Hardware Interfacing 12

12.1 OVERVIEW

This chapter describes several hardware interface solutions for connecting ADSP-2100 Family digital signal processors to peripheral devices, such as codecs. Because the peripheral devices are also programmable, this chapter includes the code for configuring these devices. The corresponding subroutines are listed at the end of each section.

This chapter includes the following hardware interface solutions:

- ADSP-2105 / AD1849 SoundPort®
- ADSP-2111 / AD1849 SoundPort
- ADSP-2101 / AD1847 SoundPort
- ADSP-2100 Family / DRAM Interfacing
- Loading an ADSP-2101 Program Through the Serial Port
- Memory Interfacing with the ADSP-2105

Although most sections of this chapter contain solutions for specific ADSP-2100 Family processor hardware interfaces, the code listings can be modified to accommodate additional solutions. Several sections also include suggestions for modifying the code.

When you change the code to accommodate ADSP-2100 Family DSPs other than the ones specified, you must consider differences in interrupt vectors, chip architecture, peripheral devices, and memory configurations.

12.2 SOUNDPORT INTERFACES

This section contains three solutions for interfacing the ADSP-2100 Family DSPs with AD1849 and AD1847 SoundPorts.

12.2.1 ADSP-2111/AD1849 SoundPort Interface

This code in this section provides the initialization of the control functions needed to connect the AD1849 SoundPort Stereo Codec to the ADSP-2111. Listing 12.1 performs an ADSP-2111 / AD1849 talk-through routine.

You should consider the following points when using this program:

- The AD1849 works on 64-bit words. The DSP sees this as four 16-bit words. Be careful when you set up control words so that the appropriate control bits are in the correct locations.

- The source of SPORT signals is different between data and control modes. In *control mode*, the DSP is expected to provide all signals. In *data mode*, the AD1849 provides the signals. You must reset the DSP SPORT when you change from control mode to data mode.

- When you change sampling rates (or other system parameters), you must change the appropriate control field in the command word *and* in the mask word used to check DCB status. Also, remember that the AD1849 initiates its auto calibration sequence once the sampling rate is changed.

- If you retain indirect addressing into the receive and transmit autobuffers, the I3 pointer must be set immediately after the code enters the interrupt routine. A delay can cause pointer misalignment and render the code inoperable.

The following suggestions are included for modifying the code:

- Direct Addressing: Instead of resetting an indirect pointer into the receive and transmit buffers, try using direct addressing. For example, *dm(datain)* is the left channel data, *dm(datain+1)* is the right channel data.

- Transmit Interrupt: Because of interrupt timing and other system constraints, it may be advantageous to time your system from the transmit interrupt instead of the receive interrupt. Refer to the *ADSP-2100 Family User's Manual* for more information about the different interrupt latencies that are inherent with the different approaches.

```
{************************************************************************
ADSP-2111 - AD1849(SOUNDPORT)  INTERFACE PROGRAM
Analog Devices  Sept, 1991

This program is written to run on an ADSP-2111. However, it can be easily
changed to work with an ADSP-2101 by modifying the interrupt vector table.

For the hardware connections between the ADSP-2111 and AD1849 please refer to
the diagrams shown on the AD1849 data sheet.

The AD1849 will be set up as follows:
(NC indicates a no care state, all control reg are 1 bit unless indicated
 by [#bits])

Control time slot control bits:
DCