDR        .set    0 << 8
K_REG_0           .set    K_NOP_ADDR
K_A_ADDR          .set    1 << 8                ; REG 1 address
K_A_REG           .set    36                    ;
K_REG_1           .set    K_A_ADDR|K_A_REG      ; FCLK = 144KHz => A =24h
K_B_ADDR          .set    2 << 8                ; REG 2 address
K_B_REG           .set    18                    ;
K_REG_2           .set    K_B_ADDR|K_B_REG      ; Sampling rate = 8KHz
K_AA_ADDR         .set    3 << 8                ; Register 3 address
K_AA_REG          .set    0                     ;
```

Example 10–8. 'AC01 Initialization (Continued)

```
K_REG_3           .set   K_AA_ADDR|K_AA_REG;    ; no shift
K_GAIN_ADDR       .set   4 << 8                 ; Register 4 address
K_MONITOR_GAIN    .set   00b << 4               ; Monitor output gain = squelch
K_ANLG_IN_GAIN    .set   01b << 2               ; Analog input gain = 0dB
K_ANLG_OUT_GAIN   .set   01b << 0               ; Analog output gain = 0dB
K_REG_4           .set   K_GAIN_ADDR|K_MONITOR_GAIN|K_ANLG_IN_GAIN|K_ANLG_OUT_GAIN
K_ANLG_CNF_ADDR   .set   5 << 8                 ; Register 5 address
K_ANLG_RESRV      .set   0 << 3                 ; Must be set to 0K_HGH_FILTER .set 0 << 2
                                                ; High pass filter is enabled
K_ENBL_IN         .set   01b << 0               ; Enables IN+ and IN–
K_REG_5           .set   K_ANLG_CNF_ADDR|K_ANLG_RESRV|K_HGH_FILTER|K_ENBL_IN
K_DGTL_CNF_ADDR   .set   6 << 8                 ; Register 6 address
K_ADC_DAC         .set   0 << 5                 ; ADC and DAC is inactive
K_FSD_OUT         .set   0 << 4                 ; Enabled FSD output
K_16_BIT_COMM     .set   0 << 3                 ; Normal 16–bit mode
K_SECND_COMM      .set   0 << 2                 ; Normal secondary communication
K_SOFT_RESET      .set   0 << 1                 ; Inactive reset
K_POWER_DWN       .set   0 << 0                 ; Power down external
K_REG_HIGH_6      .set   K_DGTL_CNF_ADDR|K_ADC_DAC|K_FSD_OUT|K_16_BIT_COMM
K_REG_LOW_6       .set   K_SECND_COMM|K_SOFT_RESET|K_POWER_DWN
K_REG_6           .set   K_REG_HIGH_6|K_REG_LOW_6
K_FRME_SYN_ADDR   .set   7 << 8                 ; Register 7 address
K_FRME_SYN        .set   0 << 8                 ;
K_REG_7           .set   K_FRME_SYN_ADDR|K_FRME_SYN
K_FRME_NUM_ADDR   .set   8 << 8                 ; Register 8 address
K_FRME_NUM        .set   0 << 8                 ;
K_REG_8           .set   K_FRME_NUM_ADDR|K_FRME_NUM
; primary word with D01 and D00 bits set to 11 will cause a
; secondary communications interval to start when the frame
; sync goes low next
K_SCND_CONTRL     .set   11b << 0               ; Secondary comm.bits
AIC_REG_START_LIST .sect   "aic_reg"    ; includes the aic table
    .word         AIC_REG_END_LIST–AIC_REG_START_LIST–1
    .word         K_REG_1
    .word         K_REG_2
    .word         K_REG_3
    .word         K_REG_4
    .word         K_REG_5
    .word         K_REG_6
    .word         K_REG_7
    .word         K_REG_8
AIC_REG_END_LIST
K_XRDY  .set 0800h                              ; XRDY bit in SPC1
    .sect         "aic_cnfg"
aic_init:
    CALLD         wrt_cnfg                      ; initialize AC01
    ANDM          #(~K_RINT1),IMR               ; disable receive_int1
    ORM           #(K_RINT1|K_INT1),IMR         ; enable the RINT1, INT1
    RETD
    STM           #(K_RINT1),IFR                ; service any pending interrupt
    .end
```

Example 10–9. ’AC01 Register Configuration

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    aic_cfg.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR     Padma P. Mallela/Ramesh A. Iyer
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX  77477
;
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;VERSION      DATE     /    AUTHORS           COMMENT
;  1.0     July–25–96  /   P.Mallela      original created
;
;}
;{
;    1.  ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains the AC01 initialization
;
;
;    1.3 Specification/Design Reference (optional)
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s aic_cfg.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–9. ’AC01 Register Configuration (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
        .mmregs
        .include        ”aic_cfg.inc”
;    3.2 External Data
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
       .def    wrt_cnfg
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 wrt_cnfg
;
;    5.2 Functional Description
;        Writes new configuration data into the AC01. Assuming a system
;        which processes speech signals and *  requires the following parameters
;        Low pass filter cut–off frequency = 3.6 kHz
;        Sampling rate = 8000 Hz
;        Assume the Master clock MCLK = 10.368 MHz
;        This example demonstrates how to program these parameters –
;        the registers affected are:
;        Register A which determines the division of the MCLK frequency
;        to generate the internal filter clock FCLK.
;        It also determines the –3 dB corner frequency of the low–pass filter
;        Register B  which determines the division of FCLK to generate
;        the sampling (conversion) frequency
```

Example 10–9. ’AC01 Register Configuration (Continued)

```
;        It also determines the –3dB corner frequency of the high–pass filter
;-------------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
;                CALLD   wrt_cnfg
;                STM     #K_RINT1,IFR
;
;        Reentrancy:     No
;        Recursive :     No
;
;    5.4 Inputs
;        NONE
;
;    5.5 Outputs
;        NONE
;
;    5.6 Global
;        Data structure: AR1
;        Data Format:    16–bit pointer
;        Modified:       No
;        Description:    indexes the table
;
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;

;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |NU |UM |NU |NU |UM |NU |NU |NU |UM|NU|NU|UM |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
; HeaderEnd
;   5.10 Code
;
;===============================================================================
    .asg    AR1,AIC_REG_P
    .sect   ”aic_cnfg”
wrt_cnfg:
    STM     #aic_reg_tble,AIC_REG_P             ; init AR1
    RPT     #AIC_REG_END_LIST–AIC_REG_START_LIST
    MVPD    #AIC_REG_START_LIST,*AIC_REG_P+     ; move the table
    STM     #aic_reg_tble,AIC_REG_P             ; init AR1
    STM     #K_REG_0,DXR1                       ; primary data word –
                                                ; a jump start!
```

Example 10–9. ’AC01 Register Configuration (Continued)

```
wait_xrdy
    BITF    SPC1,K_XRDY                     ; test XRDY bit in SPC1
    BC      wait_xrdy,NTC                   ; loop if not set
    STM     #K_SCND_CONTRL,DXR1             ; send primary word with
                                            ; D01–D00  = 11 to
                                            ; signify secondary communication
    LD      *AIC_REG_P+,A
    STLM    A,BRC                           ; gives the # of registers to be
                                            ; initialized
            NOP
    RPTB    aic_cfg_complte–1
wait_xrdy1
    BITF    SPC1,K_XRDY                     ; test XRDY bit in SPC1
    BC      wait_xrdy1,NTC                  ; loop if not set
    LD      *AIC_REG_P+,A                   ; Read the register contents
    STLM    A, DXR1
wait_xrdy2
    BITF    SPC1,K_XRDY                     ; test XRDY bit in SPC1
    BC      wait_xrdy2,NTC                  ; loop if not set
    STM     #K_SCND_CONTRL,DXR1             ; set to read the next register
aic_cfg_complte                             ; contents
wait_xrdy3
    BITF    SPC1,K_XRDY                     ; test XRDY bit in SPC1
    BC      wait_xrdy3,NTC                  ; loop if not set
    RET
```

Example 10–10. Receive Interrupt Service Routine

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    rcv_int1.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR     Padma P. Mallela
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /    AUTHORS           COMMENT
;          1.0     July–29–96  /   P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains interrput service routine INT1:
;        receive_int1
;    1.3 Specification/Design Reference (optional)
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s rcv_int1.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
;{
```

Example 10–10. Receive Interrupt Service Routine (Continued)

```
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
; 3 EXTERNAL RESOURCES
;
;     3.1 Include Files
          .mmregs
          .include    ”INTERRPT.INC”
          .include    ”main.inc”
;     3.2 External Data
          .ref        d_frame_flag
          .ref        d_index_count
          .ref        d_rcv_in_ptr,d_xmt_out_ptr
          .ref        RCV_INT1_DP
;     3.3 Import Functions
;}
;{
; 4 INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
          .def        receive_int1
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;=============================================================================
;
;-----------------------------------------------------------------------------
;    5.1 receive_int1
;
;    5.2 Functional Description
;        This routine services receive interrupt1. Accumulator A, AR2 and AR3
;        are pushed onto the stack since AR2 and AR3 are used in other
;        applications. A 512 buffer size of both input and output uses circular
;        addressing. After every 256 collection of input samples a flag is set to
;        process the data. A PING/PONG buffering scheme is used such that upon
;        processing PING buffer, samples are collected in the PONG buffer and vice
;        versa.
;-----------------------------------------------------------------------------
```

Example 10–10. Receive Interrupt Service Routine (Continued)

```
;    5.3 Activation
;        Activation example:
;            BD    receive_int1
;            PSHM  ST0
;            PSHM  ST1
;
;        Reentrancy:    No
;        Recursive :    No
;
;    5.4 Inputs
;        NONE
;
;    5.5 Outputs
;        NONE
;
;   5.6 Global
;
;        Data structure:     AR2
;        Data Format:        16–bit input buffer pointer
;        Modified:           Yes
;        Description:        either point to PING/PONG buffer. Upon entering
;                            AR2 is pushed onto stack and the address is restored
;                            through d_rcv_in_ptr
;
;        Data structure:     AR3
;        Data Format:        16–bit output buffer pointer
;        Modified:           Yes
;        Description:        either point to PING/PONG buffer. Upon entering
;                            AR3 is pushed onto stack and the address is restored
;                            through d_rcv_in_ptr
;
;        Data structure:     d_index_count
;        Data Format:        16–bit var
;        Modified:           Yes
;        Description:        holds the number samples that has been collected from
;                            AC01
;
;        Data structure:     d_frame_flag
;        Data Format:        16–bit variable
;        Modified:           Yes
;        Description:        flag is set if 256 samples are collected
;
;        Data structure:     d_rcv_in_ptr
;        Data Format:        16–bit var
;        Modified:           Yes
;        Description:        holds the input buffer address where the newest
;                            sample is stored
;
;        Data structure:     d_xmt_out_ptr
;        Data Format:        16–bit variable
;        Modified:           Yes
;        Description:        holds the output buffer address where the oldest
;                            sample is sent as output
```

Example 10–10. Receive Interrupt Service Routine (Continued)

```
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |U  |U  |NU |NU |NU |NU |U |NU|NU|NU |NU|NU
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |
;out|U | 1 | 1 |NU | 1  |NU |NU |NU |U  |U  |NU |NU |NU |NU |U |NU|U |NU |NU|NU
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
;
;=============================================================================
;HeaderEnd
; 5.10 Code
         .asg    AR2,GETFRM_IN_P              ; get frame input data pointer
         .asg    AR3,GETFRM_OUT_P             ; get frame output data pointer
         .asg    AR2,SAVE_RSTORE_AR2
         .asg    AR3,SAVE_RSTORE_AR3
         .sect   ”main_prg”
receive_int1:
         PSHM    AL
         PSHM    AH
         PSHM    AG
         PSHM    SAVE_RSTORE_AR2
         PSHM    SAVE_RSTORE_AR3
         PSHM    BK
         PSHM    BRC
         STM     #2*K_FRAME_SIZE,BK           ; circular buffr size of in,out
                                              ; arrays
         LD      #RCV_INT1_DP,DP              ; init. DP
         MVDK    d_rcv_in_ptr,GETFRM_IN_P     ; restore input circular bffr ptr
         MVDK    d_xmt_out_ptr,GETFRM_OUT_P   ; restore output circular bffr ptr
         ADDM    #1,d_index_count             ; increment the index count
         LD      #K_FRAME_SIZE,A
         SUB     d_index_count, A
         BC      get_samples,AGT              ;check for a frame of samples
frame_flag_set
          ST      #K_FRAME_FLAG,d_frame_flag  ; set frame flag
          ST      #0,d_index_count            ; reset the counter
```

Example 10–10. Receive Interrupt Service Routine (Continued)

```
get_samples
          LDM     DRR1,A                          ; load the input sample
          STL     A,*GETFRM_IN_P+%                ; write to buffer
          LD      *GETFRM_OUT_P+%,A               ; if not true, then the filtered
          AND     #0fffch,A                       ; signal is send as output
          STLM    A,DXR1                          ; write to DXR1
          MVKD    GETFRM_IN_P,d_rcv_in_ptr        ; save input circualr buffer ptr
          MVKD    GETFRM_OUT_P,d_xmt_out_ptr      ; save out circular bffr ptr
          POPM    BRC
          POPM    BK
          POPM    SAVE_RSTORE_AR3
          POPM    SAVE_RSTORE_AR2
          POPM    AG
          POPM    AH
          POPM    AL
          POPM    ST1
          POPM    ST0
          RETE                                    ; return and enable interrupts
```

Example 10–11. Task Scheduling

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    task.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR     Padma P. Mallela
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /    AUTHORS           COMMENT
;          1.0     July–29–96  /   P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file initiates task scheduling
;
;
;    1.3 Specification/Design Reference (optional)
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s task.asm
;
;    1.6 Notes and Special Considerations
;{
; This code is written for 541 device. The code is tested on C54x EVM
;
```

Example 10–11. Task Scheduling (Continued)

```
;}
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
        .include        ”init_54x.inc”
        .include        ”main.inc”
;    3.2 External Data
        .ref            d_task_addr,d_task_init_addr
        .ref            d_buffer_count
        .ref            present_command
        .ref            last_command
        .ref            d_output_addr,d_input_addr
        .ref            input_data,output_data
        .ref            d_frame_flag
        .ref            task_init_list,task_list
;    3.3 Import Functions
        .ref            echo_task
        .ref            fir_init,fir_task
        .ref            do_nothing,no_echo_init_task
        .ref            fir_init,fir_task
        .ref            iir_init,iir_task
        .ref            sym_fir_task,sym_fir_init
        .ref            adapt_init,adapt_task
        .ref            rfft_task
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
        .def    task_handler
;}
```

Example 10–11. Task Scheduling (Continued)

```
;    5. SUBROUTINE CODE
;       HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 task_handler
;
;    5.2 Functional Description
;        This routine handles the task scheduling. The present_command
;        can take values 1,2,3,4,5,6. If
;        present_command = 1 – Echo program is enabled
;        present_command = 2 – FIR is enabled
;        present_command = 3 – IIR is enabled
;        present_command = 4 – Symmetric FIR is enabled
;        present_command = 5 – Adaptive fitler is enabled
;        present_command = 6 – FFT is enabled
;        For every cycle the program checks if the current task is same
;        as previous task if it is, then no initialization is done.
;        If its not then cirular buffers, variables pointers are intialized
;        depending upon the task.
;-------------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
;        CALL    task_handler
;        Reentrancy:     No
;        Recursive :     No
;
;    5.4 Inputs
;
;        Data structure: present_command
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    holds the present command
;
;    5.5 Outputs
;
;        Data structure: AR6
;        Data Format:    16–bit input buffer pointer
;        Modified:       Yes
;        Description:    either point to PING/PONG buffer
;
;        Data structure: AR7
;        Data Format:    16–bit output buffer pointer
;        Modified:       Yes
;        Description:    either point to PING/PONG buffer
;
;    5.6 Global
;
;        Data structure: last_command
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    holds the last command
```

Example 10–11. Task Scheduling (Continued)

```
;
;        Data structure: d_frame_flag
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    gets reset after 256 samples
;
;        Data structure: d_buffer_count
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    used to load either PING/PONG bffr addresses
;
;        Data structure: input_data
;        Data Format:    16–bit array
;        Modified:       Yes
;        Description:    address of the input data buffer
;
;        Data structure: output_data
;        Data Format:    16–bit array
;        Modified:       Yes
;        Description:    address of the output data buffer
;
;        Data structure: d_input_addr
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    holds either PING/PONG address
;
;        Data structure: d_output_addr
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    holds either PING/PONG address
;
;        Data structure: d_task_addr
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    holds the task program address
;
;        Data structure: d_task_init_addr
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    holds the task init. address
;
;    5.7 Special considerations for data structure
;    –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |  |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |U  |NU|NU|NU|NU |NU|NU
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |  |
;out|U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |U  |U  |UM|UM|NU|NU |NU|NU
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
```

Example 10–11. Task Scheduling (Continued)

```
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
;
;===========================================================================
;HeaderEnd
;    5.10 Code
         .asg     AR6,INBUF_P                 ; PING/PONG input buffer
         .asg     AR7,OUTBUF_P                ; PING/PONG output buffer
         .sect    ”task_hnd”
task_handler:
;      LD        #TASK_VAR_DP,DP
       ST        #K_0,d_frame_flag           ; reset the frame flag
       ADDM      #1,d_buffer_count
       LD        #input_data,A               ; load PING input address
       LD        #output_data,B              ; load PING output address
       BITF      d_buffer_count,2            ; check if PING/PONG address
       BC        reset_buffer_address,NTC    ; needs to be loaded
       ADD       #K_FRAME_SIZE,A             ; PONG input address
       ADD       #K_FRAME_SIZE,B             ; PONG output address
       ST        #K_0,d_buffer_count         ; reset counter
reset_buffer_address
       STLM      A,INBUF_P                   ; input buffer address
       STL       A,d_input_addr              ; restore either PING/PONG bffr
       STLM      B,OUTBUF_P                  ; output buffer address
       STL       B,d_output_addr             ; restore either PING/PONG bffr
       LD        present_command,A
       SUB       last_command,A
       BC        new_task,ANEQ               ; check if PC = LC
       LD        d_task_addr,A               ; task_addr should
                                             ; contain previous
                                             ; PC = LC
       CALA      A                           ; call present task
       RET
new_task:
       MVKD      present_command,last_command  ; restore the present command
       LD        #task_init_list,A           ; loads PC init task
       ADD       present_command,A           ; computes the present task
       REDA      d_task_init_addr            ; save the PC into task_addr
       LD        d_task_init_addr,A
       CALA      A                           ; initializes the present task
       LD        #task_list,A;
       ADD       present_command,A           ; computes the present task
       READA     d_task_addr                 ; save the PC into
                                             ; task_addr
       LD        d_task_addr,A
       CALA      A
       RET
do_nothing:
       RET
no_echo_init_task:
       RET
       .end
```

Example 10–12. Echo the Input Signal

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    echo.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR     Padma P. Mallela
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;{
; Change history:
;
;        VERSION      DATE     /    AUTHORS           COMMENT
;          1.0     July–24–96  /   P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains one subroutines:
;            echo_task
;    1.3 Specification/Design Reference (optional)
;        called by task.asm depending upon the task
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s echo.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–12. Echo the Input Signal (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
        .mmregs
        .include        ”main.inc”
;    3.2 External Data
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
        .def    echo_task
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 echo_task
;
;    5.2 Functional Description
;        This function reads a sample from either PING/PONG buffer and puts it
;        back in the output buffer. This is repeated 256 times i.e., size of the
;        frame. The present command in this case is 1.
;-------------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
;            CALL    echo_task
;        Reentrancy:     No
;        Recursive :     No
;
```

Example 10–12. Echo the Input Signal (Continued)

```
;    5.4 Inputs
;
;                   Data structure: AR6
;                   Data Format:    16–bit input buffer pointer
;                   Modified:       Yes
;                   Description:    either point to PING/PONG buffer
;
;    5.5 Outputs
;
;                   Data structure: AR7
;                   Data Format:    16–bit output buffer pointer
;                   Modified:       Yes
;                   Description:    either point to PING/PONG buffer
;
;    5.6 Global
;
;
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |U  |U  |NU|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |UM |UM |UM|NU|NU|UM |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;            Execution time: ?cycles
;            Call rate:      not applicable for this application
; ;=============================================================================
;HeaderEnd
;    5.10 Code
          .asg      AR6,INBUF_P                 ; PING/PONG input buffer
          .asg      AR7,OUTBUF_P                ; PING/PONG output buffer
          .sect     ”echo_prg”
echo_task:
          STM       #K_FRAME_SIZE–1,BRC         ; frame size of 256
          RPTB      echo_loop–1
          LD        *INBUF_P+, A                ; load the input value
          STL       A, *OUTBUF_P+
echo_loop
          RET                                   ; output buffer
```

Example 10–13. Low-Pass FIR Filtering Using MAC Instruction

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    fir.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR    Padma P. Mallela
;
;                  Application Specific Products
;                  Data Communication System Development
;                  12203 SW Freeway, MS 701
;                  Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION       DATE     /        AUTHORS             COMMENT
;          1.0      July–26–96  /       P.Mallela         original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;         a.Core Routine
;         b.Subroutine
;
;    1.2 Functional Description
;        This file contains two subroutines:
;        1) fir_init
;        2) fir_task
;    1.3 Specification/Design Reference (optional)
;        called by task.asm depending upon the task thru CALA
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s fir.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–13. Low-Pass FIR Filtering Using MAC Instruction (Continued)

```
;{
;   2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include    ”main.inc”
;    3.2 External Data
         .ref        d_filin                    ; filter input
         .ref        d_filout                   ; filter output
         .ref        d_data_buffer
         .ref        fir_task
         .ref        COFF_FIR_START,COFF_FIR_END
         .ref        fir_coff_table
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def        fir_init                       ; initialize FIR filter
         .def        fir_filter                     ; perform FIR filtering
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 fir_init
;
;    5.1.1 Functional Description
;          This routine initializes cicular buffers both for data and coeffs.
;-------------------------------------------------------------------------------
;
```

Example 10–13. Low-Pass FIR Filtering Using MAC Instruction (Continued)

```
;    5.1.2 Activation
;        Activation example:
;            CALL    fir_init
;        Reentrancy:     No
;        Recursive :     No
;
;    5.1.3 Inputs
;          NONE
;    5.1.4 Outputs
;          NONE
;
;    5.1.5 Global
;
;          Data structure:     AR0
;          Data Format:        16–bit index pointer
;          Modified:           No
;          Description:        uses in circular addressing mode for indexing
;
;          Data structure:     AR4
;          Data Format:        16–bit x(n) data buffer pointer
;          Modified:           Yes
;          Description:        initializes the pointer
;
;          Data structure:     AR5
;          Data Format:        16–bit w(n) coeff buffer pointer
;          Modified:           Yes
;          Description:        initializes the pointer
;
;    5.1.6 Special considerations for data structure
;          –
;    5.1.7 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |UM|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |UM |NU |NU |NU |UM |UM |NU |NU |UM|NU|NU|NU |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.1.8 Execution
;            Execution time: ?cycles
;            Call rate:      not applicable for this application
; ;=============================================================================
;    HeaderEnd
;    5.1.9 Code
            .asg    AR0, FIR_INDEX_P
            .asg    AR4,FIR_DATA_P
            .asg    AR5,FIR_COFF_P
            .sect   ”fir_prog”
```

Example 10–13. Low-Pass FIR Filtering Using MAC Instruction (Continued)

```
fir_init:
            STM     #fir_coff_table,FIR_COFF_P
            RPT     #K_FIR_BFFR–1                   ; move FIR coeffs from program
            MVPD    #COFF_FIR_START,*FIR_COFF_P+    ; to data
            STM     #K_FIR_INDEX,FIR_INDEX_P
            STM     #d_data_buffer,FIR_DATA_P       ; load cir_bfr address for the
                                                    ; recent samples
            RPTZ    A,#K_FIR_BFFR
            STL     A,*FIR_DATA_P+                  ; reset the buffer
            STM     #(d_data_buffer+K_FIR_BFFR–1), FIR_DATA_P
            RETD
            STM     #fir_coff_table, FIR_COFF_P
;    5. SUBROUTINE CODE
;       HeaderBegin
;=============================================================================
;
;-----------------------------------------------------------------------------
;    5.2 fir_task
;
;    5.2.1 Functional Description
;
;          This subroutine performs FIR filtering using MAC instruction.
;          accumulator A (filter output) = h(n)*x(n–i) for i = 0,1...15
;-----------------------------------------------------------------------------
;
;    5.2.2 Activation
;          Activation example:
;          CALL    fir_task
;          Reentrancy:    No
;          Recursive :    No
;
;    5.2.3 Inputs
;
;          Data structure:     AR6
;          Data Format:        16–bit input buffer pointer
;          Modified:           Yes
;          Description:        either point to PING/PONG buffer
;
;          Data structure:     AR4
;          Data Format:        16–bit data buffer pointer
;          Modified:           Yes
;          Description:        uses circular buffer addressing mode to filter
;                              16 tap Low–Pass filter – init. in fir_init
;
;          Data structure:     AR5
;          Data Format:        16–bit coefficient buffer pointer
;          Modified:           Yes
;          Description:        The 16 tap coeffs comprise the low–pass filter
;                                init. in fir_init
```

Example 10–13. Low-Pass FIR Filtering Using MAC Instruction (Continued)

```
;    5.2.4 Outputs
;
;          Data structure:     AR7
;          Data Format:        16–bit output buffer pointer
;          Modified:           Yes
;          Description:        either point to PING/PONG buffer
;
;
;    5.2.5 Global
;          NONE
;

;    5.2.6 Special considerations for data structure
;          –
;    5.2.7 Entry and Exit conditions
;
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |U  |NU |U  | U |UM|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |UM |NU |NU |NU |UM |UM |UM |UM |UM|NU|UM|UM |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.2.8 Execution
;          Execution time: ?cycles
;          Call rate:      not applicable for this application
; ;=============================================================================
;HeaderEnd
;    5.2.9 Code
     .asg    AR6,INBUF_P
     .asg    AR7,OUTBUF_P
     .asg    AR4,FIR_DATA_P
     .asg    AR5,FIR_COFF_P
     .sect   ”fir_prog”
fir_task:
;    LD      #FIR_DP,DP
     STM     #K_FRAME_SIZE–1,BRC              ; Repeat 256 times
     RPTBD   fir_filter_loop–1
     STM     #K_FIR_BFFR,BK                   ; FIR circular bffr size
     LD      *INBUF_P+, A                     ; load the input value
fir_filter:
     STL     A,*FIR_DATA_P+%                  ; replace oldest sample with newest
                                              ; sample
     RPTZ    A,(K_FIR_BFFR–1)
     MAC     *FIR_DATA_P+0%,*FIR_COFF_P+0%,A  ; filtering
    STH      A, *OUTBUF_P+                    ; replace the oldest bffr value
fir_filter_loop
    RET
```

Example 10–14. Low-Pass Symmetric FIR Filtering Using FIRS Instruction

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    sym_fir.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR     Padma P. Mallela
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /    AUTHORS           COMMENT
;          1.0     July–26–96  /   P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains two subroutines:
;        1) sym_fir_init
;        2) sym_fir_task
;
;    1.3 Specification/Design Reference (optional)
;        called by task.asm depending upon the task thru CALA
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s sym_fir.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–14. Low-Pass Symmetric FIR Filtering Using FIRS Instruction (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
        .mmregs
        .include   "main.inc"
;    3.2 External Data
        .ref       d_datax_buffer
        .ref       d_datay_buffer
        .ref       FIR_COFF
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
        .def       sym_fir_init                    ; initialize symmetric FIR
        .def       sym_fir_task
;}
;    5. SUBROUTINE CODE
;    HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1   sym_fir_init
;
;    5.1.1 Functional Description
;          This routine initializes cicular buffers both for data and coeffs.
;-------------------------------------------------------------------------------
;
;    5.1.2 Activation
;          Activation example:
;          CALL            sym_fir_init
;          Reentrancy:     No
;          Recursive :     No
```

Example 10–14. Low-Pass Symmetric FIR Filtering Using FIRS Instruction (Continued)

```
;
;    5.1.3 Inputs
;          NONE
;    5.1.4 Outputs
;          NONE
;
;    5.1.5 Global
;
;          Data structure: AR0
;          Data Format:    16–bit index pointer
;          Modified:       No
;          Description:    uses in circular addressing mode for indexing
;
;          Data structure: AR4
;          Data Format:    16–bit x(n) data buffer pointer for 8 latest samples
;          Modified:       Yes
;          Description:    initializes the pointer
;
;          Data structure: AR5
;          Data Format:    16–bit x(n) data buffer pointer for 8 oldest samples
;          Modified:       Yes
;          Description:    initializes the pointer
;
;    5.1.6 Special considerations for data structure
;          –
;    5.1.7 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |  |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |UM|NU|NU|NU |NU|NU
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |  |
;out|U | 1 | 1 |NU | 1  |NU |UM |NU |NU |NU |U  |UM |NU |NU |UM|NU|NU|NU |NU|NU
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.1.8 Execution
;          Execution time: ?cycles
;          Call rate:      not applicable for this application
;
;===========================================================================
;HeaderEnd
;    5.1.9 Code
          .asg     AR0,SYMFIR_INDEX_P
          .asg     AR4,SYMFIR_DATX_P
          .asg     AR5,SYMFIR_DATY_P
          .sect    ”sym_fir”
sym_fir_init:
          STM      #d_datax_buffer,SYMFIR_DATX_P      ; load cir_bfr address
                                                      ; for the 8 most
                                                      ; recent samples
          STM      #d_datay_buffer+K_FIR_BFFR/2–1,SYMFIR_DATY_P
                                                      ; load cir_bfr1 address
                                                      ; for the 8 old samples
```

Example 10–14. Low-Pass Symmetric FIR Filtering Using FIRS Instruction (Continued)

```
          STM     #K_neg1,SYMFIR_INDEX_P              ; index offset –
                                                      ; whenever the pointer
                                                      ; hits the top of the bffr,
                                                      ; it automatically hits
                                                      ; bottom address of
                                                      ; buffer and decrements
                                                      ; the counter
          RPTZ    A,#K_FIR_BFFR
          STL     A,* SYMFIR_DATX_P+
          STM     #d_datax_buffer, SYMFIR_DATX_P
          RPTZ    A,#K_FIR_BFFR
          STL     A,* SYMFIR_DATY_P–
          RETD
          STM     #d_datay_buffer+K_FIR_BFFR/2–1, SYMFIR_DATY_P
;    5. SUBROUTINE CODE
;       HeaderBegin
;=============================================================================
;
;-----------------------------------------------------------------------------
;    5.2 sym_fir_init
;
;    5.2.1 Functional Description
;          This program uses the FIRS instruction to implement symmetric FIR
;          filter. Circular addressing is used for data buffers.The input scheme
;          for the data samples is divided into two circular buffers. The first
;          buffer contains samples from X(–N/2) to X(–1) and the second buffer
;          contains samples from X(–N) to X(–N/2–1).
;-----------------------------------------------------------------------------
;
;    5.2.2 Activation
;          Activation example:
;          CALL    sym_fir_init
;          Reentrancy:    No
;          Recursive :    No
;
;    5.2.3 Inputs
;
;          Data structure: AR6
;          Data Format:    16–bit input buffer pointer
;          Modified:       Yes
;          Description:    either point to PING/PONG buffer
;
;          Data structure: AR4
;          Data Format:    16–bit data buffer pointer
;          Modified:       Yes
;          Description:    uses circular buffer addressing mode to filter
;                          16 tap Low–Pass filter – init. in sym_fir_init
;
;          Data structure: AR5
;          Data Format:    16–bit coefficient buffer pointer
;          Modified:       Yes
;          Description:    The 16 tap coeffs comprise the low–pass filter
;                          init. in sym_fir_init
```

Example 10–14. Low-Pass Symmetric FIR Filtering Using FIRS Instruction (Continued)

```
;    5.2.4 Outputs
;
;          Data structure: AR7
;          Data Format:    16-bit output buffer pointer
;          Modified:       Yes
;          Description:    either point to PING/PONG buffer
;
;    5.2.5 Global
;          NONE
;
;    5.2.6 Special considerations for data structure
;          –
;    5.2.7 Entry and Exit conditions
;
;     |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN
;     |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |  |
;in   |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |U  |U  |U  | U |UM|NU|NU|NU |NU|NU
;     |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |  |
;out  |U | 1 | 1 |NU | 1  |NU |UM |NU |NU |NU |UM |UM |NU |NU |UM|UM|NU|UM |NU|NU
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.2.8 Execution
;          Execution time: ?cycles
;          Call rate:    not applicable for this application
;
;===========================================================================
;HeaderEnd
;    5.2.9 Code
          .asg       AR6,INBUF_P
          .asg       AR7,OUTBUF_P
          .asg       AR4,SYMFIR_DATX_P
          .asg       AR5,SYMFIR_DATY_P
          .sect      ”sym_fir”
sym_fir_task:
   STM   #K_FRAME_SIZE–1,BRC
   RPTBD sym_fir_filter_loop–1
   STM   #K_FIR_BFFR/2,BK
   LD    *INBUF_P+, B
symmetric_fir:
   MVDD  *SYMFIR_DATX_P,*SYMFIR_DATY_P+0%      ; move X(–N/2) to X(–N)
   STL   B,*SYMFIR_DATX_P                      ; replace oldest sample with newest
                                               ; sample
   ADD   *SYMFIR_DATX_P+0%,*SYMFIR_DATY_P+0%,A ; add X(0)+X(–N/2–1)
   RPTZ  B,#(K_FIR_BFFR/2–1)
   FIRS  *SYMFIR_DATX_P+0%,*SYMFIR_DATY_P+0%,FIR_COFF
   MAR   *+SYMFIR_DATX_P(2)%                   ; to load the next newest sample
   MAR   *SYMFIR_DATY_P+%                      ; position for the X(–N/2) sample
   STH   B, *OUTBUF_P+
sym_fir_filter_loop
   RET
   .end
```

Example 10–15. Low-Pass Biquad IIR Filter

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    iir.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR:    Padma P. Mallela
;
;                  Application Specific Products
;                  Data Communication System Development
;                  12203 SW Freeway, MS 701
;                  Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /         AUTHORS                COMMENT
;          1.0     July–26–96  /        P.Mallela            original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains two subroutines:
;        1) iir_init
;        2) iir_task
;    1.3 Specification/Design Reference (optional)
;        called by task.asm depending upon the task thru CALA
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s iir.asm
;
;    1.6 Notes and Special Considerations
;
;}
```

Example 10–15. Low-Pass Biquad IIR Filter (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
        .mmregs
        .include   ”main.inc”
;    3.2 External Data
        .ref       d_iir_y
        .ref       d_iir_d
        .ref       iir_table_start,iir_coff_table
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
        .def    iir_init
        .def    iir_task
;}
;    5. SUBROUTINE CODE
;       HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 iir_init
;
;    5.1.1 Functional Description
;          A) This routine initializes buffers both for data and coeffs.
;-------------------------------------------------------------------------------
;
;    5.1.2 Activation
;          Activation example:
;          CALL    iir_init
;          Reentrancy:     No
;          Recursive :     No
```

Example 10–15. Low-Pass Biquad IIR Filter (Continued)

```
;
;    5.1.3 Inputs
;          NONE
;    5.1.4 Outputs
;          NONE
;    5.1.5 Global
;          NONE
;    5.1.6 Special considerations for data structure
;          –
;    5.1.7 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |UM|NU|NU|NU |NU|NU
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |
;out|U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |UM |UM |NU |NU |UM|NU|NU|UM |UM|NU
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.1.8 Execution
;          Execution time: ?cycles
;          Call rate:      not applicable for this application
;
;===========================================================================
;HeaderEnd
;    5.1.9 Code
    .asg  AR5,IIR_DATA_P                        ; data samples pointer
    .asg  AR4,IIR_COFF_P                        ; IIR filter coffs pointer
    .sect ”iir”
iir_init:
   STM    #iir_coff_table,IIR_COFF_P
   RPT    #K_IIR_SIZE–1                         ; move IIR coeffs from program
   MVPD   #iir_table_start,*IIR_COFF_P+         ; to data
   LD     #IIR_DP,DP
   STM    #d_iir_d,IIR_DATA_P                   ;AR5:d(n),d(n–1),d(n–2)
   RPTZ   A,#5                                  ;initial d(n),d(n–1),d(n–2)=0
   STL    A,*IIR_DATA_P+
   RET
;    5. SUBROUTINE CODE
;       HeaderBegin
;===========================================================================
;
;---------------------------------------------------------------------------
; 5.2 iir_task
;
; 5.2.1 Functional Description
;
;       This subroutine performs IIR filtering using biquad sections
;       IIR Low pass filter design
;       Filter type : Elliptic Filter
;       Filter order : 4 order (cascade: 2nd order + 2nd order)
;       cut freq. of pass band : 200 Hz
;       cut freq. of stop band : 500 Hz
```

Example 10–15. Low-Pass Biquad IIR Filter (Continued)

```
;                            B0
;        ... -----> + ------> d(n) ---- x -> + -----....
;                |           |    |
;                |    A1     |        B1    |
;                + <- x -- d(n-1) -- x -> +
;                |           |    |
;                |    A2     |        B2    |
;                + <- x -- d(n-2) -- x -> +
;
;                    second order IIR
;-------------------------------------------------------------------------------
;
;    5.2.2 Activation
;          Activation example:
;          CALL    iir_task
;          Reentrancy:    No
;          Recursive :    No
;
;    5.2.3 Inputs
;
;          Data structure:    AR6
;          Data Format:       16-bit input buffer pointer
;          Modified:          Yes
;          Description:       either point to PING/PONG buffer
;
;    5.2.4 Outputs
;
;          Data structure:    AR7
;          Data Format:       16-bit output buffer pointer
;          Modified:          Yes
;          Description:       either point to PING/PONG buffer
;
;    5.2.5 Global
;          Data structure:    AR1
;          Data Format:       16-bit index counter
;          Modified:          Yes
;          Description:       checks if 256 samples are processed
;
;          Data structure:    AR5
;          Data Format:       16-bit data buffer pointer
;          Modified:          Yes
;          Description:       inlcudes both feed forward and feedback paths
;
;          Data structure:    AR4
;          Data Format:       16-bit coefficient buffer pointer
;          Modified:          Yes
;          Description:       contains 2 biquad sections
;
;          Data structure:    d_iir_y
;          Data Format:       16-bit variable
;          Modified:          Yes
;          Description:       holds the output of the 2 biquad sections
```

Example 10–15. Low-Pass Biquad IIR Filter (Continued)

```
;    5.2.6 Special considerations for data structure
;          –
;    5.2.7 Entry and Exit conditions
;
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |U  |U  |UM|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |NU |UM |NU |NU |UM |UM |UM |UM |UM|NU|NU|UM |UM|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.2.8 Execution
;          Execution time: ?cycles
;          Call rate:    not applicable for this application
;
;===============================================================================
;HeaderEnd
;    5.2.9 Code
   .asg         AR5,IIR_DATA_P          ; data samples pointer
   .asg         AR4,IIR_COFF_P          ; IIR filter coffs pointer
   .asg         AR6,INBUF_P
   .asg         AR7,OUTBUF_P
   .asg         AR1,IIR_INDEX_P
   .sect        ”iir”
iir_task:
   STM    #K_FRAME_SIZE–1,BRC           ; Perform filtering for 256 samples
   RPTB   iir_filter_loop–1
   LD     *INBUF_P+,8,A                 ; load the input value
iir_filter:
   STM    #d_iir_d+5,IIR_DATA_P         ;AR5:d(n),d(n–1),d(n–2)
   STM    #iir_coff_table,IIR_COFF_P    ;AR4:coeff of IIR filter A2,A1,B2,B1,B0
   STM    #K_BIQUAD–1,IIR_INDEX_P
feedback_path:
   MAC    *IIR_COFF_P+,*IIR_DATA_P–,A   ;input+d(n–2)*A2
   MAC    *IIR_COFF_P,*IIR_DATA_P,A     ;input+d(n–2)*A2+d(n–1)*A1/2
   MAC    *IIR_COFF_P+,*IIR_DATA_P–,A   ; A = A+d(n–1)*A1/2
   STH    A,*IIR_DATA_P+                ;d(n) = input+d(n–2)*A2+d(n–1)*A1
   MAR    *IIR_DATA_P+
* Forward path
   MPY    *IIR_COFF_P+,*IIR_DATA_P–,A   ;d(n–2)*B2
   MAC    *IIR_COFF_P+,*IIR_DATA_P,A    ;d(n–2)*B2+d(n–1)*B1
   DELAY  *IIR_DATA_P–                  ;d(n–2)=d(n–1)
eloop:
   BANZD  feedback_path, *IIR_INDEX_P–
   MAC    *IIR_COFF_P+,*IIR_DATA_P,A    ;d(n–2)*B2+d(n–1)*B1+d(n)*B0
   DELAY  *IIR_DATA_P–                  ;d(n–1)=d(n)
   STH    A,d_iir_y                     ;output=d(n–2)*B2+d(n–1)*B1+d(n)*B0
   LD     d_iir_y,2,A                   ; scale the output
   STL    A, *OUTBUF_P+                 ; replace the oldest bffr value
iir_filter_loop
   RET
   .end
```

Example 10–16. Adaptive Filtering Using LMS Instruction

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
;
; Archives:    PVCS
; Filename:    adapt.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR     Padma P. Mallela
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /    AUTHORS           COMMENT
;          1.0     July–24–96  /   P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains two subroutines:
;        1) adapt_init
;        2) adapt_task
;    1.3 Specification/Design Reference (optional)
;        called by task.asm depending upon the task
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s adapt.asm
;
```

Example 10–16. Adaptive Filtering Using LMS Instruction (Continued)

```
;    1.6 Notes and Special Considerations
;        –
;}
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include    ”main.inc”
;    3.2 External Data
;        .ref        ADAPT_DP
         .ref        d_mu,d_error,d_primary,d_output,d_mu,d_mu_e,d_new_x
         .ref        scoff,hcoff,wcoff
         .ref        xh,xw,d_adapt_count
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
        .def        adapt_init,adapt_task
;}
;    5. SUBROUTINE CODE
;    HeaderBegin
;===============================================================================
;
; ------------------------------------------------------------------------------
;    5.1 adapt_init
;
;    5.1.1 Functional Description
;
;        This subroutine moves filter coeffcients from program to data space.
;        Initializes the adaptive coefficients, buffers,vars,and sets the circular
;        buffer address for processing.
;-------------------------------------------------------------------------------
```

Example 10–16. Adaptive Filtering Using LMS Instruction (Continued)

```
;
;    5.1.2 Activation
;          Activation example:
;          CALL    adapt_init
;          Reentrancy:     No
;          Recursive :     No
;
;    5.1.3 Inputs
;          NONE
;
;    5.1.4 Outputs
;          NONE
;
;    5.1.5 Global
;          Data structure: AR0
;          Data Format:    16–bit index pointer
;          Modified:       No
;          Description:    uses in circular addressing mode for indexing
;
;          Data structure: AR1
;          Data Format:    16–bit pointer
;          Modified:       Yes
;          Description:    used in initializing buffers and vars
;
;          Data structure: AR3
;          Data Format:    16–bit x(n) data buffer pointer for H(z)
;          Modified:       Yes
;          Description:    initializes the pointer
;
;          Data structure: AR5
;          Data Format:    16–bit x(n) data buffer pointer for W(z)
;          Modified:       Yes
;          Description:    initializes the pointer
;
;    5.1.6 Special considerations for data structure
;          –
;    5.1.7 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |UM|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |UM |UM |NU |UM |NU |UM |NU |NU |UM|UM|NU|NU |NU|NU |
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.1.8 Execution
;          Execution time: ?cycles
;          Call rate:      not applicable for this application
;
;=============================================================================
;HeaderEnd
```

Example 10–16. Adaptive Filtering Using LMS Instruction (Continued)

```
;    5.1.9 Code
       .asg   AR0,INDEX_P
       .asg   AR1,INIT_P                      ; initialize buffer pointer
       .asg   AR3,XH_DATA_P                   ; data coeff buffer pointer
       .asg   AR5,XW_DATA_P                   ; data coeff buffer pointer
                                              ; for cal.y output
       .sect  ”filter”
adapt_init:
;   initialize input data location, input to hybrid, with Zero.
       STM    #xh,INIT_P
       RPTZ   A,#H_FILT_SIZE–1
       STL    A,*INIT_P+
;   initialize input data location, input to adaptive filter, with Zero.
       STM    #xw,INIT_P
       RPTZ   A,#ADPT_FILT_SIZE–1
       STL    A,*INIT_P+
;   initialize adaptive coefficient with Zero.
       STM    #wcoff,INIT_P
       RPTZ   A,#ADPT_FILT_SIZE–1
       STL    A,*INIT_P+
;   initialize temporary storage loacations with zero
       STM    #d_primary,INIT_P
       RPTZ   A,#6
       STL    A,*INIT_P+
;   copy system coefficient into RAM location, Rverse order
       STM    #hcoff,INIT_P
       RPT    #H_FILT_SIZE–1
       MVPD   #scoff,*INIT_P+
;      LD     #ADAPT_DP,DP                    ;set DP now and not worry about it
       ST     #K_mu,d_mu
       STM    #1,INDEX_P                      ; increment value to be used by
                                              ; dual address
;   associate auxilary registers for circular computation
       STM    #xh+H_FILT_SIZE–1,XH_DATA_P     ; last input of hybrid buffer
       RETD
       STM    #xw+ADPT_FILT_SIZE–1,XW_DATA_P  ;last element of input buffer
;    5. SUBROUTINE CODE
;       HeaderBegin
 ;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.2    adapt_task
;
;    5.2.1 Functional Description
;
;      This subroutine performs the adaptive filtering.The newest sample is
;      stored in a seperate location since filtering and adaptation are performed
;      at the same time. Otherwise the oldest sample is over written before
;      up dating the w(N–1) coefficient.
;
;      d_primary = xh *hcoff
;      d_output = xw *wcoff
```

Example 10–16. Adaptive Filtering Using LMS Instruction (Continued)

```
;      LMS algorithm:
;      w(i+1) = w(i)+d*mu_error*xw(n–i) for i = 0,1,...127 and n = 0,1,.....
;      This program can run in two steps
;      1. Initial stepsize, d_mu = 0x0. At this point, the system is not
;      identified since the coefficients are not adapted and the error
;      signal e (n) is d (n). This is the default mode
;      2. At the EVM debugger command window change the step size
;      d_mu – 0x000, with the command e * d_mu = 0x1000
;      This changes the stepsize. The error signal e(n) in this case
;      is approximately 0 (theoretically) and the system is identified.
;-------------------------------------------------------------------------------;
;    5.2.2 Activation
;       Activation example:
;       CALL    adapt_task
;       Reentrancy:     No
;       Recursive :     No
;
;    5.2.3 Inputs
;
;        Data structure: AR3
;        Data Format:    16–bit x(n) data buffer pointer for H(Z)
;        Modified:       Yes
;        Description:    uses circular buffer addressing mode of size 128
;
;        Data structure: AR5
;        Data Format:    16–bit x(n) data buffer pointer for W(z)
;        Modified:       Yes
;        Description:    uses circular buffer addressing mode of size 128
;
;        Data structure: AR6
;        Data Format:    16–bit input buffer pointer
;        Modified:       Yes
;        Description:    either point to PING/PONG buffer
;
;    5.2.4 Outputs
;
;        Data structure: AR7
;        Data Format:    16–bit output buffer pointer
;        Modified:       Yes
;        Description:    either point to PING/PONG buffer
;
;    5.2.5 Global
;
;        Data structure: AR2
;        Data Format:    16–bit H(z) coeff buffer pointer
;        Modified:       Yes
;        Description:    uses circular buffer addressing mode of size 128
;
;        Data structure: AR4
;        Data Format:    16–bit W(z) coeff buffer pointer
;        Modified:       Yes
;        Description:    uses circular buffer addressing mode of size 128
;
```

Example 10–16. Adaptive Filtering Using LMS Instruction (Continued)

```
;        Data structure: d_adapt_count
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    counter to check for processing 256 samples
;
;        Data structure: d_new_x
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    holds the newest sample
;
;        Data structure: d_primary
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    d_primary = xh * hcoeff
;
;        Data structure: d_output
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    d_output = xw * wcoff
;
;        Data structure: d_error
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    d_error = d_primary–d_output
;
;        Data structure: d_mu_e
;        Data Format:    16–bit variable
;        Modified:       Yes
;        Description:    d_mu_e = mu*d_error
;
;    5.2.6 Special considerations for data structure
;          –
;    5.2.7 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |U  |NU |NU |U  |NU | U |U  | U |UM|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |U  |NU |UM |UM |UM |UM |UM |UM |UM|UM|UM|UM |UM|NU |
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.2.8 Execution
;          Execution time: ?cycles
;          Call rate:      not applicable for this application
;
;===========================================================================
;    HeaderEnd
```

Example 10–16. Adaptive Filtering Using LMS Instruction (Continued)

```
; 5.2.9 Code
        .asg    AR2,H_COFF_P                ; H(Z) coeff buffer pointer
        .asg    AR3,XH_DATA_P               ; data coeff buffer pointer
        .asg    AR6,INBUF_P                 ; input buffer address pointer
        .asg    AR7,OUTBUF_P                ; output buffer address pointer
                                            ; for cal. primary input
        .asg    AR4,W_COFF_P                ; W(z) coeff buffer pointer
        .asg    AR5,XW_DATA_P               ; data coeff buffer pointer
        .sect   ”filter”
adapt_task:
        STM     #H_FILT_SIZE,BK             ; first circular buffer size
        STM     #hcoff,H_COFF_P             ; H_COFF_P --> last of sys coeff
        ADDM    #1,d_adapt_count
        LD      *INBUF_P+, A                ; load the input sample
        STM     #wcoff,W_COFF_P             ; reset coeff buffer
        STL     A,d_new_x                   ; read in new data
        LD      d_new_x,A                   ;
        STL     A,*XH_DATA_P+0%             ; store in the buffer
        RPTZ    A,#H_FILT_SIZE–1            ; Repeat 128 times
        MAC     *H_COFF_P+0%,*XH_DATA_P+0%,A ; mult & acc:a = a + (h * x)
        STH     A,d_primary                 ; primary signal
;       start simultaneous filtering and updating the adaptive filter here.
        LD      d_mu_e,T                    ; T = step_size*error
        SUB     B,B                         ; zero acc B
        STM     #(ADPT_FILT_SIZE–2),BRC     ; set block repeat counter
        RPTBD   lms_end–1
        MPY     *XW_DATA_P+0%, A            ; error * oldest sample
        LMS     *W_COFF_P, *XW_DATA_P       ; B = filtered output (y)
                                            ; Update filter coeff
        ST      A, *W_COFF_P+               ; save updated filter coeff
        ||      MPY*XW_DATA_P+0%,A          ; error *x[n–(N–1)]
        LMS     *W_COFF_P, *XW_DATA_P        ; B = accum filtered output y
                                            ; Update filter coeff
lms_end
        STH     A, *W_COFF_P                ; final coeff
        MPY     *XW_DATA_P,A                ; x(0)*h(0)
        MVKD    #d_new_x,*XW_DATA_P         ; store the newest sample
        LMS     *W_COFF_P,*XW_DATA_P+0%
        STH     B, d_output                 ; store the fitlered output
        LD      d_primary,A
        SUB     d_output,A
        STL     A, d_error                  ; store the residual error signal
        LD      d_mu,T
        MPY     d_error,A                   ; A=u*e
        STH     A,d_mu_e                    ; save the error *step_size
        LD      d_error,A                   ; residual error signal
        STL     A, *OUTBUF_P+
        LD      #K_FRAME_SIZE,A             ; check if a frame of samples
        SUB     d_adapt_count,A             ; have been processed
        BC      adapt_task,AGT
        RETD
        ST      #K_0,d_adapt_count          ; restore the count
       .end
```

Example 10–16. Adaptive Filtering Using LMS Instruction (Continued)

```
* This is an input file used by the adaptive filter program.
* The transfer function is the system to be identifued by the adaptive filter
    .word    0FFFDh
    .word    24h
    .word    6h
    .word    0FFFDh
    .word    3h
    .word    3h
    .word    0FFE9h
    .word    7h
    .word    12h
    .word    1Ch
    .word    0FFF3h
    .word    0FFE8h
    .word    0Ch
    .word    3h
    .word    1Eh
    .word    1Ah
    .word    22h
    .word    0FFF5h
    .word    0FFE5h
    .word    0FFF1h
    .word    0FFC5h
    .word    0Ch
    .word    0FFE8h
    .word    37h
    .word    0FFE4h
    .word    0FFCAh
    .word    1Ch
    .word    0FFFDh
    .word    21h
    .word    0FFF7h
    .word    2Eh
    .word    28h
    .word    0FFC6h
    .word    53h
    .word    0FFB0h
    .word    55h
    .word    0FF36h
    .word    5h
    .word    0FFCFh
    .word    0FF99h
    .word    64h
    .word    41h
    .word    0FFF1h
    .word    0FFDFh
    .word    0D1h
    .word    6Ch
    .word    57h
    .word    36h
    .word    0A0h
    .word    0FEE3h
    .word    6h
```

Example 10–16. Adaptive Filtering Using LMS Instruction (Continued)

```
    .word   0FEC5h
    .word   0ABh
    .word   185h
    .word   0FFF6h
    .word   93h
    .word   1Fh
    .word   10Eh
    .word   59h
    .word   0FEF0h
    .word   96h
    .word   0FFBFh
    .word   0FF47h
    .word   0FF76h
    .word   0FF0Bh
    .word   0FFAFh
    .word   14Bh
    .word   0FF3Bh
    .word   132h
    .word   289h
    .word   8Dh
    .word   0FE1Dh
    .word   0FE1Bh
    .word   0D4h
    .word   0FF69h
    .word   14Fh
    .word   2AAh
    .word   0FD43h
    .word   0F98Fh
    .word   451h
    .word   13Ch
    .word   0FEF7h
    .word   0FE36h
    .word   80h
    .word   0FFBBh
    .word   0FC8Eh
    .word   10Eh
    .word   37Dh
    .word   6FAh
    .word   1h
    .word   0FD89h
    .word   198h
    .word   0FE4Ch
    .word   0FE78h
    .word   0F215h
    .word   479h
    .word   749h
    .word   289h
    .word   0F667h
    .word   304h
    .word   5F8h
    .word   34Fh
    .word   47Bh
    .word   0FF7Fh
```

Example 10–16. Adaptive Filtering Using LMS Instruction (Continued)

```
    .word   85Bh
    .word   0F837h
    .word   0F77Eh
    .word   0FF80h
    .word   0B9Bh
    .word   0F03Ah
    .word   0EE66h
    .word   0FE28h
    .word   0FAD0h
    .word   8C3h
    .word   0F5D6h
    .word   14DCh
    .word   0F3A7h
    .word   0E542h
    .word   10F2h
    .word   566h
    .word   26AAh
    .word   15Ah
    .word   2853h
    .word   0EE95h
    .word   93Dh
    .word   20Dh
    .word   1230h
    .word   238Ah
```

Example 10–17. 256-Point Real FFT Initialization

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:    PVCS
; Filename:    rfft.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR     Simon Lau and Nathan Baltz
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /      AUTHORS              COMMENT
;          1.0     July–17–96  /   Simon & Nathan      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains core routine:
;        rfft
;
;    1.3 Specification/Design Reference (optional)
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s rfft.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–17. 256-Point Real FFT Initialization (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include        ”main.inc”
         .include        ”init_54x.inc”
;    3.2 External Data
         .ref            bit_rev, fft, unpack
         .ref            power
         .ref            sine,cosine
         .ref            sine_table,cos_table
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def            rfft_task
;}
;    5. SUBROUTINE CODE
;    HeaderBegin
;============================================================================
;
;----------------------------------------------------------------------------
;    5.1 rfft
;
;    5.2 Functional Description
;        The following code implements a Radix–2, DIT, 2N–point Real FFT for the
;        TMS320C54x.  This main program makes four function calls, each
;        corresponds to a different phase of the algorithm. For more details about
;        how each phase is implemented, see bit_rev.asm, fft.asm, unpack.asm, and
;        power.asm assembly files.
;----------------------------------------------------------------------------
;
```

Example 10–17. 256-Point Real FFT Initialization (Continued)

```
;    5.3 Activation
;        Activation example:
;        CALL    rfft
;        Reentrancy:     No
;        Recursive :     No
;    5.4 Inputs
;        NONE
;    5.5 Outputs
;        NONE
;
;    5.6 Global
;
;        Data structure: AR1
;        Data Format:    16–bit pointer
;        Modified:       No
;        Description:    used for moving the twiddle tables from
;                        program to data
;
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN
;    |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU
;    |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |
;out|U | 1 | 1 |NU | 1  |U  |NU |UM |NU |NU |NU |NU |NU |NU |NU|NU|NU|UM |NU|NU
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;          Execution time: ?cycles
;          Call rate:      not applicable for this application
;
;===========================================================================
;HeaderEnd
;    5.10 Code
          .asg    AR1,FFT_TWID_P
          .sect   ”rfft_prg”
rfft_task:
          STM     #sine,FFT_TWID_P
          RPT     #K_FFT_SIZE–1               ; move FIR coeffs from program
          MVPD    #sine_table,*FFT_TWID_P+    ; to data
          STM     #cosine,FFT_TWID_P
          RPT     #K_FFT_SIZE–1               ; move FIR coeffs from program
          MVPD    #cos_table,*FFT_TWID_P+     ; to data
          CALL    bit_rev
          CALL    fft
          CALL    unpack
          CALLD   power
          STM     #K_ST1,ST1                  ; restore the original contents of
                                              ; ST1 since ASM field has changed
          RET                                 ; return to main program
          .end
```

Example 10–18. Bit Reversal Routine

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP ;
;
; Archives:    PVCS
; Filename:    bit_rev.asm
; Version:     1.0
; Status :     draft         ( )
;              proposal      (X)
;              accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR     Simon Lau and Nathan Baltz
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /      AUTHORS             COMMENT
;          1.0     July–17–96  /   Simon & Nathan      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains one subroutine:
;        bit_rev
;
;    1.3 Specification/Design Reference (optional)
;        called by rfft.asm depending upon the task thru CALA
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s bit_rev.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–18. Bit Reversal Routine (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include    ”main.inc”
;    3.2 External Data
         .ref        d_input_addr, fft_data
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def        bit_rev
;}
;    5. SUBROUTINE CODE
;    HeaderBegin
;*****************************************************************************
;
;-----------------------------------------------------------------------------
;    5.1 bit_rev
;
;    5.2 Functional Description
;        This function is called from the main module of the ’C54x Real FFT code.
;        It reorders the original 2N–point real input sequence by using
;        bit–reversed addressing.  This new sequence is stored into the data
;        processing buffer of size 2N, where FFT will be performed in–place
;        during Phase Two.
;
;-----------------------------------------------------------------------------
;
```

Example 10–18. Bit Reversal Routine (Continued)

```
;    5.3 Activation
;        Activation example:
;        CALL    bit_rev
;        Reentrancy:     No
;        Recursive :     No
;
;    5.4 Inputs
;        NONE
;    5.5 Outputs
;        NONE
;
;    5.6 Global
;
;        Data structure: AR0
;        Data Format:    16–bit index pointer
;        Modified:       No
;        Description:    used for bit reversed addressing
;
;        Data structure: AR2
;        Data Format:    16–bit pointer
;        Modified:       Yes
;        Description:    pointer to processed data in bit–reversed order
;
;        Data structure: AR3
;        Data Format:    16–bit pointer
;        Modified:       Yes
;        Description:    pointer to original input data in natural order
;
;        Data structure: AR7
;        Data Format:    16–bit pointer
;        Modified:       Yes
;        Description:    starting addressing of data processing buffer
;
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |UM |NU |UM |UM |NU |NU |NU |UM |NU|NU|NU|UM |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
;
;===========================================================================
;HeaderEnd
```

Example 10–18. Bit Reversal Routine (Continued)

```
;    5.10 Code
      .asg    AR2,REORDERED_DATA
      .asg    AR3,ORIGINAL_INPUT
      .asg    AR7,DATA_PROC_BUF
      .sect   ”rfft_prg”
bit_rev:
      SSBX    FRCT                                ; fractional mode is on
      MVDK    d_input_addr,ORIGINAL_INPUT         ; AR3 –> 1 st original input
      STM     #fft_data,DATA_PROC_BUF             ; AR7 –> data processing buffer
      MVMM    DATA_PROC_BUF,REORDERED_DATA        ; AR2 –> 1st bit–reversed data
      STM     #K_FFT_SIZE–1,BRC
      RPTBD   bit_rev_end–1
      STM     #K_FFT_SIZE,AR0                     ; AR0 = 1/2 size of circ buffer
      MVDD    *ORIGINAL_INPUT+,*REORDERED_DATA+
      MVDD    *ORIGINAL_INPUT–,*REORDERED_DATA+
      MAR     *ORIGINAL_INPUT+0B
bit_rev_end:
      RET                                         ; return to Real FFT main module
      end
```

Example 10–19. 256-Point Real FFT Routine

```
; TEXAS INSTRUMENTS INCORPORATED
; DSP Data Communication System Development / ASP
;
; Archives:   PVCS
; Filename:   fft.asm
; Version:    1.0
; Status :    draft         ( )
;             proposal      (X)
;             accepted      ( ) dd–mm–yy/?acceptor.
;
; AUTHOR    Simon Lau and Nathan Baltz
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
; IPR statements description (can be collected).
;}
;(C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
; Change history:
;
;        VERSION      DATE     /      AUTHORS             COMMENT
;          1.0     July–17–96  /   Simon & Nathan      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains one subroutine:
;        fft
;
;    1.3 Specification/Design Reference (optional)
;        called by rfft.asm depending upon the task thru CALA
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s fft.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–19. 256-Point Real FFT Routine (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
        .mmregs
        .include    ”main.inc”
;    3.2 External Data
        .ref        fft_data, d_grps_cnt, d_twid_idx, d_data_idx, sine, cosine
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
        .def        fft
;}
;    5. SUBROUTINE CODE
;    HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 fft
;
;    5.2 Functional Description
;        PHASE TWO   (LogN)–Stage Complex FFT
;        This function is called from main module of the ’C54x Real FFT code.
;        Here we assume that the original 2N–point real input sequence is al
;        ready packed into an N–point complex sequence and stored into the
;        data processing buffer in bit–reversed order (as done in Phase One).
;        Now we perform an in–place, N–point complex FFT on the data proces
;        sing buffer, dividing the outputs by 2 at the end of each stage to
;        prevent overflow. The resulting N–point complex sequence will be un–
;        packed into a 2N–point complex sequencein Phase Three & Four.
;-------------------------------------------------------------------------------
;
```

Example 10–19. 256-Point Real FFT Routine (Continued)

```
;    5.3 Activation
;        Activation example:
;        CALL    fft
;        Reentrancy:     No
;        Recursive :     No
;
;    5.4 Inputs
;        NONE
;    5.5 Outputs
;        NONE
;
;    5.6 Global
;
;        Data structure:    AR0
;        Data Format:       16–bit index pointer
;        Modified:          No
;        Description:       index to twiddle tables
;
;        Data structure:    AR1
;        Data Format:       16–bit counter
;        Modified:          No
;        Description:       group counter
;
;        Data structure:    AR2
;        Data Format:       16–bit pointer
;        Modified:          Yes
;        Description:       pointer to 1st butterfly data PR,PI
;
;        Data structure:    AR3
;        Data Format:       16–bit pointer
;        Modified:          Yes
;        Description:       pointer to 2nd butterfly data QR,QI
;
;        Data structure:   AR4
;        Data Format:      16–bit pointer
;        Modified:         Yes
;        Description:      pointer to cosine value WR
;
;        Data structure:   AR5
;        Data Format:      16–bit pointer
;        Modified:         Yes
;        Description:      pointer to cosine value WI
;
;        Data structure:   AR6
;        Data Format:      16–bit counter
;        Modified:         Yes
;        Description:      butterfly counter
;
;        Data structure:   AR7
;        Data Format:      16–bit pointer
;        Modified:         Yes
;        Description:      start address of data processing buffer
;
```

Example 10–19. 256-Point Real FFT Routine (Continued)

```
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  | 0 |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |–1 |UM |UM |UM |UM |UM |UM |UM |UM |UM|UM|UM|UM |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate: not applicable for this application
;
;===========================================================================
;    HeaderEnd
;    5.10 Code
          .asg     AR1,GROUP_COUNTER
          .asg     AR2,PX
          .asg     AR3,QX
          .asg     AR4,WR
          .asg     AR5,WI
          .asg     AR6,BUTTERFLY_COUNTER
          .asg     AR7,DATA_PROC_BUF              ; for Stages 1 & 2
          .asg     AR7,STAGE_COUNTER              ; for the remaining stages
          .sect    ”rfft_prg”
fft:
;   Stage 1 ------------------------------------------------------------------
          STM      #K_ZERO_BK,BK                 ; BK=0 so that *ARn+0% == *ARn+0
          LD       #–1,ASM                       ; outputs div by 2 at each stage
          MVMM     DATA_PROC_BUF,PX              ; PX –> PR
          LD       *PX,A                         ; A  :=  PR
          STM      #fft_data+K_DATA_IDX_1,QX     ; QX –> QR
          STM      #K_FFT_SIZE/2–1,BRC
          RPTBD    stage1end–1
          STM      #K_DATA_IDX_1+1,AR0
          SUB      *QX,16,A,B                    ; B  :=  PR–QR
          ADD      *QX,16,A                      ; A  :=  PR+QR
          STH      A,ASM,*PX+                    ; PR’:= (PR+QR)/2
          ST       B,*QX+                        ; QR’:= (PR–QR)/2
          ||LD     *PX,A                         ; A  :=  PI
          SUB      *QX,16,A,B                    ; B  :=  PI–QI
          ADD      *QX,16,A                      ; A  :=  PI+QI
          STH      A,ASM,*PX+0                   ; PI’:= (PI+QI)/2
          ST       B,*QX+0%                      ; QI’:= (PI–QI)/2
          ||LD     *PX,A                         ; A  :=  next PR
```

Example 10–19. 256-Point Real FFT Routine (Continued)

```
stage1end:
;   Stage 2 ------------------------------------------------------------------------
          MVMM      DATA_PROC_BUF,PX                ; PX -> PR
          STM       #fft_data+K_DATA_IDX_2,QX       ; QX -> QR
          STM       #K_FFT_SIZE/4-1,BRC
          LD        *PX,A                           ; A  :=  PR
          RPTBD     stage2end-1
          STM       #K_DATA_IDX_2+1,AR0
; 1st butterfly
          SUB       *QX,16,A,B                      ; B  :=  PR-QR
          ADD       *QX,16,A                        ; A  :=  PR+QR
          STH       A,ASM,*PX+                      ; PR':= (PR+QR)/2
          ST        B,*QX+                          ; QR':= (PR-QR)/2
          ||LD      *PX,A                           ; A  :=  PI
          SUB       *QX,16,A,B                      ; B  :=  PI-QI
          ADD       *QX,16,A                        ; A  :=  PI+QI
          STH       A,ASM,*PX+                      ; PI':= (PI+QI)/2
          STH       B,ASM,*QX+                      ; QI':= (PI-QI)/2
; 2nd butterfly
          MAR       *QX+
          ADD       *PX,*QX,A                       ; A  :=  PR+QI
          SUB       *PX,*QX-,B                      ; B  :=  PR-QI
          STH       A,ASM,*PX+                      ; PR':= (PR+QI)/2
          SUB       *PX,*QX,A                       ; A  :=  PI-QR
          ST        B,*QX                           ; QR':= (PR-QI)/2
          ||LD      *QX+,B                          ; B  :=     QR
          ST        A, *PX                          ; PI':= (PI-QR)/2
          ||ADD     *PX+0%,A                        ; A  :=  PI+QR
          ST        A,*QX+0%                        ; QI':= (PI+QR)/2
          ||LD      *PX,A                           ; A  :=  PR
stage2end:
; Stage 3 thru Stage logN-1 ------------------------------------------------------
       STM    #K_TWID_TBL_SIZE,BK                   ; BK = twiddle table size always
       ST     #K_TWID_IDX_3,d_twid_idx              ; init index of twiddle table
       STM    #K_TWID_IDX_3,AR0                     ; AR0 = index of twiddle table
       STM    #cosine,WR                            ; init WR pointer
       STM    #sine,WI                              ; init WI pointer
       STM    #K_LOGN-2-1,STAGE_COUNTER             ; init stage counter
       ST     #K_FFT_SIZE/8-1,d_grps_cnt            ; init group counter
       STM    #K_FLY_COUNT_3-1,BUTTERFLY_COUNTER    ; init butterfly counter
       ST     #K_DATA_IDX_3,d_data_idx              ; init index for input data
stage:
       STM    #fft_data,PX                          ; PX -> PR
       LD     d_data_idx, A
       ADD    *(PX),A
       STLM   A,QX                                  ; QX -> QR
       MVDK   d_grps_cnt,GROUP_COUNTER              ; AR1 contains group counter
```

Example 10–19. 256-Point Real FFT Routine (Continued)

```
group:
       MVMD   BUTTERFLY_COUNTER,BRC              ; # of butterflies in each grp
       RPTBD  butterflyend–1
       LD     *WR,T                              ; T  :=  WR
       MPY    *QX+,A                             ; A  :=  QR*WR  || QX–>QI
       MACR   *WI+0%,*QX–,A                      ; A  :=  QR*WR+QI*WI
                                                 ; || QX–>QR
       ADD    *PX,16,A,B                         ; B  := (QR*WR+QI*WI)+PR
       ST     B,*PX                              ; PR':=((QR*WR+QI*WI)+PR)/2
       ||SUB  *PX+,B                             ; B  :=  PR–(QR*WR+QI*WI)
                                                 ; || PX–>PI
       ST     B,*QX                              ; QR':= (PR–(QR*WR+QI*WI))/2
       ||MPY  *QX+,A                             ; A  :=  QR*WI [T=WI]
                                                 ; || QX–>QI
       MASR   *QX,*WR+0%,A                       ; A  :=  QR*WI–QI*WR
       ADD    *PX,16,A,B                         ; B  := (QR*WI–QI*WR)+PI
       ST     B,*QX+                             ; QI':=((QR*WI–QI*WR)+PI)/2
                                                 ; || QX–>QR
       ||SUB  *PX,B                              ; B  :=  PI–(QR*WI–QI*WR)
       LD     *WR,T                              ; T  :=  WR
       ST     B,*PX+                             ; PI':= (PI–(QR*WI–QI*WR))/2
                                                 ; || PX–>PR
       ||MPY  *QX+,A                             ; A  :=  QR*WR  || QX–>QI
butterflyend:
; Update pointers for next group
       PSHM   AR0                                ; preserve AR0
       MVDK   d_data_idx,AR0
     MAR      *PX+0                              ; increment PX for next group
     MAR      *QX+0                              ; increment QX for next group
     BANZD    group,*GROUP_COUNTER–
     POPM     AR0                                ; restore AR0
     MAR      *QX–
; Update counters and indices for next stage
     LD       d_data_idx,A
     SUB      #1,A,B                             ; B = A–1
     STLM     B,BUTTERFLY_COUNTER                ; BUTTERFLY_COUNTER = #flies–1
     STL      A,1,d_data_idx                     ; double the index of data
     LD       d_grps_cnt,A
     STL      A,ASM,d_grps_cnt                   ; 1/2 the offset to next group
     LD       d_twid_idx,A
     STL      A,ASM,d_twid_idx                   ; 1/2 the index of twiddle table
     BANZ     D stage,*STAGE_COUNTER–
     MVDK     d_twid_idx,AR0                     ; AR0 = index of twiddle table
fft_end:
     RET                                        ; return to Real FFT main module
    .end
```

Example 10–20. Unpack 256-Point Real FFT Output

```
;    TEXAS INSTRUMENTS INCORPORATED
;    DSP Data Communication System Development / ASP
;    Archives:    PVCS
;    Filename:    unpack.asm
;    Version:     1.0
;    Status :     draft         ( )
;                 proposal      (X)
;                 accepted      ( ) dd–mm–yy/?acceptor.
;
;    AUTHOR    Simon Lau and Nathan Baltz
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
;    IPR statements description (can be collected).
;}
;    (C) Copyright 1996. Texas Instruments. All rights reserved.
;{
;    Change history:
;
;        VERSION      DATE     /      AUTHORS              COMMENT
;          1.0     July–17–96  /   Simon & Nathan      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains one subroutine:
;        unpack
;
;    1.3 Specification/Design Reference (optional)
;        called by rfft.asm depending upon the task thru CALA
;
;    1.4 Module Test Document Reference
;        Not done
;

;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s unpack.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–20. Unpack 256-Point Real FFT Output (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include        ”main.inc”
;    3.2 External Data
         .ref            fft_data,sine, cosine
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def        unpack
;}
;    5. SUBROUTINE CODE
;    HeaderBegin
;============================================================================
;
;    5.1  unpack
;
;    5.2 Functional Description
;
;        PHASE THREE & FOUR      Unpacking to 2N Outputs
;        This function is called from the main module of the ’C54x Real FFT
;        code. It first computes four intermediate sequences (RP,RM, IP, IM)
;        from the resulting complex sequence at the end of the previous phase.
;        Next, it uses the four intermediate sequences to form the FFT of the
;        original 2N–point real input. Again, the outputs are divided by 2 to
;        prevent overflow
;----------------------------------------------------------------------------
```

Example 10–20. Unpack 256-Point Real FFT Output (Continued)

```
;
;    5.3 Activation
;        Activation example:
;          CALL    unpack
;
;        Reentrancy:     No
;        Recursive :     No
;
;    5.4 Inputs
;        NONE
;    5.5 Outputs
;        NONE
;
;    5.6 Global
;
;    5.6.1 Phase Three Global
;
;          Data structure:    AR0
;          Data Format:       16–bit index pointer
;          Modified:          No
;          Description:       index to twiddle tables
;
;          Data structure:    AR2
;          Data Format:       16–bit pointer
;          Modified:          Yes
;          Description:       pointer to R[k], I[k], RP[k], IP[k]
;
;          Data structure:    AR3
;          Data Format:       16–bit pointer
;          Modified:          Yes
;          Description:       pointer to R[N–k], I[N–k], RP[N–k], IP[N–k]
;
;          Data structure:    AR6
;          Data Format:       16–bit pointer
;          Modified:          Yes
;          Description:       pointer to RM[k], IM[k]
;
;          Data structure:    AR7
;          Data Format:       16–bit pointer
;          Modified:          Yes
;          Description:       pointer to RM[n–k], IM[n–k]
;
;    5.6.2 Phase Four Global
;
;          Data structure: AR0
;          Data Format:       16–bit index pointer
;          Modified:          No
;          Description:       index to twiddle tables
;
;          Data structure: AR2
;          Data Format:       16–bit counter
;          Modified:          No
;          Description:       pointer to RP[k], IP[k], AR[k], AI[k], AR[0]
;
```

Example 10–20. Unpack 256-Point Real FFT Output (Continued)

```
;          Data structure: AR3
;          Data Format:    16–bit pointer
;          Modified:       Yes
;          Description:    pointer to RM[k], IM[k], AR[2N–k], AI[2N–k]
;
;          Data structure: AR4
;          Data Format:    16–bit pointer
;          Modified:       Yes
;          Description:    pointer to cos(k*pi/N), AI[0]
;
;          Data structure: AR5
;          Data Format:    16–bit pointer
;          Modified:       Yes
;          Description:    pointer to sin(k*pi/N), AR[N], AI[N]
;
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;    5.8.1 Phase Three Entry and Exit Conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |  |
;in |2 | 1 | 1 | 0 | 1  | 0 |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|0 |NU |NU|NU
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |  |
;out|2 | 1 | 1 | 0 | 1  |–1 |UM |NU |UM |UM |NU |NU |UM |UM |UM|UM|UM|UM |NU|NU
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.8.2 Phase Four Entry and Exit Conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |  |
;in |U | 1 | 1 | 0 | 1  |–1 |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |  |
;out|U | 1 | 1 | 0 | 1  |–1 |UM |NU |UM |UM |UM |UM |NU |NU |UM|UM|UM|UM |NU|NU
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
;
;===========================================================================
;         HeaderEnd
;    5.10  Code
    .sect        ”rfft_prg”
unpack:
; Compute intermediate values RP, RM, IP, IM
    .asg    AR2,XP_k
    .asg    AR3,XP_Nminusk
    .asg    AR6,XM_k
    .asg    AR7,XM_Nminusk
```

Example 10–20. Unpack 256-Point Real FFT Output (Continued)

```
     STM    #fft_data+2,XP_k                      ; AR2 –> R[k] (temp RP[k])
     STM    #fft_data+2*K_FFT_SIZE–2,XP_Nminusk   ; AR3 –> R[N–k] (temp
                                                          RP[N–k])
     STM    #fft_data+2*K_FFT_SIZE+3,XM_Nminusk   ; AR7 –> temp RM[N–k]
     STM    #fft_data+4*K_FFT_SIZE–1,XM_k         ; AR6 –> temp RM[k]
     STM    #–2+K_FFT_SIZE/2,BRC
     RPTBD  phase3end–1
     STM    #3,AR0
     ADD    *XP_k,*XP_Nminusk,A                   ; A := R[k]+R[N–k] =
                                                          2*RP[k]
     SUB    *XP_k,*XP_Nminusk,B                   ; B := R[k]–R[N–k] =
                                                          2*RM[k]
     STH    A,ASM,*XP_k+                          ; store RP[k] at AR[k]
     STH    A,ASM,*XP_Nminusk+                    ; store RP[N–k]=RP[k] at
                                                          AR[N–k]
     STH    B,ASM,*XM_k–                          ; store RM[k] at AI[2N–k]
     NEG    B                                     ; B := R[N–k]–R[k] =
                                                          2*RM[N–k]
     STH    B,ASM,*XM_Nminusk–                    ; store RM[N–k] at AI[N+k]
     ADD    *XP_k,*XP_Nminusk,A                   ; A := I[k]+I[N–k] =
                                                          2*IP[k]
     SUB    *XP_k,*XP_Nminusk,B                   ; B := I[k]–I[N–k] =
                                                          2*IM[k]
     STH    A,ASM,*XP_k+                          ; store IP[k] at AI[k]
     STH    A,ASM,*XP_Nminusk–0                   ; store IP[N–k]=IP[k] at
                                                          AI[N–k]
     STH    B,ASM,*XM_k–                          ; store IM[k] at AR[2N–k]
     NEG    B                                     ; B := I[N–k]–I[k] =
                                                          2*IM[N–k]
     STH    B,ASM,*XM_Nminusk+0                   ; store IM[N–k] at AR[N+k]
phase3end:
     ST     #0,*XM_k–                             ; RM[N/2]=0
     ST     #0,*XM_k                              ; IM[N/2]=0
; Compute AR[0],AI[0], AR[N], AI[N]
     .asg   AR2,AX_k
     .asg   AR4,IP_0
     .asg   AR5,AX_N
     STM    #fft_data,AX_k                        ; AR2 –> AR[0] (temp
                                                            RP[0])
     STM    #fft_data+1,IP_0                      ; AR4 –> AI[0] (temp
                                                            IP[0])
     STM    #fft_data+2*K_FFT_SIZE+1,AX_N         ; AR5 –> AI[N]
     ADD    *AX_k,*IP_0,A                         ; A := RP[0]+IP[0]
     SUB    *AX_k,*IP_0,B                         ; B := RP[0]–IP[0]
     STH    A,ASM,*AX_k+                          ; AR[0] = (RP[0]+IP[0])/2
     ST     #0,*AX_k                                  ; AI[0] = 0
     MVDD   *AX_k+,*AX_N–                             ; AI[N] = 0
     STH     B,ASM,*AX_N                              ; AR[N] = (RP[0]–IP[0])/2
; Compute final output values AR[k], AI[k]
     .asg     AR3,AX_2Nminusk
     .asg     AR4,COS
     .asg     AR5,SIN
```

Example 10–20. Unpack 256-Point Real FFT Output (Continued)

```
     STM      #fft_data+4*K_FFT_SIZE–1,AX_2Nminusk    ; AR3 –> AI[2N–1]
                                                          (temp RM[1])
     STM      #cosine+K_TWID_TBL_SIZE/K_FFT_SIZE,COS  ; AR4 –> cos(k*pi/N)
     STM      #sine+K_TWID_TBL_SIZE/K_FFT_SIZE,SIN    ; AR5 –> sin(k*pi/N)
     STM      #K_FFT_SIZE–2,BRC
     RPTBD    phase4end–1
     STM      #K_TWID_TBL_SIZE/K_FFT_SIZE,AR0         ; index of twiddle
                                                          tables
     LD       *AX_k+,16,A                             ; A := RP[k] ||
                                                          AR2–>IP[k]
     MACR     *COS,*AX_k,A                            ; A :=A+cos(k*pi/N)
                                                           *IP[k]
     MASR     *SIN,*AX_2Nminusk–,A                    ; A := A–sin(k*pi/N)
                                                          *RM[k]
                                                     ; || AR3–>IM[k]
     LD       *AX_2Nminusk+,16,B                      ; B := IM[k] ||
                                                          AR3–>RM[k]
     MASR     *SIN+0%,*AX_k–,B                        ; B := B–sin(k*pi/N)
                                                          *IP[k]
                                                     ; || AR2–>RP[k]
     MASR     *COS+0%,*AX_2Nminusk,B                  ; B := B–cos(k*pi/N)
                                                          *RM[k]
     STH      A,ASM,*AX_k+                            ; AR[k] = A/2
     STH      B,ASM,*AX_k+                            ; AI[k] = B/2
     NEG      B                                       ; B := –B
     STH      B,ASM,*AX_2Nminusk–                     ; AI[2N–k] = –AI[k]
                                                               = B/2
     STH      A,ASM,*AX_2Nminusk–                     ; AR[2N–k] = AR
                                                           [k] = A/2

phase4end:
     RET                                              ; returntoRealFFTmain module
     .end
```

Example 10–21. Compute the Power Spectrum of the Complex Output of the 256-Point Real FFT

```
;    TEXAS INSTRUMENTS INCORPORATED
;    DSP Data Communication System Development / ASP
;
;    Archives:    PVCS
;    Filename:    power.asm
;    Version:     1.0
;    Status :     draft         ( )
;                 proposal      (X)
;                 accepted      ( ) dd–mm–yy/?acceptor.
;
;    AUTHOR    Simon Lau and Nathan Baltz
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
;    IPR statements description (can be collected).
;}
;   (C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
;    Change history:
;
;        VERSION      DATE     /      AUTHORS              COMMENT
;          1.0     July–17–96  /   Simon & Nathan      original created
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains one subroutine:
;        power
;
;    1.3 Specification/Design Reference (optional)
;        called by rfft.asm depending upon the task thru CALA
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s power.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–21. Compute the Power Spectrum of the Complex Output of the 256-Point Real FFT (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include     ”main.inc”
;    3.2 External Data
         .ref        fft_data, d_output_addr
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def        power
;}
;    5. SUBROUTINE CODE
;    HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 power
;
;    5.2 Functional Description
;        PHASE FIVE  Power Spectrum
;        This function is called from the main module of the ’C54x Real FFT
;        code. It computes the power spectrum of the Real FFT output.
;-------------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
;        CALL    power
;        Reentrancy:     No
;        Recursive :     No
```

Example 10–21. Compute the Power Spectrum of the Complex Output of the 256-Point Real FFT (Continued)

```
;
;    5.4 Inputs
;        NONE
;    5.5 Outputs
;        NONE
;
;    5.6 Global
;        Data structure: AR2
;        Data Format:    16–bit pointer
;        Modified:       Yes
;        Description:    pointer to AR[k], AI[k]
;
;        Data structure: AR3
;        Data Format:    16–bit pointer
;        Modified:       Yes
;        Description:    pointer to output buffer
;
;    5.7 Special considerations for data structure
;        –
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |NU |NU|NU|NU|NU |NU|NU |
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;out|U | 1 | 1 |NU | 1  |NU |NU |NU |UM |UM |NU |NU |NU |NU |UM|NU|NU|UM |NU|NU |
;
; Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:    not applicable for this application
;
;==========================================================================
;    HeaderEnd
;    5.10 Code
         .asg      AR2,AX
         .asg      AR3,OUTPUT_BUF
         .sect     ”pwr_prog” power:
         MVDK      d_output_addr,OUTPUT_BUF      ; AR3 points to output buffer
         STM       #K_FFT_SIZE*2–1,BRC
         RPTBD     power_end–1
         STM       #fft_data,AX                  ; AR2 points to AR[0]
         SQUR      *AX+,A                        ; A := AR^2
         SQURA     *AX+,A                        ; A := AR^2 + AI^2
         STH       A,*OUTPUT_BUF+
   power_end:
         RET                                     ; return to main program
         .end
```

Example 10–22. Data Transfer from FIFO

```
;    TEXAS INSTRUMENTS INCORPORATED
;    DSP Data Communication System Development / ASP
;
;    Archives:    PVCS
;    Filename:    fifo.asm
;    Version:     1.0
;    Status :     draft         ( )
;                 proposal      (X)
;                 accepted      ( ) dd–mm–yy/?acceptor.
;
;    AUTHOR    Padma P. Mallela
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
;    IPR statements description (can be collected).
;}
;    (C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
;    Change history:
;
;        VERSION      DATE     /     AUTHORS           COMMENT
;          1.0     July–25–96  /    P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains one subroutines:
;        fifo_host_transfer
;    1.3 Specification/Design Reference (optional)
;        called by main.asm depending upon if K_HOST_FLAG is set
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s fifo.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–22. Data Transfer from FIFO (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include        ”target.inc”
;    3.2 External Data
         .ref            d_command_reg
         .ref            d_fifo_count
         .ref            d_command_value
         .ref            d_fifo_ptr
         .ref            d_output_addr
;    3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def            fifo_host_transfer
;}
;    5. SUBROUTINE CODE
;    HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 fifo_host_transfer
;
;    5.2 Functional Description
;        This routine transfers a FIFO(64) of data to host thru CH B.
;        In the process, after transferring data from DSP to FIFO sends a com–
;        mand to host thru CH A.The host acknowledges and sends a command to
;        target (DSP) thru CH A. The host transfer can be disabled by setting
;        the K_HOST_FLAG =0
;-------------------------------------------------------------------------------
;
```

Example 10–22. Data Transfer from FIFO (Continued)

```
;    5.3 Activation
;        Activation example:
;                   CALL    fifo_host_transfer
;
;        Reentrancy:     No
;        Recursive :     No
;
;    5.4 Inputs
;                   Data structure: d_output_addr
;                   Data Format:    16–bit variable
;                   Modified:       NO
;    Description:   holds the starting addr of either PING/PONG addr.
;
;                   Data structure: d_fifo_count
;                   Data Format:    16–bit var
;                   Modified:       Yes
;                   Description:    counter for # of transfers
;
;                   Data structure: d_fifo_ptr
;                   Data Format:    16–bit variable
;                   Modified:        Yes
;                   Description: holds the output bffr addr. and incremented by
;                   32 for every transfer
;
;    5.5 Outputs
;
;                   Data structure: AR7
;                   Data Format:    16–bit output buffer pointer
;                   Modified:       Yes
;                   Description:    either point to PING/PONG buffer
;
;    5.6 Global
;                   Data structure: d_command_reg
;                   Data Format:    16–bit variable
;                   Modified:       Yes
;                   Description:    command from host is read thru CH A
;                   Data structure: d_command_value
;                   Data Format:    16–bit variable
;                   Modified:       Yes
;                   Description:    holds the command value
;
;    5.7 Special considerations for data structure
;        –
;
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN|
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |   |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |U  |NU|NU|NU|NU |NU|NU |
;out|U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |U  |UM|NU|NU|NU |NU|NU |
;
;    Note : UM – Used & Modified, U – Used, NU – Not Used
;
```

Example 10–22. Data Transfer from FIFO (Continued)

```
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
;
;===========================================================================
;HeaderEnd
;    5.10 Code
;         fifo_host_transfer:
;         LD       #FIFO_DP,DP
          .if      K_HOST_FLAG =1
          PORTR    K_TRGCR_ADDR,d_command_reg    ; while (port14 & BXST)
          BITF     d_command_reg,K_BXST
          BC       fifo_discard,TC               ; FIFO discard
          MVDK     d_output_addr,OUTBUF_P        ; load PING/PONG bffr address
          RPT      #K_FIFO_SIZE-1                ; write first set of 64 data
                                                 ; to FIFO
          PORTW    *OUTBUF_P+,K_CHB              ; Fill FIFO
          ST       #K_FIFO_FULL,d_command_value
          PORTW    d_command_value, K_CHA        ; write command to comnd reg A
          ST       #1,d_fifo_count               ; start counting for tranfers
          MVKD    OUTBUF_P,d_fifo_ptr            ; save the fifo_ptr
fifo_discard
          .endif
          RET
          .end
*****************************************************************************
* This file includes the TCR register configuration of EVM
*****************************************************************************
K_AIC_RST           .set      0b << 15         ; if AICRST=0, aic is reset
K_USR_BOT           .set      000b << 12       ; User discrete output bits
                                               ; 0,1,2
K_RESRV             .set      0000b << 8       ; Reserved bits
K_USR_BIN           .set      00b << 6         ; User discrete input bits 0,1
K_RCV_BRST          .set      00b << 4         ; Channel B receive status regs
                                               ; buffer half or more
K_XMT_BXST          .set      11b << 2         ; Ch B trasnmit status register
                                               ; buffer half or more
K_RCV_ARST          .set      0b << 1          ; Ch A receive register
K_XMT_AXST          .set      0b << 1          ; Ch A transmit register
K_TCR_HIGH          .set      K_AIC_RST|K_USR_BOT|K_RESRV
K_TCR_LOW           .set      K_USR_BIN|K_RCV_BRST|K_XMT_BXST|K_RCV_ARST|K_XMT_AXST
K_TCR               .set      K_TCR_HIGH|K_TCR_LOW
*****************************************************************************
* this includes I/O address of CH_A, CH_B and different commands that's been
* passed between host and the target
*****************************************************************************
K_0                 .set      0h               ; constant 0
K_FIFO_FULL         .set      0xFF             ; Full FIFO command written by
                                               ; target
K_FIFO_EMPTY        .set      0xEE             ; Empty FIFO command
                                               ; written by host
K_AXST_CLEAR        .set      0xAE             ; Clear AXST empty command
                                               ; written by the target
```

Example 10–22. Data Transfer from FIFO (Continued)

```
K_HANDSHAKE_CMD     .set      0xAB            ; handshake CMD written by host
K_CHB               .set      12h             ; Use Channel B as I/O interface
                                              ; to 54x EVM for sending data
K_CHA               .set      10h             ; Use Channel A as I/O interface
                                              ; to 54x EVM for send command
                                              ; to host
K_TRGCR_ADDR        .set      14h             ; Target status control register
                                              ; I/O address location
K_AXST              .set      1h              ; 0h
K_ARST              .set      2h              ; used to check the control bits
K_BXST              .set      3h              ; check if K_FIFO_SIZE
                    .set      64              ; its a 64 FIFO
K_FRAME_SIZE        .set      256             ; Frame size
K_HOST_FLAG         .set      1               ; if 0, then host interface
                                              ; is disabled
```

Example 10–23. Interrupt 1 Service Routine

```
;    TEXAS INSTRUMENTS INCORPORATED
;    DSP Data Communication System Development / ASP
;
;    Archives:    PVCS
;    Filename:    hst_int1.asm
;    Version:     1.0
;    Status :     draft         ( )
;                 proposal      (X)
;                 accepted      ( ) dd–mm–yy/?acceptor.
;
;    AUTHOR    Padma P. Mallela
;
;            Application Specific Products
;            Data Communication System Development
;            12203 SW Freeway, MS 701
;            Stafford, TX 77477
;{
;    IPR statements description (can be collected).
;}
;   (C) Copyright 1996. Texas Instruments. All rights reserved.
;
;{
;    Change history:
;
;        VERSION      DATE     /     AUTHORS           COMMENT
;          1.0     July–25–96  /    P.Mallela      original created
;
;}
;{
;    1. ABSTRACT
;
;    1.1 Function Type
;        a.Core Routine
;        b.Subroutine
;
;    1.2 Functional Description
;        This file contains interrput service routine INT1:
;        1) host_command_int1
;    1.3 Specification/Design Reference (optional)
;        INT1 is serviced whenever host writes to CH A
;
;    1.4 Module Test Document Reference
;        Not done
;
;    1.5 Compilation Information
;        Compiler:       TMS320C54X  ASSEMBLER
;        Version:        1.02 (PC)
;        Activation:     asm500 –s hst_int1.asm
;
;    1.6 Notes and Special Considerations
;        –
;}
```

Example 10–23. Interrupt 1 Service Routine (Continued)

```
;{
;    2. VOCABULARY
;
;    2.1 Definition of Special Words, Keywords (optional)
;        –
;    2.2 Local Compiler Flags
;        –
;    2.3 Local Constants
;        –
;}
;{
;    3. EXTERNAL RESOURCES
;
;    3.1 Include Files
         .mmregs
         .include    ”target.inc”
;    3.2 External Data
;        .ref        FIFO_DP
         .ref        d_command_reg
         .ref        d_fifo_count
         .ref        FIFO_DP
         .ref        d_command_value
         .ref        d_fifo_ptr
         .ref        d_output_addr
;        3.3 Import Functions
;}
;{
;    4. INTERNAL RESOURCES
;
;    4.1 Local Static Data
;        –
;    4.2 Global Static Data
;        –
;    4.3 Dynamic Data
;        –
;    4.4 Temporary Data
;        –
;    4.5 Export Functions
         .def        host_command_int1
;}
;    5. SUBROUTINE CODE
;    HeaderBegin
;===============================================================================
;
;-------------------------------------------------------------------------------
;    5.1 host_command_int1
;
;    5.2 Functional Description
;        The host generates INT1 DSP whenever it writes to CH A. In INT1
;        service routine, the command from host is read whether the FIFO
;        has been empty. Writes another 32 data from target to FIFO.
;        Sends a command to host. The host acknowledges the command and read
;        the 32 data from the FIFO and sends a command to the target for
```

Example 10–23. Interrupt 1 Service Routine (Continued)

```
;        another set of 32 data. This process continues for 6 times till all
;        256 processed samples are transferred to host. Processing INT1 is
;        done background, i.e., INT1 is globally enabled.
;-------------------------------------------------------------------------------
;
;    5.3 Activation
;        Activation example:
;        BD    host_command_int1
;        PSHM    ST0
;        PSHM    ST1
;
;        Reentrancy:    No
;        Recursive :    No
;
;    5.4 Inputs
;        Data structure:  d_fifo_count
;        Data Format:     16–bit var
;        Modified:        Yes
;        Description:     counter for # of transfers
;
;        Data structure:  d_fifo_ptr
;        Data Format:     16–bit variable
;        Modified:        Yes
;        Description:     holds the output bffr addr. and incremented by
;                         32 for every transfer
;
;    5.5 Outputs
;
;        Data structure:  AR7
;        Data Format:     16–bit output buffer pointer
;        Modified:        Yes
;        Description:     either point to PING/PONG buffer
;
;    5.6 Global
;        Data structure:  d_command_reg
;        Data Format:     16–bit variable
;        Modified:        Yes
;        Description:     command from host is read thru CH A
;
;        Data structure:  d_command_value
;        Data Format:     16–bit variable
;        Modified:        Yes
;        Description:     holds the command value
;
;    5.7 Special considerations for data structure
;        –
```

Example 10–23. Interrupt 1 Service Routine (Continued)

```
;    5.8 Entry and Exit conditions
;
;   |DP|OVM|SXM|C16|FRCT|ASM|AR0|AR1|AR2|AR3|AR4|AR5|AR6|AR7|A |B |BK|BRC| T|TRN
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |
;in |U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |U  |U  |NU|NU|NU|NU |NU|NU
;   |  |   |   |   |    |   |   |   |   |   |   |   |   |   |  |  |  |   |  |
;out|U | 1 | 1 |NU | 1  |NU |NU |NU |NU |NU |NU |NU |NU |U  |UM|NU|NU|NU |NU|NU
;
;     Note : UM – Used & Modified, U – Used, NU – Not Used
;
;    5.9 Execution
;        Execution time: ?cycles
;        Call rate:      not applicable for this application
;
;=============================================================================
;    HeaderEnd
;    5.10 Code
          .asg    AR7,OUTBUF_P               ; output buffer pointer
          .asg    AR7,SV_RSTRE_AR7
          .sect   ”fifo_fil”
host_command_int1:
          PSHM    AL
          PSHM    AH
          PSHM    AG
          PSHM    SV_RSTRE_AR7               ; AR7 is used as a poiner for
                                             ; output buffer
          LD      #FIFO_DP,DP                ; restore the DP
          PORTR   K_CHA,d_command_value      ; read command from host
wait_host_receive_data
          PORTR   K_TRGCR_ADDR,d_command_reg  ; while (port14 & AXST)
          BITF    d_command_reg,K_ARST       ; check FIFO empty
          BC      wait_host_receive_data,TC  ; branch occurs
          LD      #K_FIFO_EMPTY,A            ; indicate of FIFO empty
          SUB     d_command_value,A
bad_command
          BC      bad_command,ANEQ           ; read the command send by host
          LD      #(K_FRAME_SIZE/(K_FIFO_SIZE/2))–1,A
          SUB     d_fifo_count,A             ; check for complete transfer of
                                             ; 256 samples
          BC      start_remain_fifo_transfer,AGT
          BD      transfer_over
          ST      #0,d_fifo_count            ; reset the fifo count
                                             ; start_remain_fifo_transfer
          MVDK    d_fifo_ptr,OUTBUF_P        ; load PING/PONG bffr address
```

Example 10–23. Interrupt 1 Service Routine (Continued)

```
          RPT     #(K_FIFO_SIZE/2)–1          ; write 32 data to FIFO
          PORTW   *OUTBUF_P+,K_CHB            ; Fill FIFO half
          ST      #K_FIFO_FULL,d_command_value
          PORTW   d_command_value, K_CHA      ; write command to cmmnd reg A
                                              ; for FIFO half
          ADDM    #1,d_fifo_count
          MVKD    OUTBUF_P,d_fifo_ptr         ; save the fifo_ptr
transfer_over:
          POPM    SV_RSTRE_AR7                ; restore AR7
          POPM    AG
          POPM    AH
          POPM    AL
          POPM    ST1
          POPM    ST0
          RETE
          .end
```

Example 10–24. Function Calls on Host Side

```
    Host Action
/*************************************************************************/
/*                            FILE NAME: HOST.C                          */
/*                  C54x EVM/HOST COMMUNICATION FUNCTIONS -- HOST SIDE    */
/*************************************************************************/
    #include ”graphic2.c”
    #include ”host.h”         /* flag names, constants       */
/*_______________________________________________________________________
    This function initializes the data buffer and reads the FIFO so that FIFO
    is empty when the real data transfers start
-------------------------------------------------------------------------*/
    void initialize_slave(void)
{
        int j;
        for (j=0;j < 64; j++)
        dataa[j] = inport(BDAT_REG)  ; /* read data from data reg.    */
        for (j=0;j <256; j++)
        dataa[j] = 0;
        outport(CONT_REG, inport(CONT_REG) & 0xf7ff);
 }
/*_______________________________________________________________________
 The target sends a command to the host after collecting 32 word data from DSP
    memory to FIFO. The host checks if the command has been received
 ------------------------------------------------------------------------*/
int receive_command_FIFO_FULL(void)
                        /* RECEIVE COMMAND FROM EVM    */
{
while(!(inport(CONT_REG) & ARST))    ; /* wait for evm to send command*/

reply = inport(ADAT_REG)             ; /* read command into reply*/

while ((reply & 0xFF) !=0xFF)        ; return(reply)
                                     ; /* return command for process’g*/
 }
/*_______________________________________________________________________*/
/*    This function sends a command to target for a new set of data from FIFO*/
      void send_command_new_FIFO(command unsigned int command;
 {
               command = 0xEE;
               outport (ADAT_REG,command);
               while(inport(CONT_REG) & AXST);
  }
/*-------------------------------------------------------------------------
      This initiates the handshake between the target and host. The host writes
      a command to target which sets the AXST flag to 1. The INT1 is generated
      whenever host writes to CH A. On the target side, INT1 is polled and reads
      the CH A.This clears ARST on target side. A command is written to Ch A on
      target after emptying the FIFO that sets AXSt =1. Later sets XF to go low.
      On host XF is polled and then reads CH A that clears ARST to 0 and AXST to 0
      on the target side
*/
   int receive_clear_AXST(void)
                                  /* RECEIVE COMMAND FROM EVM    */
```

Example 10–24. Function Calls on Host Side (Continued)

```
{
    command = 0xAB;
    outport (ADAT_REG,command);
    while(inport(CONT_REG) & AXST);      /* write command to evm */
    while((inport(CONT_REG) & XF));
    while(!(inport(CONT_REG) & ARST));  /* wait for evm to send command*/
    reply = inport(ADAT_REG);           /* read command into reply     */
    while ((reply & 0xAE) !=0xAE);
    return(reply);                              /* return command for process'g*/
    }
/*-------------------------------------------------------------------------*/
```

Example 10–25. Main Function Call on Host Side

```
/*************************************************************************/
/*                              FILE NAME: MASTER.C                      */
/*                  C54x EVM/HOST COMMUNICATION FUNCTIONS -- HOST SIDE    */
/*************************************************************************/
    #define   F1    15104
    #define   F2    15360
    #define   F3    15616
    #define   DATA_FRAME  256
    #include <bios.h>
    #include ”view2.c”
    int get_kbhit(void);
    extern void send_command_new_FIFO(unsigned int);
    void main(void)
{
    int count=0,n,fifo_size;
    int main_done =0;
    int done = 0;
    int hit;
    initialize_slave();
/*   a command is written to CH A to initiate handshake */
     command_AXST = receive_clear_AXST()
;    while (!main_done)
{
      done =0;
      init_graphics()
    ; while(!done)
{
      count =0;
      if(kbhit())
{
      hit = get_kbhit();
      setviewport(120,433,290,445,0);
      clearviewport();
      switch(hit)
{
    case    (F1):    amplitude = amplitude * 2    ; outtextxy(1,1,”Amplitude
                                                  ; decreased”)
                                                  ; break;
//  case    (F2):    if (amplitude>=2) amplitude = amplitude / 2
                                                  ; outtextxy(1,1,”Amplitude
                                                  ; increased”)
                                                  ; break;
    case    (F2):    done =1                      ; closegraph()
                                                  ; break;
    case    (F3):    done =1                      ;main_done=1
                                                  ;closegraph()
                                                  ;break;
}
}
else
```

Example 10–25. Main Function Call on Host Side (Continued)

```
{
    command_FIFO = receive_command_FIFO_FULL();
    for (fifo_size=0; fifo_size < 64; fifo_size++)
    dataa[fifo_size+count] = inport(BDAT_REG);
    send_command_new_FIFO(command);
    for (count=64; count< 256; count++)
{
    command_FIFO = receive_command_FIFO_FULL();     /* command from target*/
    for (fifo_size=0; fifo_size < 32; fifo_size++)
    dataa[fifo_size+count] = inport(BDAT_REG);      /* read 32 word fifo*/
    count = count+31;
    send_command_new_FIFO(command);                /* send command to target*/
}
    screen();
}
}
}
    closegraph();
}
int get_kbhit(void)
{
    unsigned int key = bioskey(0);
    fflush(stdin);
    return(key);
}
/*****************************************************************************/
/*        FILE NAME: HOST.H  C54x                                            */
/*      HOST SIDE FLAGS, CONSTANTS, COMMAND NUMBERS, AND GLOBAL VARIABLES    */
/*                                                                           */
/*****************************************************************************/
/* The numbers I've picked for the file I/O constants, command numbers, and  */
/* basic control constants are not important. These numbers could really     */
/* be anything, as long as two of them are not the same.  Please notice the  */
/* pattern I used for file commands. Masks and pointers CANNOT be changed.   */
#include <stdio.h>
/*----------------------- FILE I/O CONSTANTS AND COMMAND ---------------------*/
#define  MAX_FRAME        256      /* size of data frame to be passed */
/*------------------------- BASIC CONTROL CONSTANTS --------------------------*/
#define  STOP         99
#define  NO           98
#define  YES          97
#define  READY        96
#define  CLEAR        95
#define  ACKNOWLEDGE  0
/*------------- POINTERS TO DATA, COMMAND, AND CONTROL REGISTERS -------------*/
unsigned int    ADAT_REG     = (unsigned int )(0x240 + 0x800);
unsigned int    BDAT_REG     = (unsigned int )(0x240 + 0x804);
unsigned int    CONT_REG     = (unsigned int )(0x240 + 0x808);
```

Example 10–25. Main Function Call on Host Side (Continued)

```
/*------------ MASKS FOR READING MESSAGE FLAGS OF CONTROL REGISTER -----------*/
unsigned int    XF          = (unsigned int ) 0x0020;
unsigned int    ARST        = (unsigned int ) 0x0002;
unsigned int    AXST        = (unsigned int ) 0x0001;
unsigned int    BRST_MASK   = (unsigned int ) 0x0200;
unsigned int    BXST_MASK   = (unsigned int ) 0x0008;
/*---------------- GLOBAL VARIABLES USED BY EVM.C AND MASTER.C ---------------*/
FILE*file[20];            /* stores ptrs to files in files.dat*/
int     reply;            /* integer for whatever            */
int     reply1[128];
int     data[256];
int     index;
unsigned int command;
unsigned int command1;
unsigned int command_new_data;
unsigned int command_FIFO;
unsigned int command_AXST;
unsigned int command_HANDSHAKE;
int amplitude = –10;
```

Example 10–26. Graphic Drivers Routine

```
/*************************************************************************/
/*                          FILE NAME: GRAPHIC2.C                        */
/*                  GRAPHICS DRIVER INITIALIZATION ROUTINE               */
/*                                                                       */
/*************************************************************************/
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
void init_graphics(void)
{
            int i;
            int gdriver = DETECT, gmode, errorcode; int left, top, right, bottom;
            initgraph(&gdriver, &gmode, ""); errorcode = graphresult();
            if (errorcode != grOk);
            {
                        printf("Graphics error: %s\n", grapherrormsg(errorcode));
                        printf("Press any key to halt.\n");
                        getch();
    }
            cleardevice();
            setlinestyle(0,0,1);
            setcolor(1);
            setfillstyle(1,3);
            rectangle(1,1,getmaxx()–1,getmaxy()–1);
            setcolor(7);
            left = 5;
            top = getmaxy()/2 + – 123;
            right = 108;
            bottom = getmaxy()/2+ 163;
            rectangle(left,top,right,bottom);
            line(left,top + 30,right,top + 30);
            line(left,bottom+29,right,bottom+29);
            rectangle(left,top – 110,right+523,top – 5);
            rectangle(right+4,bottom + 5, right + 523,bottom + 70);
            rectangle(left,botom +5,right,bottom+70);
            setcolor(15);
            settextstyle(0,0,1);
/*          outtextxy(left+25,top+12,"CONTROL");
            outtextxy(right+16,bottom+32,"MESSAGES:");
            outtextxy(left+13,bottom+13,"AIC STATUS");*/
            outtextxy(left+2,bottom+38,"Freq: 8kHz");
            outtextxy(left+2,bottom+53,"Gain: 2");
            settextstyle(0,0,3);
            outtextxy(left+10,top–72,"C54X EVM Spectrum Analyzer");
            setlinestyle(0,0,3);
            setcolor(15);
            left   = getmaxx() / 2 – 206;
            top    = getmaxy() / 2 – 123;
            right  = getmaxx() / 2 + 312;
            bottom = getmaxy() / 2 + 163;
            rectangle(left, top, right, bottom);
            floodfill(left+10,top+10,15);
```

Example 10–26. Graphic Drivers Routine (Continued)

```
            setcolor(15);
            setlinestyle(0,0,3);
            left   = getmaxx()/2 – 311;
            top    = getmaxy()/2 + 57 – 40;
            right  = left + 25;
            bottom = top +  25; rectangle(left,top,right,bottom);
            setfillstyle(1,1); floodfill(left+5,top+5,15);
            settextstyle(0,0,1);
            outtextxy(left+5,top+10,”F3”);
//          outtextxy(left+30,top+10,”Quit”);
            setcolor(15); setlinestyle(0,0,3);
            left   = getmaxx()/2 – 311;
            top    = getmaxy()/2 – 23 – 40;
            right  = left + 25;
            bottom = top + 25; rectangle(left,top,right,bottom)
            setfillstyle(1,1); floodfill(left+5,top+5,15);
            settextstyle(0,0,1); outtextxy(left+6,top+10,”F1”);
            outtextxy(left+30,top+9,”Decrease”);
            outtextxy(left+30,top+16,”amplitude”);
            setcolor(15);
            setlinestyle(0,0,3);
            left   = getmaxx()/2 – 311;
            top    = getmaxy()/2 + 17 – 40;
            right  = left + 25;
            bottom = top + 25;
            rectangle(left,top,right,bottom);
            setfillstyle(1,1);
            floodfill(left+5,top+5,15);
            settextstyle(0,0,1); outtextxy(left+6,top+10,”F2”);
            outtextxy(left+30,top+9,”Increase”);
            outtextxy(left+30,top+16,”amplitude”);
```

Example 10–27. Display the Data on the Screen

```
/*************************************************************************/
/*                            FILE NAME: VIEW2.C                         */
/*                       DISPLAYS DATA ON THE SCREEN                     */
/*                                                                       */
/*************************************************************************/
#include ”host.c”
void screen(void)
{
    int x,y;
    int i,n;
    /* setup window (viewport) to display the AIC data */
    x = getmaxx()/2 – 203;
    y = getmaxy()/2;
    setviewport(x,y–120,x+512,y+160,1);
    /* move CP to left side of viewport */
    x = getx();
    y = gety() + 143;
    moveto(x,y);
/* make waveform by drawing lines btwn 256 data points sent by EVM */
setlinestyle(0,0,1);
setcolor(14);
for(i=0;i<256;i++) lineto(x+1+2*i,y+dataa[i]/amplitude);
/* erase waveform just drawn by re–writing it in background color */
moveto(x,y);
setcolor(3);
for(i=0;i<256;i++) lineto(x+1+2*i,y+dataa[i]/amplitude);
}
```

Example 10–28. Linker Command File for the Application

```
/*************************************************************************
* This linker command file is assigns the memory allocation for the application
* based on the EVM54x specifically 541.
*************************************************************************/
    vectors.obj
    init_54x.obj
    init_ser.obj
    init_aic.obj
    aic_cfg.obj
    memory.obj
    main.obj
    prcs_int.obj
    rcv_int1.obj
    task.obj
    fir.obj
    iir.obj
    sym_fir.obj
    adapt.obj
    echo.obj
    rfft.obj
    bit_rev.obj
    fft.obj
    unpack.obj
    power.obj
    hand_shk.obj
    hst_int1.obj
    fifo.obj
    –o main.out
    –m main.map
MEMORY
{
PAGE 0:
    PROG    : origin = 0x7000,    length = 0x1000
    VECS    : origin = 0xff80,    length = 0x7f
    COFF_SYM: origin = 0x1400,    length = 0x40
    COFF_FIR: origin = 0x1440,    length = 0x40
    AIC_TBLE: origin = 0x1480,    length = 0x10
    TASK_TBL: origin = 0x1490,    length = 0x10
    TASK_INT: origin = 0x14a0,    length = 0x10
    IIR_COFF: origin = 0x14b0,    length = 0x10
    COEFFH  : origin = 0x1500,    length = 0x100
    TWID_SIN: origin = 0x1000,    length = 0x80
    TWID_COS: origin = 0x1200,    length = 0x80
    PAGE 1:                     /* Data space */
    ALL_VARS: origin = 0x0080,    length = 0x0080
    DRAM    : origin = 0x0100,    length = 0x1300
    EXT_DAT : origin = 0x1400,    length = 0xE000
    REGS    : origin = 0x0000,    length = 0x0060
}
```

Example 10–28. Linker Command File for the Application (Continued)

```
SECTIONS
  {
    .text           : {} >  PROG            PAGE 0          /* code */
    vectors         : {} >  VECS            PAGE 0         /* Vector table */
    main_prg        : {} >  PROG            PAGE 0
    zeropad         : {} >  PROG            PAGE 0
    aic_cnfg        : {} >  PROG            PAGE 0
    ser_cnfg        : {} >  PROG            PAGE 0
    fifo_fil        : {} >  PROG            PAGE 0
    task_hnd        : {} >  PROG            PAGE 0
    handshke        : {} >  PROG            PAGE 0
    fir_prog        : {} >  PROG            PAGE 0
    iir             : {} >  PROG            PAGE 0
    filter          : {} >  PROG            PAGE 0
    rfft_prg        : {} >  PROG            PAGE 0
    aic_reg         : {} >  AIC_TBLE        PAGE 0
    task_int        : {} >  TASK_INT        PAGE 0
    task_tbl        : {} >  TASK_TBL        PAGE 0
    coff_fir        : {} >  COFF_FIR        PAGE 0
    sym_fir         : {} >  COFF_SYM        PAGE 0
    iir_coff        : {} >  IIR_COFF        PAGE 0
    coeffh          : {} >  COEFFH          PAGE 0
    sin_tbl         : {} >  TWID_SIN        PAGE 0
    cos_tbl         : {} >  TWID_COS        PAGE 0
    inpt_buf        : {} >  DRAM,align(1024)PAGE 1
    outdata         : {} >  DRAM,align(1024)PAGE 1
UNION:                  >  DRAM align(1024)PAGE 1
{
    fft_bffr
    adpt_sct:
{
    *(bufferw)              /* This is needed for alignment of 128 words */
    .+=80h;
    *(bufferp)
}
}
UNION:                   >  DRAM align(256) PAGE 1
{
    fir_bfr
    cir_bfr
    coff_iir
    bufferh
    twid_sin
}
UNION:                   >  DRAM align(256) PAGE 1
{
    fir_coff
    cir_bfr1
    bufferx
    twid_cos
}
```

Example 10–28. Linker Command File for the Application (Continued)

```
GROUP:                    > ALL_VARS    PAGE 1
{
    aic_vars
    rcv_vars
    fifo_var
    tsk_vars
    fir_vars
    iir_vars
    adpt_var
    fft_vars
}
stack        : {} >  DRAM PAGE 1
}
```

Example 10–29. Memory Map of TMS320C541

```
;TMS320C541 MEMORY MAP
MR
;
MA 0x0000, 1, 0x002A, RAM  ; MMRs
MA 0x0030, 1, 0x0003, RAM  ;
MA 0x0060, 1, 0x0020, RAM  ; SCRATCH PAD
MA 0x0080, 1, 0x1380, RAM  ; INTERNAL DATA RAM
MA 0x0080, 0, 0x1380, RAM  ; INTERNAL PROGRAM RAM
MA 0x9000, 0, 0x7000, ROM  ; INTERNAL ROM
ma 0x1400, 0, 0xec00, ram  ; external ram
ma 0x1400, 1, 0xec00, ram  ; external ram
ma 0x0000, 2, 0x15, ioport ; i/o space
map on
;
;Define reset alias to set PMST for MC mode
;
;alias myreset, ”e pmst = 0xff80; reset ”
;e pmst = 0xffe0  ; MP mode, OVLY, DROM off CLKOUT on
;e hbpenbl = 0x0000
e *0x28 = 0x2000 ; two wait states on i/o, none for memory
e *0x29 = 0x0000 ; no bank switching necessary
dasm pc
echo Loaded TMS320C54x evminit.cmd
```

Appendix A

Design Considerations for Using XDS510 Emulator

This appendix assists you in meeting the design requirements of the Texas Instruments XDS510 emulator with respect to IEEE-1149.1 designs and discusses the XDS510 cable (manufacturing part number 2617698-0001). This cable is identified by a label on the cable pod marked *JTAG 3/5V* and supports both standard 3-V and 5-V target system power inputs.

The term *JTAG,* as used in this book, refers to TI scan-based emulation, which is based on the IEEE 1149.1 standard.

For more information concerning the IEEE 1149.1 standard, contact IEEE Customer Service:

Address: IEEE Customer Service
445 Hoes Lane, PO Box 1331
Piscataway, NJ 08855-1331

Phone: (800) 678–IEEE in the US and Canada
(908) 981–1393 outside the US and Canada

FAX: (908) 981–9667 Telex: 833233

Topic **Page**

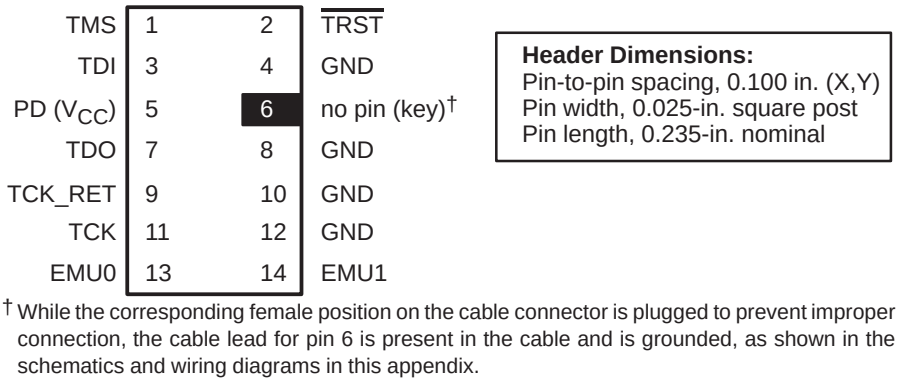
A.1 Designing Your Target System's Emulator Connector (14-Pin Header)

JTAG target devices support emulation through a dedicated emulation port. This port is accessed directly by the emulator and provides emulation functions that are a superset of those specified by IEEE 1149.1. To communicate with the emulator, *your target system must have a 14-pin header* (two rows of seven pins) with the connections that are shown in Figure A–1. Table A–1 describes the emulation signals.

Although you can use other headers, the recommended unshrouded, straight header has these DuPont connector systems part numbers:

- 65610–114
- 65611–114
- 67996–114
- 67997–114

Figure A–1. 14-Pin Header Signals and Header Dimensions



† While the corresponding female position on the cable connector is plugged to prevent improper connection, the cable lead for pin 6 is present in the cable and is grounded, as shown in the schematics and wiring diagrams in this appendix.

Table A–1. 14-Pin Header Signal Descriptions

Signal	Description	Emulator† State	Target† State
EMU0	Emulation pin 0	I	I/O
EMU1	Emulation pin 1	I	I/O
GND	Ground		
PD(V_{CC})	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD should be tied to V_{CC} in the target system.	I	O
TCK	Test clock. TCK is a 10.368-MHz clock source from the emulation cable pod. This signal can be used to drive the system test clock.	O	I
TCK_RET	Test clock return. Test clock input to the emulator. May be a buffered or unbuffered version of TCK.	I	O
TDI	Test data input	O	I
TDO	Test data output	I	O
TMS	Test mode select	O	I
$\overline{\text{TRST}}$‡	Test reset	O	I

† I = input; O = output

‡ Do not use pullup resistors on $\overline{\text{TRST}}$: it has an internal pulldown device. In a low-noise environment, $\overline{\text{TRST}}$ can be left floating. In a high-noise environment, an additional pulldown resistor may be needed. (The size of this resistor should be based on electrical current considerations.)

A.2 Bus Protocol

The IEEE 1149.1 specification covers the requirements for the test access port (TAP) bus slave devices and provides certain rules, summarized as follows:

- The TMS and TDI inputs are sampled on the rising edge of the TCK signal of the device.
- The TDO output is clocked from the falling edge of the TCK signal of the device.

When these devices are daisy-chained together, the TDO of one device has approximately a half TCK cycle setup time before the next device's TDI signal. This timing scheme minimizes race conditions that would occur if both TDO and TDI were timed from the same TCK edge. The penalty for this timing scheme is a reduced TCK frequency.

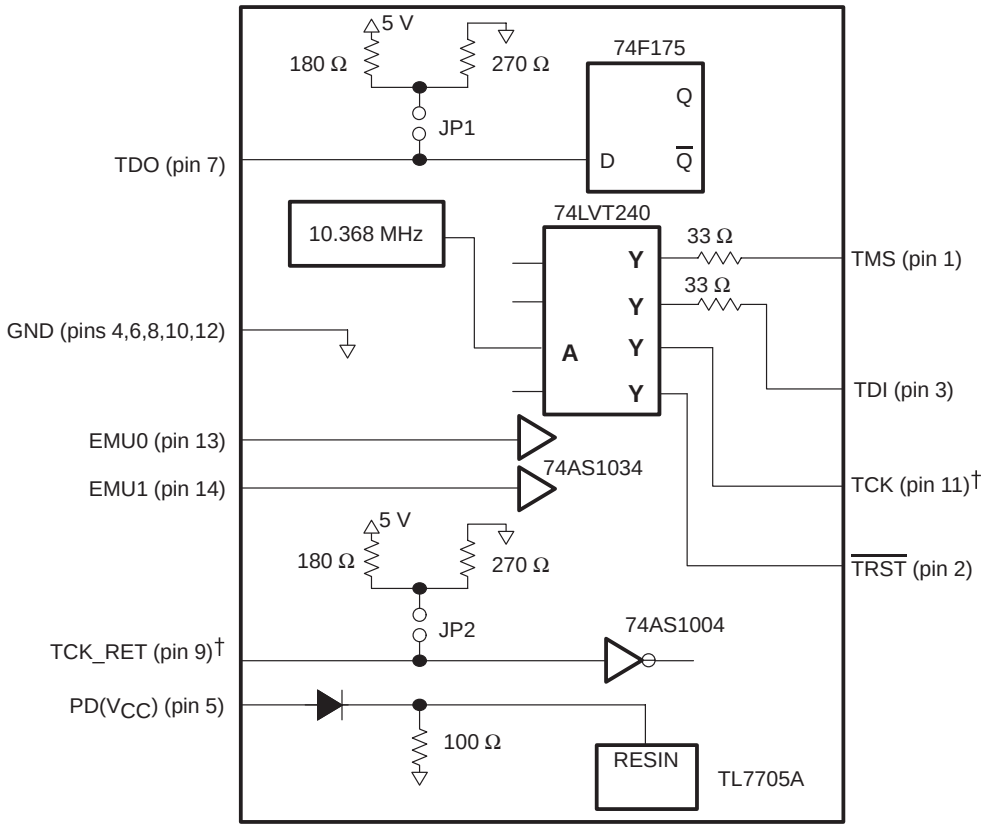
The IEEE 1149.1 specification does not provide rules for bus master (emulator) devices. Instead, it states that the device expects a bus master to provide bus slave compatible timings. The XDS510 provides timings that meet the bus slave rules.

A.3 Emulator Cable Pod

Figure A–2 shows a portion of the emulator cable pod. The functional features of the pod are:

- TDO and TCK_RET can be parallel-terminated inside the pod if required by the application. By default, these signals are not terminated.
- TCK is driven with a 74LVT240 device. Because of the high-current drive (32-mA I_{OL}/I_{OH}), this signal can be parallel-terminated. If TCK is tied to TCK_RET, you can use the parallel terminator in the pod.
- TMS and TDI can be generated from the falling edge of TCK_RET, according to the IEEE 1149.1 bus slave device timing rules.
- TMS and TDI are series-terminated to reduce signal reflections.
- A 10.368-MHz test clock source is provided. You can also provide your own test clock for greater flexibility.

Figure A–2. Emulator Cable Pod Interface



† The emulator pod uses TCK_RET as its clock source for internal synchronization. TCK is provided as an optional target system test clock source.

A.4 Emulator Cable Pod Signal Timing

Figure A–3 shows the signal timings for the emulator cable pod. Table A–2 defines the timing parameters illustrated in the figure. These timing parameters are calculated from values specified in the standard data sheets for the emulator and cable pod and are for reference only. Texas Instruments does not test or guarantee these timings.

The emulator pod uses TCK_RET as its clock source for internal synchronization. TCK is provided as an optional target system test clock source.

Figure A–3. Emulator Cable Pod Timings

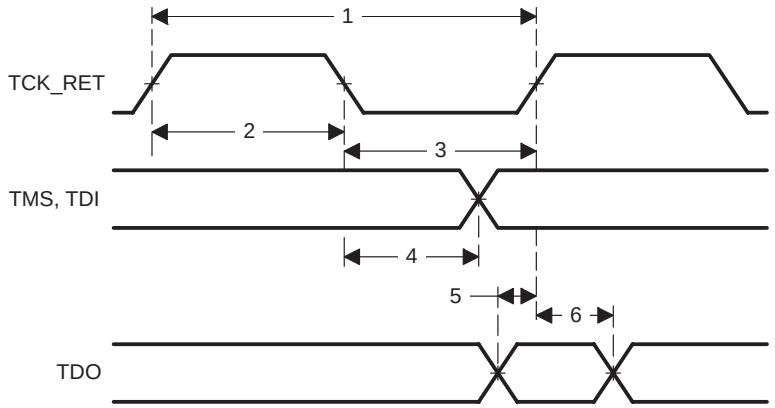


Table A–2. Emulator Cable Pod Timing Parameters

No.	Parameter	Description	Min	Max	Unit
1	$t_{c(TCK)}$	Cycle time, TCK_RET	35	200	ns
2	$t_{w(TCKH)}$	Pulse duration, TCK_RET high	15		ns
3	$t_{w(TCKL)}$	Pulse duration, TCK_RET low	15		ns
4	$t_{d(TMS)}$	Delay time, TMS or TDI valid for TCK_RET low	6	20	ns
5	$t_{su(TDO)}$	Setup time, TDO to TCK_RET high	3		ns
6	$t_{h(TDO)}$	Hold time, TDO from TCK_RET high	12		ns

A.5 Emulation Timing Calculations

Example A–1 and Example A–2 help you calculate emulation timings in your system. For actual target timing parameters, see the appropriate data sheet for the device you are emulating.

The examples use the following assumptions:

$t_{su(TTMS)}$	Setup time, target TMS or TDI to TCK high	10 ns
$t_{d(TTDO)}$	Delay time, target TDO from TCK low	15 ns
$t_{d(bufmax)}$	Delay time, target buffer maximum	10 ns
$t_{d(bufmin)}$	Delay time, target buffer minimum	1 ns
$t_{bufskew}$	Skew time, target buffer between two devices in the same package: $[t_{d(bufmax)} - t_{d(bufmin)}] \times 0.15$	1.35 ns
$t_{TCKfactor}$	Duty cycle, assume a 40/60% duty cycle clock	0.4 (40%)

Also, the examples use the following values from Table A–2 on page A-6:

$t_{d(TMSmax)}$	Delay time, emulator TMS or TDI from TCK_RET low, maximum	20 ns
$t_{su(TDOmin)}$	Setup time, TDO to emulator TCK_RET high, minimum	3 ns

There are two key timing paths to consider in the emulation design:

- The TCK_RET-to-TMS or TDI path, called $t_{pd(TCK\_RET\text{-}TMS/TDI)}$ (propagation delay time)
- The TCK_RET-to-TDO path, called $t_{pd(TCK\_RET\text{-}TDO)}$

In the examples, the worst-case path delay is calculated to determine the maximum system test clock frequency.

Example A–1. Key Timing for a Single-Processor System Without Buffers

$$t_{pd\,(TCK\_RET\text{-}TMS/TDI)} = \frac{\left[t_{d\,(TMSmax)} + t_{su\,(TTMS)}\right]}{t_{TCKfactor}}$$

$$= \frac{(20\ ns + 10\ ns)}{0.4}$$

$$= 75\ ns,\ or\ 13.3\ MHz$$

$$t_{pd\,(TCK\_RET–TDO)} = \frac{\left[t_{d\,(TTDO)} + t_{su\,(TDOmin)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\ ns + 3\ ns)}{0.4}$$

$$= 45\ ns,\ or\ 22.2\ MHz$$

In this case, because the TCK_RET-to-TMS/TDI path requires more time to complete, it is the limiting factor.

Example A–2. Key Timing for a Single- or Multiple-Processor System With Buffered Input and Output

$$t_{pd\,(TCK\_RET\text{-}TMS/TDI)} = \frac{\left[t_{d\,(TMSmax)} + t_{su\,(TTMS)} + t_{bufskew}\right]}{t_{TCKfactor}}$$

$$= \frac{(20\ ns + 10\ ns + 1.35\ ns)}{0.4}$$

$$= 78.4\ ns,\ or\ 12.7\ MHz$$

$$t_{pd\,(TCK\_RET–TDO)} = \frac{\left[t_{d\,(TTDO)} + t_{su\,(TDOmin)} + t_{d\,(bufmax)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\ ns + 3\ ns + 10\ ns)}{0.4}$$

$$= 70\ ns,\ or\ 14.3\ MHz$$

In this case also, because the TCK_RET-to-TMS/TDI path requires more time to complete, it is the limiting factor.

In a multiprocessor application, it is necessary to ensure that the EMU0 and EMU1 lines can go from a logic low level to a logic high level in less than 10 μs, this parameter is called rise time, t_r. This can be calculated as follows:

$$
\begin{aligned}
t_r &= 5(R_{pullup} \times N_{devices} \times C_{load\_per\_device}) \\
&= 5(4.7\ k\Omega \times 16 \times 15\ pF) \\
&= 5(4.7 \times 10^3\ \Omega \times 16 \times 15 = no^{-12}\ F) \\
&= 5(1128 \times 10^{-9}\ ) \\
&= 5.64\ \mu s
\end{aligned}
$$

A.6 Connections Between the Emulator and the Target System

It is extremely important to provide high-quality signals between the emulator and the JTAG target system. You must supply the correct signal buffering, test clock inputs, and multiple processor interconnections to ensure proper emulator and target system operation.

Signals applied to the EMU0 and EMU1 pins on the JTAG target device can be either input or output. In general, these two pins are used as both input and output in multiprocessor systems to handle global run/stop operations. EMU0 and EMU1 signals are applied only as inputs to the XDS510 emulator header.

A.6.1 Buffering Signals

If the distance between the emulation header and the JTAG target device is greater than 6 inches, the emulation signals must be buffered. If the distance is less than 6 inches, no buffering is necessary. Figure A–4 shows the simpler, no-buffering situation.

The distance between the header and the JTAG target device must be no more than 6 inches. The EMU0 and EMU1 signals must have pullup resistors connected to V_{CC} to provide a signal rise time of less than 10 μs. A 4.7-kΩ resistor is suggested for most applications.

Figure A–4. Emulator Connections Without Signal Buffering

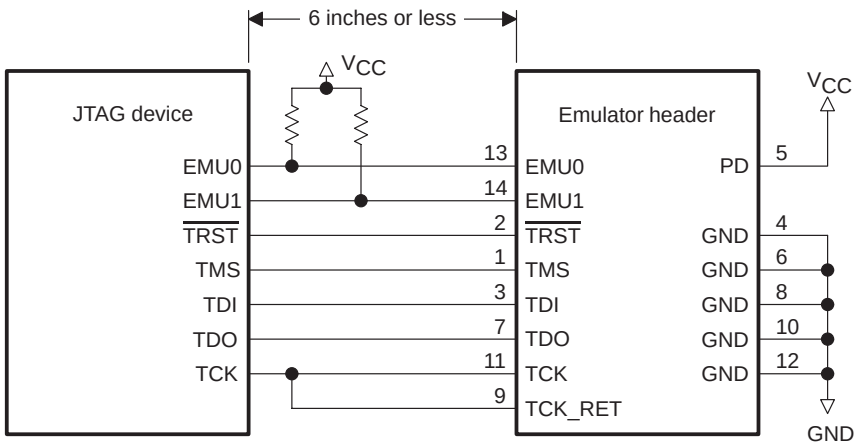
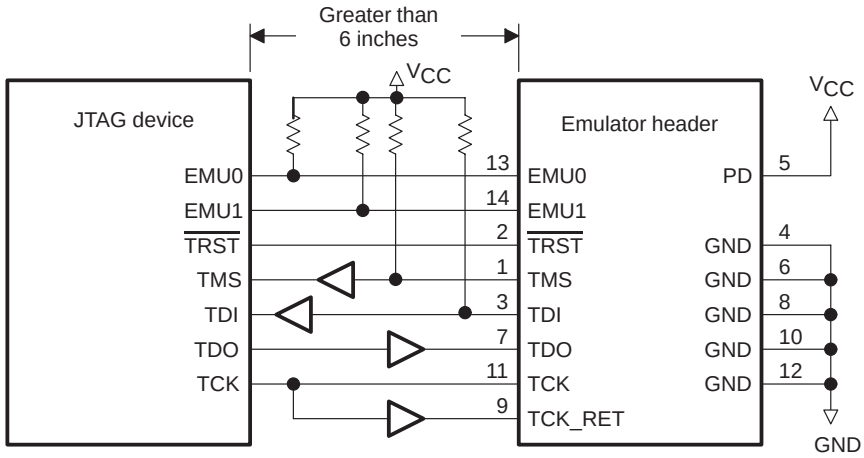


Figure A–5 shows the connections necessary for buffered transmission signals. The distance between the emulation header and the processor is greater than 6 inches. Emulation signals TMS, TDI, TDO, and TCK_RET are buffered through the same device package.

Figure A–5. Emulator Connections With Signal Buffering



The EMU0 and EMU1 signals must have pullup resistors connected to V_{CC} to provide a signal rise time of less than 10 μs. A 4.7-kΩ resistor is suggested for most applications.

The input buffers for TMS and TDI should have pullup resistors connected to V_{CC} to hold these signals at a known value when the emulator is not connected. A resistor value of 4.7 kΩ or greater is suggested.

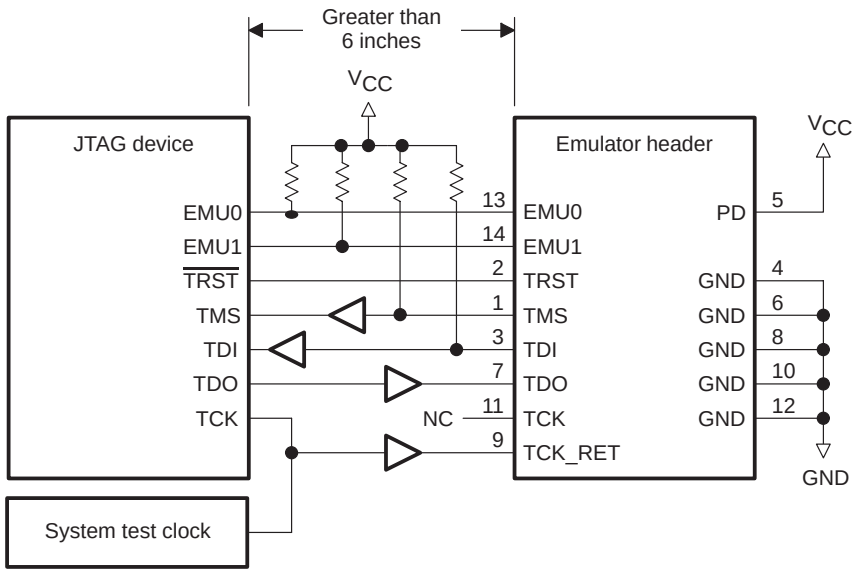
To have high-quality signals (especially the processor TCK and the emulator TCK_RET signals), you may have to employ special care when routing the printed wiring board trace. You also may have to use termination resistors to match the trace impedance. The emulator pod provides optional internal parallel terminators on the TCK_RET and TDO. TMS and TDI provide fixed series termination.

Because $\overline{\text{TRST}}$ is an asynchronous signal, it should be buffered as needed to ensure sufficient current to all target devices.

A.6.2 Using a Target-System Clock

Figure A–6 shows an application with the system test clock generated in the target system. In this application, the emulator's TCK signal is left unconnected.

Figure A–6. Target-System-Generated Test Clock



Note: When the TMS and TDI lines are buffered, pullup resistors must be used to hold the buffer inputs at a known level when the emulator cable is not connected.

There are two benefits in generating the test clock in the target system:

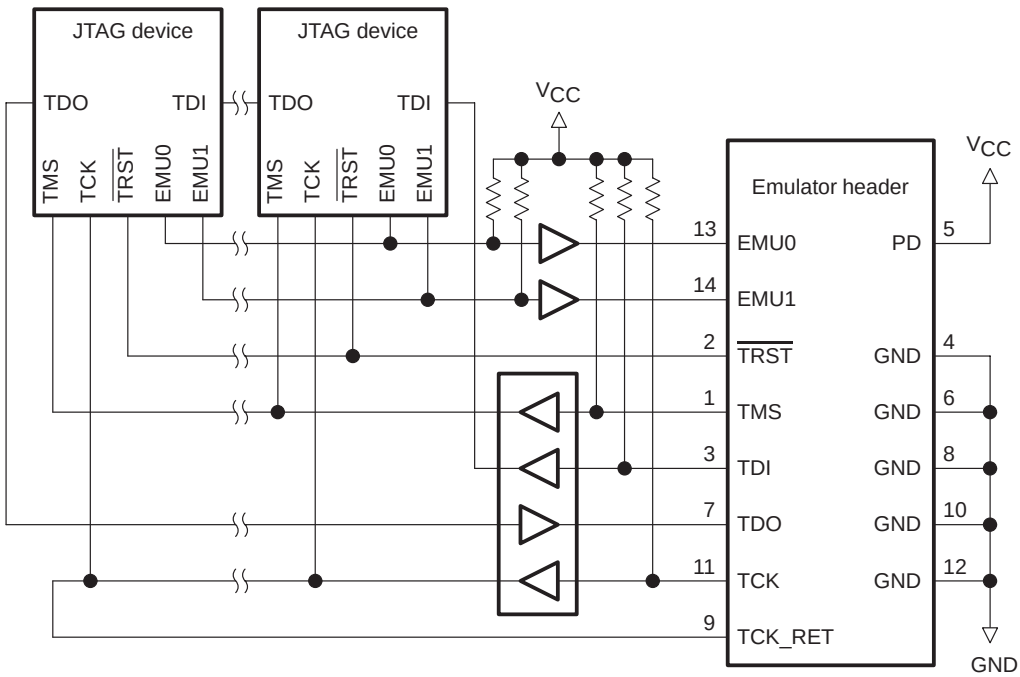
- The emulator provides only a single 10.368-MHz test clock. If you allow the target system to generate your test clock, you can set the frequency to match your system requirements.
- In some cases, you may have other devices in your system that require a test clock when the emulator is not connected. The system test clock also serves this purpose.

A.6.3 Configuring Multiple Processors

Figure A–7 shows a typical daisy-chained multiprocessor configuration that meets the minimum requirements of the IEEE 1149.1 specification. The emulation signals are buffered to isolate the processors from the emulator and provide adequate signal drive for the target system. One of the benefits of this interface is that you can slow down the test clock to eliminate timing problems. Follow these guidelines for multiprocessor support:

- The processor TMS, TDI, TDO, and TCK signals must be buffered through the same physical device package for better control of timing skew.
- The input buffers for TMS, TDI, and TCK should have pullup resistors connected to V_{CC} to hold these signals at a known value when the emulator is not connected. A resistor value of 4.7 kΩ or greater is suggested.
- Buffering EMU0 and EMU1 is optional but highly recommended to provide isolation. These are not critical signals and do not have to be buffered through the same physical package as TMS, TCK, TDI, and TDO.

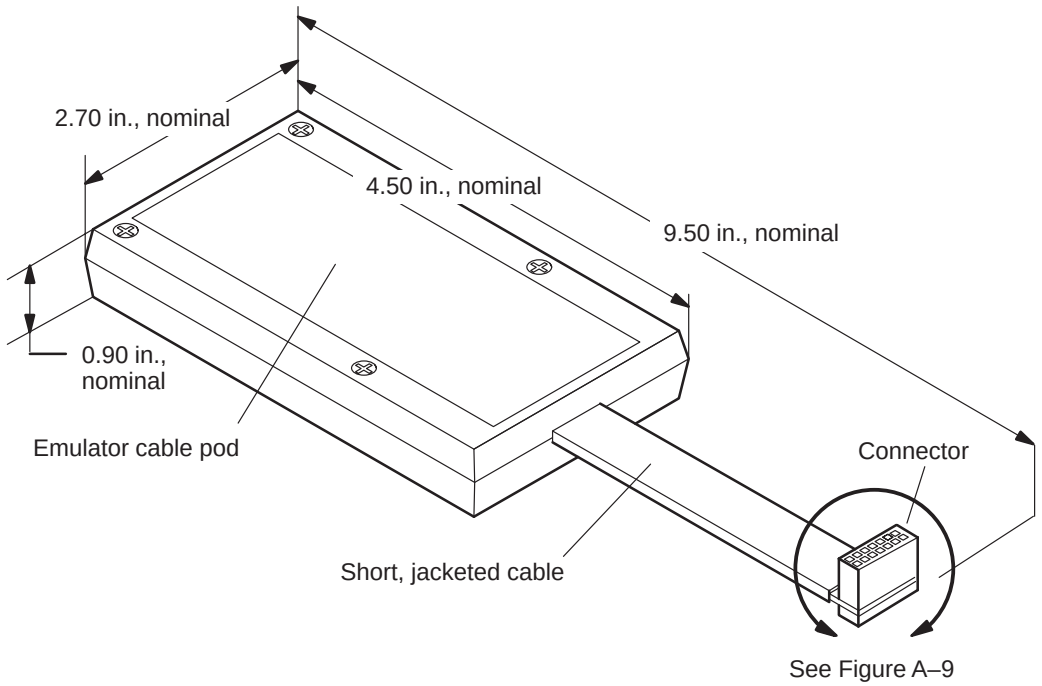
Figure A–7. Multiprocessor Connections



A.7 Physical Dimensions for the 14-Pin Emulator Connector

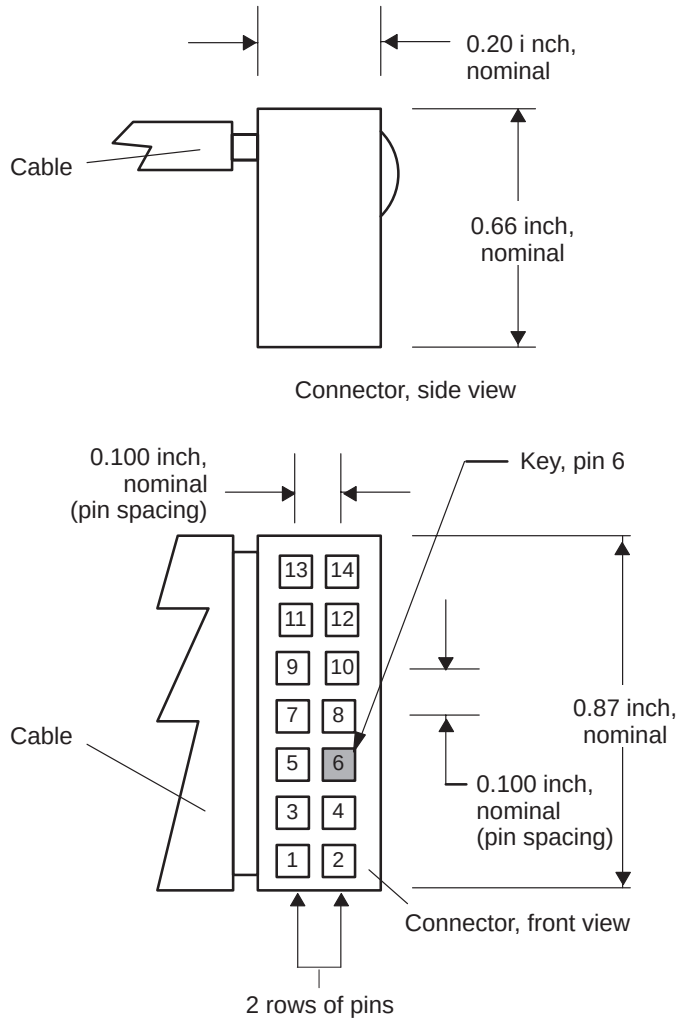
The JTAG emulator target cable consists of a 3-foot section of jacketed cable that connects to the emulator, an active cable pod, and a short section of jacketed cable that connects to the target system. The overall cable length is approximately 3 feet 10 inches. Figure A–8 and Figure A–9 (page A-15) show the physical dimensions for the target cable pod and short cable. The cable pod box is nonconductive plastic with four recessed metal screws.

Figure A–8. Pod/Connector Dimensions



Note: All dimensions are in inches and are nominal dimensions, unless otherwise specified. Pin-to-pin spacing on the connector is 0.100 inches in both the X and Y planes.

Figure A–9. 14-Pin Connector Dimensions



A.8 Emulation Design Considerations

This section describes the use and application of the scan path linker (SPL), which can simultaneously add all four secondary JTAG scan paths to the main scan path. It also describes the use of the emulation pins and the configuration of multiple processors.

A.8.1 Using Scan Path Linkers

You can use the TI ACT8997 scan path linker (SPL) to divide the JTAG emulation scan path into smaller, logically connected groups of 4 to 16 devices. As described in the *Advanced Logic and Bus Interface Logic Data Book*, the SPL is compatible with the JTAG emulation scanning. The SPL is capable of adding any combination of its four secondary scan paths into the main scan path.

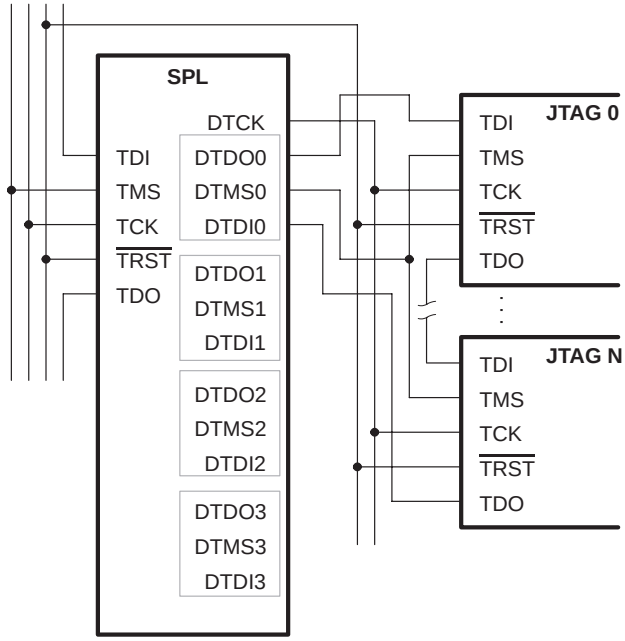
A system of multiple, secondary JTAG scan paths has better fault tolerance and isolation than a single scan path. Since an SPL has the capability of adding all secondary scan paths to the main scan path simultaneously, it can support global emulation operations, such as starting or stopping a selected group of processors.

TI emulators do not support the nesting of SPLs (for example, an SPL connected to the secondary scan path of another SPL). However, you can have multiple SPLs on the main scan path.

Scan path selectors are not supported by this emulation system. The TI ACT8999 scan path selector is similar to the SPL, but it can add only one of its secondary scan paths at a time to the main JTAG scan path. Thus, global emulation operations are not assured with the scan path selector.

You can insert an SPL on a backplane so that you can add up to four device boards to the system without the jumper wiring required with nonbackplane devices. You connect an SPL to the main JTAG scan path in the same way you connect any other device. Figure A–10 shows how to connect a secondary scan path to an SPL.

Figure A–10. Connecting a Secondary JTAG Scan Path to a Scan Path Linker



The $\overline{\text{TRST}}$ signal from the main scan path drives all devices, even those on the secondary scan paths of the SPL. The TCK signal on each target device on the secondary scan path of an SPL is driven by the SPL's DTCK signal. The TMS signal on each device on the secondary scan path is driven by the respective DTMS signals on the SPL.

DTDO0 on the SPL is connected to the TDI signal of the first device on the secondary scan path. DTDI0 on the SPL is connected to the TDO signal of the last device in the secondary scan path. Within each secondary scan path, the TDI signal of a device is connected to the TDO signal of the device before it. If the SPL is on a backplane, its secondary JTAG scan paths are on add-on boards; if signal degradation is a problem, you may need to buffer both the $\overline{\text{TRST}}$ and DTCK signals. Although degradation is less likely for DTMS*n* signals, you may also need to buffer them for the same reasons.

A.8.2 Emulation Timing Calculations for a Scan Path Linker (SPL)

Example A–3 and Example A–4 help you to calculate the key emulation timings in the SPL secondary scan path of your system. For actual target timing parameters, see the appropriate device data sheet for your target device.

The examples use the following assumptions:

$t_{su(TTMS)}$	Setup time, target TMS/TDI to TCK high	10 ns
$t_{d(TTDO)}$	Delay time, target TDO from TCK low	15 ns
$t_{d(bufmax)}$	Delay time, target buffer, maximum	10 ns
$t_{d(bufmin)}$	Delay time, target buffer, minimum	1 ns
$t_{(bufskew)}$	Skew time, target buffer, between two devices in the same package: $[t_{d(bufmax)} - t_{d(bufmin)}] \times 0.15$	1.35 ns
$t_{(TCKfactor)}$	Duty cycle, TCK assume a 40/60% clock	0.4 (40%)

Also, the examples use the following values from the SPL data sheet:

$t_{d(DTMSmax)}$	Delay time, SPL DTMS/DTDO from TCK low, maximum	31 ns
$t_{su(DTDLmin)}$	Setup time, DTDI to SPL TCK high, minimum	7 ns
$t_{d(DTCKHmin)}$	Delay time, SPL DTCK from TCK high, minimum	2 ns
$t_{d(DTCKLmax)}$	Delay time, SPL DTCK from TCK low, maximum	16 ns

There are two key timing paths to consider in the emulation design:

- The TCK-to-DTMS/DTDO path, called $t_{pd(TCK-DTMS)}$
- The TCK-to-DTDI path, called $t_{pd(TCK-DTDI)}$

Of the following two cases, the worst-case path delay is calculated to determine the maximum system test clock frequency.

Example A–3. Key Timing for a Single-Processor System Without Buffering (SPL)

$$t_{pd\,(TCK\text{-}DTMS)} = \frac{\left[t_{d\,(DTMSmax)} + t_{d\,(DTCKHmin)} + t_{su\,(TTMS)}\right]}{t_{TCKfactor}}$$

$$= \frac{(31\ ns + 2\ ns + 10\ ns)}{0.4}$$

$$= 107.5\ ns,\ or\ 9.3\ MHz$$

$$t_{pd\,(TCK\text{-}DTDI)} = \frac{\left[t_{d\,(TTDO)} + t_{d\,(DTCKLmax)} + t_{su\,(DTDLmin)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\ ns + 16\ ns + 7\ ns)}{0.4}$$

$$= 9.5\ ns,\ or\ 10.5\ MHz$$

In this case, the TCK-to-DTMS/DTDL path is the limiting factor.

Example A–4. Key Timing for a Single- or Multiprocessor-System With Buffered Input and Output (SPL)

$$t_{pd\,(TCK\text{-}TDMS)} = \frac{\left[t_{d\,(DTMSmax)} + t_{(DTCKHmin)} + t_{su\,(TTMS)} + t_{(bufskew)}\right]}{t_{TCKfactor}}$$

$$= \frac{(31\ ns + 2\ ns + 10\ ns + 1.35\ ns)}{0.4}$$

$$= 110.9\ ns,\ or\ 9.0\ MHz$$

$$t_{pd\,(TCK\text{–}DTDI)} = \frac{\left[t_{d\,(TTDO)} + t_{d\,(DTCKLmax)} + t_{su\,(DTDLmin)} + t_{d\,(bufskew)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\ ns + 15\ ns + 7\ ns + 10\ ns)}{0.4}$$

$$= 120\ ns,\ or\ 8.3\ MHz$$

In this case, the TCK-to-DTDI path is the limiting factor.

A.8.3 Using Emulation Pins

The EMU0/1 pins of TI devices are bidirectional, 3-state output pins. When in an inactive state, these pins are at high impedance. When the pins are active, they provide one of two types of output:

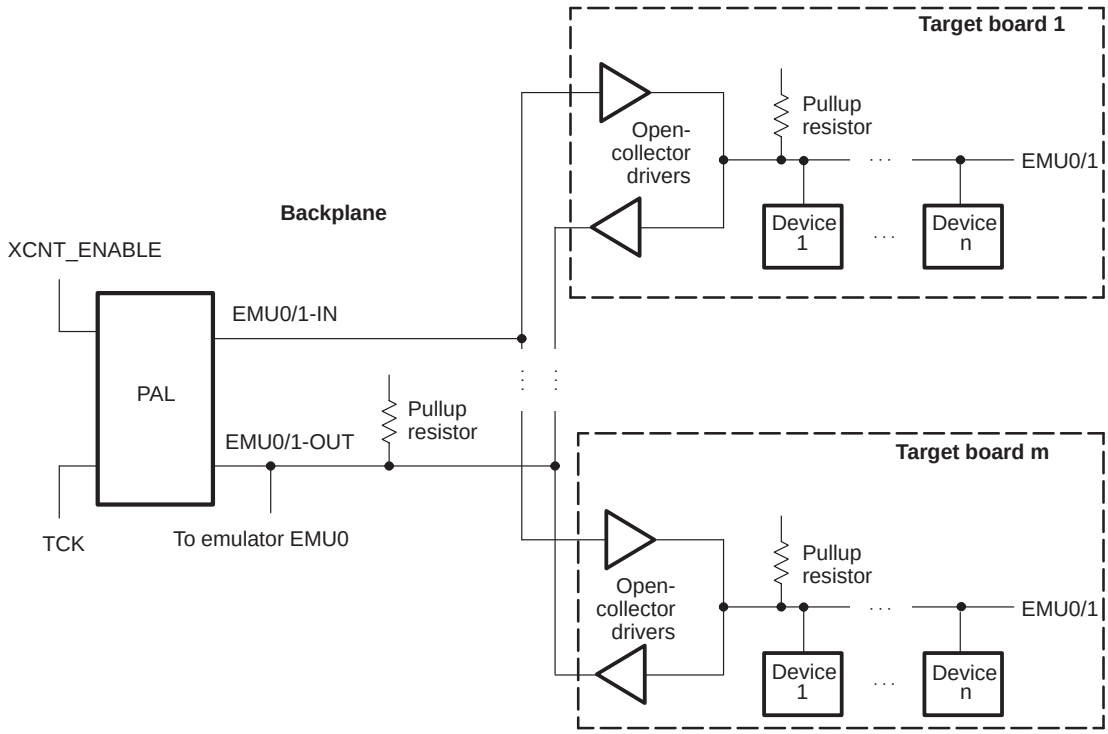
- Signal Event. The EMU0/1 pins can be configured via software to signal internal events. In this mode, driving one of these pins low can cause devices to signal such events. To enable this operation, the EMU0/1 pins function as open-collector sources. External devices such as logic analyzers can also be connected to the EMU0/1 signals in this manner. If such an external source is used, it must also be connected via an open-collector source.

- External Count. The EMU0/1 pins can be configured via software as totem-pole outputs for driving an external counter. If the output of more than one device is configured for totem-pole operation, then these devices can be damaged. The emulation software detects and prevents this condition. However, the emulation software has no control over external sources on the EMU0/1 signal. Therefore, all external sources must be inactive when any device is in the external count mode.

TI devices can be configured by software to halt processing if their EMU0/1 pins are driven low. This feature combined with the signal event output, allows one TI device to halt all other TI devices on a given event for system-level debugging.

If you route the EMU0/1 signals between multiple boards, they require special handling because they are more complex than normal emulation signals. Figure A–11 shows an example configuration that allows any processor in the system to stop any other processor in the system. Do not tie the EMU0/1 pins of more than 16 processors together in a single group without using buffers. Buffers provide the crisp signals that are required during a RUNB (run benchmark) debugger command or when the external analysis counter feature is used.

Figure A–11. EMU0/1 Configuration to Meet Timing Requirements of Less Than 25 ns



Notes: 1) The low time on EMU0/1-IN should be at least one TCK cycle and less than 10 μs. Software sets the EMU0/1-OUT pin to a high state.

2) To enable the open-collector driver and pullup resistor on EMU1 to provide rise/fall times of less than 25 ns, the modification shown in this figure is suggested. Rise times of more than 25 ns can cause the emulator to detect false edges during the RUNB command or when the external counter selected from the debugger analysis menu is used.

These seven important points apply to the circuitry shown in Figure A–11 and the timing shown in Figure A–12:

- ❑ Open-collector drivers isolate each board. The EMU0/1 pins are tied together on each board.
- ❑ At the board edge, the EMU0/1 signals are split to provide both input and output connections. This is required to prevent the open-collector drivers from acting as latches that can be set only once.
- ❑ The EMU0/1 signals are bused down the backplane. Pullup resistors must be installed as required.

- The bused EMU0/1 signals go into a programmable logic array device PAL® whose function is to generate a low pulse on the EMU0/1-IN signal when a low level is detected on the EMU0/1-OUT signal. This pulse must be longer than one TCK period to affect the devices but less than 10 µs to avoid possible conflicts or retriggering once the emulation software clears the device's pins.
- During a RUNB debugger command or other external analysis count, the EMU0/1 pins on the target device become totem-pole outputs. The EMU1 pin is a ripple carry-out of the internal counter. EMU0 becomes a *processor-halted* signal. During a RUNB or other external analysis count, the EMU0/1-IN signal to all boards must remain in the high (disabled) state. You must provide some type of external input (XCNT_ENABLE) to the PAL® to disable the PAL® from driving EMU0/1-IN to a low state.
- If you use sources other than TI processors (such as logic analyzers) to drive EMU0/1, their signal lines must be isolated by open-collector drivers and be inactive during RUNB and other external analysis counts.
- You must connect the EMU0/1-OUT signals to the emulation header or directly to a test bus controller.

Figure A–12. Suggested Timings for the EMU0 and EMU1 Signals

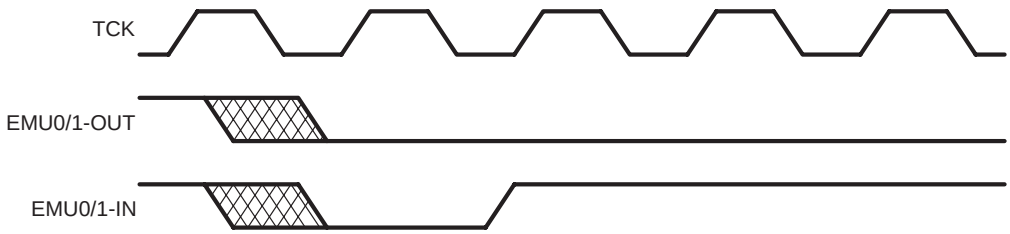
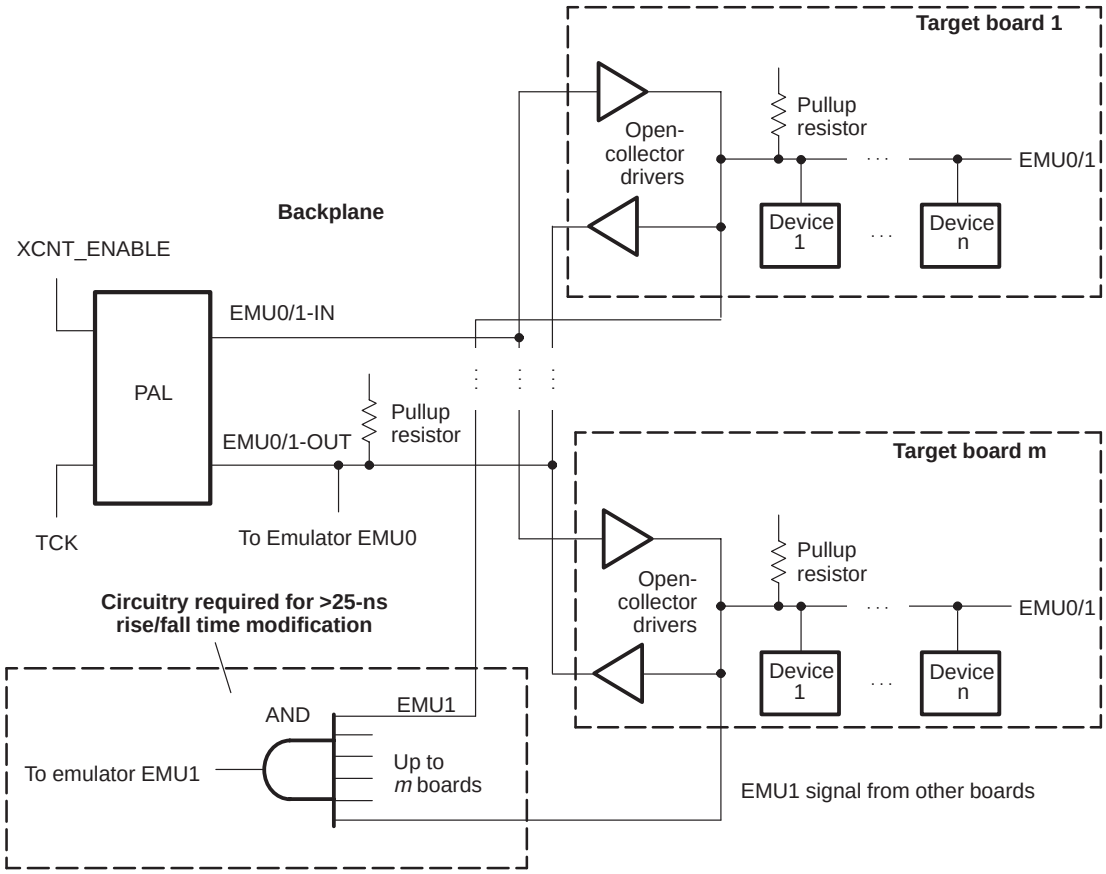


Figure A–13. EMU0/1 Configuration With Additional AND Gate to Meet Timing Requirements of Greater Than 25 ns

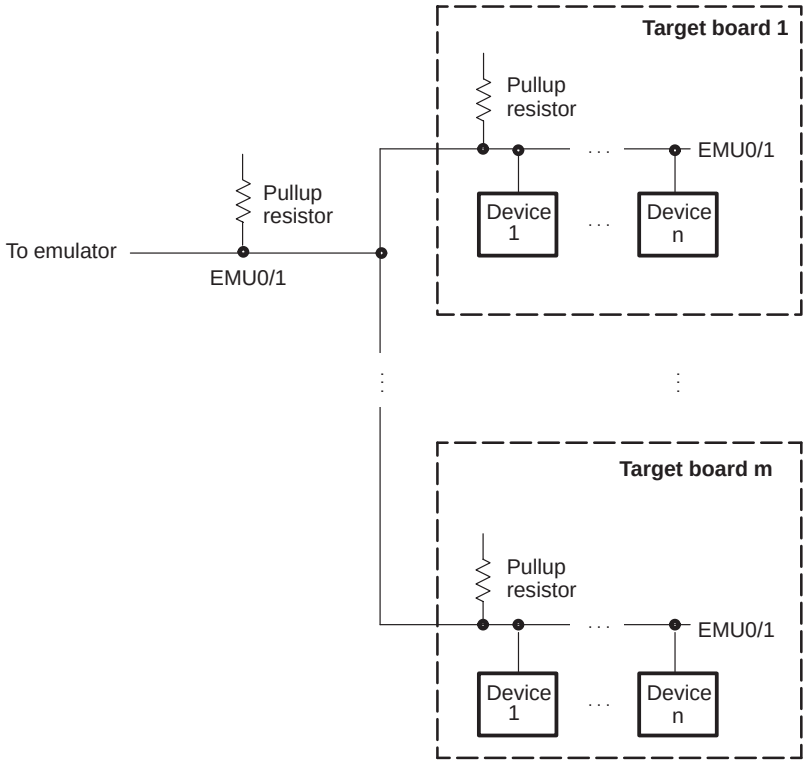


Notes: 1) The low time on EMU0/1-IN should be at least one TCK cycle and less than 10 μs. Software will set the EMU0/1-OUT port to a high state.

2) To enable the open-collector driver and pullup resistor on EMU1 to provide rise/fall time of greater than 25 ns, the modification shown in this figure is suggested. Rise times of more than 25 ns can cause the emulator to detect false edges during the RUNB command or when the external counter selected from the debugger analysis menu is used.

You do not need to have devices on one target board stop devices on another target board using the EMU0/1 signals (see the circuit in Figure A–14). In this configuration, the global-stop capability is lost. It is important not to overload EMU0/1 with more than 16 devices.

Figure A–14. EMU0/1 Configuration Without Global Stop

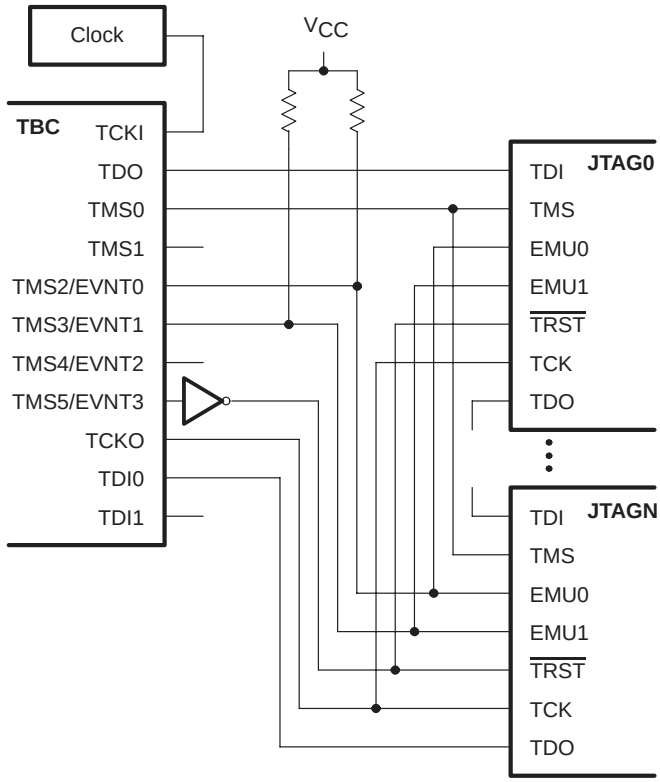


Note: The open-collector driver and pullup resistor on EMU1 must be able to provide rise/fall times of less than 25 ns. Rise times of more than 25 ns can cause the emulator to detect false edges during the RUNB command or when the external counter selected from the debugger analysis menu is used. If this condition cannot be met, then the EMU0/1 signals from the individual boards must be ANDed together (as shown in Figure A–14) to produce an EMU0/1 signal for the emulator.

A.8.4 Performing Diagnostic Applications

For systems that require built-in diagnostics, it is possible to connect the emulation scan path directly to a TI ACT8990 test bus controller (TBC) instead of the emulation header. The TBC is described in the Texas Instruments *Advanced Logic and Bus Interface Logic Data Book*. Figure A–15 shows the scan path connections of *n* devices to the TBC.

Figure A–15. TBC Emulation Connections for n JTAG Scan Paths



In the system design shown in Figure A–15, the TBC emulation signals TCKI, TDO, TMS0, TMS2/EVNT0, TMS3/EVNT1, TMS5/EVNT3, TCKO, and TDI0 are used, and TMS1, TMS4/EVNT2, and TDI1 are not connected. The target devices' EMU0 and EMU1 signals are connected to V_{CC} through pullup resistors and tied to the TBC's TMS2/EVNT0 and TMS3/EVNT1 pins, respectively. The TBC's TCKI pin is connected to a clock generator. The TCK signal for the main JTAG scan path is driven by the TBC's TCKO pin.

On the TBC, the TMS0 pin drives the TMS pins on each device on the main JTAG scan path. TDO on the TBC connects to TDI on the first device on the main JTAG scan path. TDI0 on the TBC is connected to the TDO signal of the last device on the main JTAG scan path. Within the main JTAG scan path, the TDI signal of a device is connected to the TDO signal of the device before it. $\overline{\text{TRST}}$ for the devices can be generated either by inverting the TBC's TMS5/EVNT3 signal for software control or by logic on the board itself.

Appendix B

Development Support and Part Order Information

This appendix provides development support information, device part numbers, and support tool ordering information for the ’54x.

Each ’54x support product is described in the *TMS320 DSP Development Support Reference Guide*. In addition, more than 100 third-party developers offer products that support the TI TMS320 family. For more information, refer to the *TMS320 Third-Party Support Reference Guide*.

For information on pricing and availability, contact the nearest TI Field Sales Office or authorized distributor. See the list at the back of this book.

Topic **Page**

B.1 Development Support

This section describes the development support provided by Texas Instruments.

B.1.1 Development Tools

TI offers an extensive line of development tools for the ’54x generation of DSPs, including tools to evaluate the performance of the processors, generate code, develop algorithm implementations, and fully integrate and debug software and hardware modules.

Code Generation Tools

- The optimizing ANSI C compiler translates ANSI C language directly into highly optimized assembly code. You can then assemble and link this code with the TI assembler/linker, which is shipped with the compiler. This product is currently available for PCs (DOS, DOS extended memory, OS/2), HP workstations, and SPARC workstations. See the *TMS320C54x Optimizing C Compiler User’s Guide* for detailed information about this tool.
- The assembler/linker converts source mnemonics to executable object code. This product is currently available for PCs (DOS, DOS extended memory, OS/2). The ’54x assembler for HP and SPARC workstations is available only as part of the optimizing ’54x compiler. See the *TMS320C54x Assembly Language Tools User’s Guide* for detailed information about available assembly-language tools.

System Integration and Debug Tools

- The simulator simulates (via software) the operation of the ’54x and can be used in C and assembly software development. This product is currently available for PCs (DOS, Windows), HP workstations, and SPARC workstations. See the *TMS320C54x C Source Debugger User’s Guide* for detailed information about the debugger.
- The XDS510 emulator performs full-speed in-circuit emulation with the ’54x, providing access to all registers as well as to internal and external memory of the device. It can be used in C and assembly software development and has the capability to debug multiple processors. This product is currently available for PCs (DOS, Windows, OS/2), HP workstations, and SPARC workstations. This product includes the emulator board (emulator box, power supply, and SCSI connector cables in the HP and SPARC versions), the ’54x C source debugger and the JTAG cable.

Because the ’C2xx, ’C3x, ’C4x, and ’C5x XDS510 emulators also come with the same emulator board (or box) as the ’54x, you can buy the ’54x C Source Debugger Software as a separate product called the ’54x C Source Debugger Conversion Software. This enables you to debug ’54x applications with a previously purchased emulator board. The emulator cable that comes with the ’C3x XDS510 emulator cannot be used with the ’54x. You need the JTAG emulation conversion cable (see Section B.2) instead. The emulator cable that comes with the ’C5x XDS510 emulator can be used with the ’54x without any restriction. See the *TMS320C54x C Source Debugger User’s Guide*) for detailed information about the ’54x emulator.

- The TMS320C54x evaluation module (EVM) is a PC/AT plug-in card that lets you evaluate certain characteristics of the ’54x digital signal processor to see if it meets your application requirements. The ’54x EVM carries a ’541 DSP on board to allow full-speed verification of ’54x code. The EVM has 5K bytes of on-chip program/data RAM, 28K bytes of on-chip ROM, two serial ports, a timer, access to 64K bytes each of external program and data RAM, and an external analog interface for evaluation of the ’54x family of devices for applications. See the *TMS320C54x Evaluation Module Technical Reference* for detailed information about the ’54x EVM.

B.1.2 Third-Party Support

The TMS320 family is supported by products and services from more than 100 independent third-party vendors and consultants. These support products take various forms (both as software and hardware), from cross-assemblers, simulators, and DSP utility packages to logic analyzers and emulators. The expertise of those involved in support services ranges from speech encoding and vector quantization to software/hardware design and system analysis.

To ask about third-party services, products, applications, and algorithm development packages, contact the third party directly. Refer to the *TMS320 Third-Party Support Reference Guide* for addresses and phone numbers.

B.1.3 Technical Training Organization (TTO) TMS320 Workshops

'54x DSP Design Workshop. This workshop is tailored for hardware and software design engineers and decision-makers who will be designing and utilizing the '54x generation of DSP devices. Hands-on exercises throughout the course give participants a rapid start in developing '54x design skills. Microprocessor/assembly language experience is required. Experience with digital design techniques and C language programming experience is desirable.

These topics are covered in the '54x workshop:

- '54x architecture/instruction set
- Use of the PC-based software simulator
- Use of the '54x assembler/linker
- C programming environment
- System architecture considerations
- Memory and I/O interfacing
- Development support

For registration information, pricing, or to enroll, call (800)336–5236, ext. 3904.

B.1.4 Assistance

For assistance to TMS320 questions on device problems, development tools, documentation, software upgrades, and new products, you can contact TI. See *If You Need Assistance* in *Preface* for information.

B.2 Part Order Information

This section describes the part numbers of ’54x devices, development support hardware, and software tools.

B.2.1 Device and Development Support Tool Nomenclature Prefixes

To designate the stages in the product development cycle, TI assigns prefixes to the part numbers of all TMS320 devices and support tools. Each TMS320 device has one of three prefix designators: TMX, TMP, or TMS. Each support tool has one of two possible prefix designators: TMDX or TMDS. These prefixes represent evolutionary stages of product development from engineering prototypes (TMX/TMDX) through fully qualified production devices and tools (TMS/TMDS). This development flow is defined below.

Device Development Evolutionary Flow:

TMX The part is an experimental device that is not necessarily representative of the final device’s electrical specifications.

TMP The part is a device from a final silicon die that conforms to the device’s electrical specifications but has not completed quality and reliability verification.

TMS The part is a fully qualified production device.

Support Tool Development Evolutionary Flow:

TMDX The development-support product that has not yet completed Texas Instruments internal qualification testing.

TMDS The development-support product is a fully qualified development support product.

TMX and TMP devices and TMDX development support tools are shipped with the following disclaimer:

“Developmental product is intended for internal evaluation purposes.”

TMS devices and TMDS development support tools have been fully characterized, and the quality and reliability of the device has been fully demonstrated. Texas Instruments standard warranty applies to these products.

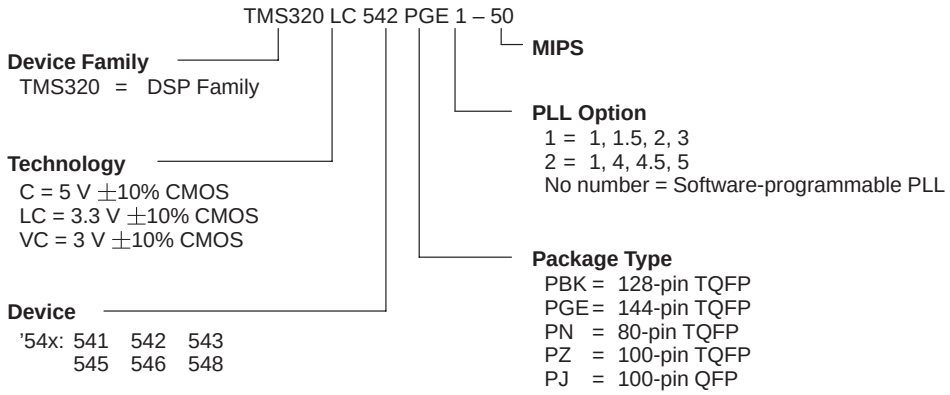
Note:

It is expected that prototype devices (TMX or TMP) have a greater failure rate than standard production devices. Texas Instruments recommends that these devices *not* be used in any production system, because their expected end-use failure rate is still undefined. Only qualified production devices should be used.

B.2.2 Device Nomenclature

TI device nomenclature includes the device family name and a suffix. Figure B–1 provides a legend for reading the complete device name for any ’54x device family member.

Figure B–1. TMS320C54x Device Nomenclature



B.2.3 Development Support Tools

Table B–1 lists the development support tools available for the ’54x, the platform on which they run, and their part numbers.

Table B–1. Development Support Tools Part Numbers

Development Tool	Platform	Part Number
Assembler/Linker	PC (DOS™)	TMDS324L850-02
C Compiler/Assembler/Linker	PC (DOS™, Windows™, OS/2™)	TMDS324L855-02
C Compiler/Assembler/Linker	HP (HP-UX™) / SPARC™ (Sun OS™)	TMDS324L555-08
C Source Debugger Conversion Software	PC (DOS™, Windows™, OS/2™) (XDS510™)	TMDS32401L0
C Source Debugger Conversion Software	HP (HP-UX™) / SPARC™ (Sun OS™) (XDS510WS™)	TMDS32406L0
Evaluation Module (EVM)	PC (DOS™, Windows™, OS/2™)	TMDX3260051
Simulator (C language)	PC (DOS™, Windows™)	TMDS324L851-02
Simulator (C language)	HP (HP-UX™) / SPARC™ (Sun OS™)	TMDS324L551-09
XDS510 Emulator†	PC (DOS™, Windows™, OS/2™)	TMDS00510
XDS510WS Emulator‡	HP (HP-UX™) / SPARC™ (Sun OS™) (SCSI)	TMDS00510WS
3 V/5 V PC/SPARC JTAG Emulation Cable	XDS510™ / XDS510WS™	TMDS3080002

† Includes XDS510 board and JTAG cable; TMDS32401L0 C-source debugger conversion software not included
‡ Includes XDS510WS box, SCSI cable, power supply, and JTAG cable; TMDS32406L0 C-source debugger conversion software not included

Appendix C

Glossary

A

A: See *accumulator A*.

accumulator: A register that stores the results of an operation and provides an input for subsequent arithmetic logic unit (ALU) operations.

accumulator A: 40-bit register that stores the result of an operation and provides an input for subsequent arithmetic logic unit (ALU) operations.

accumulator B: 40-bit registers that stores the result of an operation and provides an input for subsequent arithmetic logic unit (ALU) operations.

adder: A unit that adds or subtracts two numbers.

address: The location of a word in memory.

address bus: A group of connections used to route addresses. The '54x has four 16-bit address busses: CAB, DAB, EAB, and PAB.

addressing mode: The method by which an instruction calculates the location of an object in memory.

address visibility mode bit (AVIS): A bit in processor mode status register (PMST) that determines whether or not the internal program address appears on the device's external address bus pins.

ALU: *arithmetic logic unit*. The part of the CPU that performs arithmetic and logic operations.

analog-to-digital (A/D) converter: Circuitry that translates an analog signal to a digital signal.

AR0–AR7: *auxiliary registers 0–7*. Eight 16-bit registers that can be accessed by the CPU and modified by the auxiliary register arithmetic units (ARAUs) and are used primarily for data memory addressing.

ARAU: See *auxiliary register arithmetic unit*.

ARP: See *auxiliary register pointer*.

ASM: See *accumulator shift mode field*.

auxiliary register arithmetic unit: An unsigned, 16-bit arithmetic logic unit (ALU) used to calculate indirect addresses using auxiliary registers.

auxiliary register file: The area in data memory containing the eight 16-bit auxiliary registers. See also *auxiliary registers*.

auxiliary register pointer (ARP): A 3-bit field in status register 0 (ST0) used as a pointer to the currently-selected auxiliary register, when the device is operating in 'C5x/'C2xx compatibility mode.

auxiliary registers: Eight 16-bit registers (AR7 – AR0) that are used as pointers to an address within data space. These registers are operated on by the auxiliary register arithmetic units (ARAUs) and are selected by the auxiliary register pointer (ARP). See also *auxiliary register arithmetic unit*.

AVIS: See *address visibility mode bit*.

B

B: See *accumulator B*.

bank-switching control register (BSCR): A 16-bit register that defines the external memory bank size and enables or disables automatic insertion of extra cycles when accesses cross memory bank boundaries.

barrel shifter: A unit that rotates bits in a word.

BK: See *circular buffer size register*.

block-repeat active flag (BRAF): A bit in status register 1 (ST1) that indicates whether or not a block repeat is currently active.

block-repeat counter (BRC): A 16-bit register that specifies the number of times a block of code is to be repeated when a block repeat is performed.

block-repeat end address register (REA): A 16-bit memory-mapped register containing the end address of a code segment being repeated.

block-repeat start address register (RSA): A 16-bit memory-mapped register containing the start address of a code segment being repeated.

boot: The process of loading a program into program memory.

boot loader: A built-in segment of code that transfers code from an external source to program memory at power-up.

BRC: See *block-repeat counter*.

BSCR: See *bank-switching control register*.

BSP: *buffered serial port*. An enhanced synchronous serial port that includes an autobuffering unit (ABU) that reduces CPU overhead in performing serial operations.

BSPCE: *BSP control extension register*. A 16-bit memory-mapped register that contains status and control bits for the buffered serial port (BSP) interface. The 10 LSBs of the SPCE are dedicated to serial port interface control, whereas the 6 MSBs are used for autobuffering unit (ABU) control.

butterfly: A kernel function for computing an N-point fast Fourier transform (FFT), where N is a power of 2. The combinational pattern of inputs resembles butterfly wings.

C

C: See *carry bit*.

C16: A bit in status register 1 (ST1) that determines whether the ALU operates in dual 16-bit mode or in double-precision mode.

carry bit: A bit in status register 0 (ST0) used by the ALU in extended arithmetic operations and accumulator shifts and rotates. The carry bit can be tested by conditional instructions.

circular buffer size register (BK): A 16-bit register used by the auxiliary register arithmetic units (ARAUs) to specify the data-block size in circular addressing.

code: A set of instructions written to perform a task; a computer program or part of a program.

cold boot: The process of loading a program into program memory at power-up.

compare, select, and store unit (CSSU): An application-specific hardware unit dedicated to add/compare/select operations of the Viterbi operator.

CSSU: *See compare, select, and store unit*

D

DAGEN: See *data-address generation logic (DAGEN)*.

DARAM: *dual-access RAM*. Memory that can be read from and written to in the same clock cycle.

data address bus: A group of connections used to route data memory addresses. The ’54x has three 16-bit buses that carry data memory addresses: *CAB*, *DAB*, and *EAB*.

data-address generation logic (DAGEN): Logic circuitry that generates the addresses for data memory reads and writes. See also *program-address generation logic (PAGEN)*.

data bus: A group of connections used to route data. The ’54x has three 16-bit data buses: CB, DB, and EB.

data memory: A memory region used for storing and manipulating data. Addresses 00h–1Fh of data memory contain CPU registers. Addresses 20h–5Fh of data memory contain peripheral registers.

data page pointer (DP): A 9-bit field in status register 0 (ST0) that specifies which of 512, 128 × 16 word pages is currently selected for direct address generation. DP provides the nine MSBs of the data-memory address; the dma provides the lower seven. See also *dma.*

data ROM bit (DROM): A bit in PMST that determines whether or not part of the on-chip ROM is mapped into data space.

digital-to-analog (D/A) converter: Circuitry that translates a digital signal to an analog signal.

direct data-memory address bus: A 16-bit bus that carries the direct address for data memory.

direct memory address (dma, DMA) : The seven LSBs of a direct-addressed instruction that are concatenated with the data page pointer (DP) to generate the entire data memory address. See also *data page pointer.*

dma: *See direct memory address*.

DP: *See data page pointer.*

DRB: *direct data-memory address bus*. A 16-bit bus that carries the direct address for data memory.

DROM: *See data ROM bit*.

E

exponent encoder (EXP): An application-specific hardware device that computes the exponent value of the accumulator.

external interrupt: A hardware interrupt triggered by a pin ($\overline{\text{INT0}}$–$\overline{\text{INT3}}$).

F

fast Fourier transform (FFT): An efficient method of computing the discrete Fourier transform, which transforms functions between the time domain and frequency domain. The time-to-frequency domain is called the forward transform, and the frequency-to-time domain is called the inverse transformation. See also *butterfly*.

fast return register (RTN): A 16-bit register used to hold the return address for the fast return from interrupt (RETF[D]) instruction.

G

general-purpose input/output pins: Pins that can be used to supply input signals from an external device or output signals to an external device. These pins are not linked to specific uses; rather, they provide input or output signals for a variety of purposes. These pins include the general-purpose $\overline{\text{BIO}}$ input pin and XF output pin.

H

hardware interrupt: An interrupt triggered through physical connections with on-chip peripherals or external devices.

host port interface (HPI): An 8-bit parallel interface that the CPU uses to communicate with a host processor.

HPI control register (HPIC): A 16-bit register that contains status and control bits for the host port interface (HPI).

I

IFR: See *interrupt flag register*.

IMR: See *interrupt mask register*.

interrupt: A condition caused either by an event external to the CPU or by a previously executed instruction that forces the current program to be suspended and causes the processor to execute an interrupt service routine corresponding to the interrupt.

interrupt flag register (IFR): A 16-bit memory-mapped register that flags pending interrupts.

interrupt mask register (IMR): A 16-bit memory-mapped register that masks external and internal interrupts.

interrupt mode bit (INTM): A bit in status register 1 (ST1) that globally masks or enables all interrupts.

interrupt service routine (ISR): A module of code that is executed in response to a hardware or software interrupt.

IPTR: *interrupt vector pointer.* A 9-bit field in the processor mode status register (PMST) that points to the 128-word page where interrupt vectors reside.

IR: *instruction register.* A 16-bit register used to hold a fetched instruction.

L

latency: The delay between when a condition occurs and when the device reacts to the condition. Also, in a pipeline, the necessary delay between the execution of two instructions to ensure that the values used by the second instruction are correct.

LSB: *least significant bit.* The lowest order bit in a word.

M

maskable interrupts: A hardware interrupt that can be enabled or disabled through software.

memory map: A map of the addressable memory space accessed by the ’54x processor partitioned according to functionality (memory, registers, etc.).

memory-mapped registers: The ’54x processor registers mapped into page 0 of the data memory space.

microcomputer mode: A mode in which the on-chip ROM is enabled and addressable for program accesses.

microprocessor/microcomputer (MP/$\overline{\text{MC}}$) bit: A bit in the processor mode status register (PMST) that indicates whether the processor is operating in microprocessor or microcomputer mode. See also *microcomputer mode; microprocessor mode.*

microprocessor mode: A mode in which the on-chip ROM is disabled for program accesses.

micro stack: A stack that provides temporary storage for the address of the next instruction to be fetched when the program address generation logic is used to generate sequential addresses in data space.

MSB: *most significant bit*. The highest order bit in a word.

multiplier: A 17-bit × 17-bit multiplier that generates a 32-bit product. The multiplier executes multiple operations in a single cycle and operates using either signed or unsigned 2s-complement arithmetic.

N

nested interrupt: A higher-priority interrupt that must be serviced before completion of the current interrupt service routine (ISR). An executing ISR can set the interrupt mask register (IMR) bits to prevent being suspended by another interrupt.

nonmaskable interrupt: An interrupt that can be neither masked by the interrupt mask register (IMR) nor disabled by the INTM bit of status register 1 (ST1).

O

overflow: A condition in which the result of an arithmetic operation exceeds the capacity of the register used to hold that result.

overflow flag: A flag that indicates whether or not an arithmetic operation has exceeded the capacity of the corresponding register.

P

PAGEN: See *program-address generation logic (PAGEN)*.

PAR: *program address register*. A 16-bit register used to address the program-memory operands in FIRS, MACD, MACP, MVDP, MVPD, READA, and WRITA instructions.

PC: *program counter.* A 16-bit register that indicates the location of the next instruction to be executed.

pipeline: A method of executing instructions in an assembly-line fashion.

pma: *program memory address.* A register that provides the address of a multiplier operand that is contained in program memory.

PMST: *processor mode status register*. A 16-bit status register that controls the memory configuration of the device. See also *ST0, ST1.*

pop: Action of removing a word from a stack.

program-address generation logic (PAGEN): Logic circuitry that generates the address for program memory reads and writes, and the address for data memory in instructions that require two data operands. This circuitry can generate one address per machine. See also *data-address generation logic (DAGEN)*.

program counter (PC): A 16-bit register that indicates the location of the next instruction to be executed.

program controller: Logic circuitry that decodes instructions, manages the pipeline, stores status of operations, and decodes conditional operations.

program memory: A memory region used for storing and executing programs.

push: Action of placing a word onto a stack.

R

RC: *repeat counter*. A 16-bit register used to specify the number of times a single instruction is executed.

register: A group of bits used for temporarily holding data or for controlling or specifying the status of a device.

repeat counter (RC): A 16-bit register used to specify the number of times a single instruction is executed.

reset: A means of bringing the CPU to a known state by setting the registers and control bits to predetermined values and signaling execution to start at a specified address.

REA: *block-repeat end address*. A 16-bit register that specifies the end address of a code segment to be repeated in repeat mode.

RSA: *block-repeat start address*. A 16-bit register that specifies the start address of a code segment to be repeated in repeat mode.

RTN: *fast return register*. A 16-bit register used to hold the return address for the fast return from interrupt (RETF[D]) instruction.

S

SARAM: *single-access RAM*. Memory that can be read written once during one clock cycle.

serial port interface: An on-chip full-duplex serial port interface that provides direct serial communication to serial devices with a minimum of external hardware, such as codecs and serial analog-to-digital (A/D) and digital-to-analog (D/A) converters. Status and control of the serial port is specified in the serial port control register (SPC).

shifter: A hardware unit that shifts bits in a word to the left or to the right.

sign-control logic: Circuitry used to extend data bits (signed/unsigned) to match the input data format of the multiplier, ALU, and shifter.

sign extension: An operation that fills the high order bits of a number with the sign bit.

software interrupt (SINT): An interrupt caused by the execution of an INTR or TRAP instruction.

software wait-state register (SWWSR): *software wait-state register*. A 16-bit register that selects the number of wait states for the program, data, and I/O spaces of off-chip memory.

SP: *stack pointer*. A register that always points to the last element pushed onto the stack.

ST0: A 16-bit register that contains ’54x status and control bits. See also *PMST; ST1.*

ST1: A16-bit register that contains ’54x status and control bits. See also *PMST, ST0.*

stack: A block of memory used for storing return addresses for subroutines and interrupt service routines and for storing data.

stack pointer (SP): A register that always points to the last element pushed onto the stack.

T

temporary register (T): A 16-bit register that holds one of the operands for multiply operations, the dynamic shift count for the LACT, ADDT, and SUBT instructions, or the dynamic bit position for the BITT instruction.

time-division multiplexing (TDM): The process by which a single serial bus is shared by up to eight ’54x devices with each device taking turns to communicate on the bus. There are a total of eight time slots (channels) available. During a time slot, a given device may talk to any combination of devices on the bus.

transition register (TRN): A 16-bit register that holds the transition decision for the path to new metrics to perform the Viterbi algorithm.

TSPC: *TDM serial port control register.* A 16-bit memory-mapped register that contains status and control bits for the TDM serial port.

W

wait state: A period of time that the CPU must wait for external program, data, or I/O memory to respond when reading from or writing to that external memory. The CPU waits one extra cycle (one CLKOUT1 cycle) for every wait state.

warm boot: The process by which the processor transfers control to the entry address of a previously-loaded program.

X

XF: *XF status flag.* A bit in status register ST1 that indicates the status of the XF pin.

XPC: *program counter extension.* A register that contains the upper 7 bits of the current program memory address.

Z

zero fill: A method of filling the low or high order bits with zeros when loading a 16-bit number into a 32-bit field.

Index

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z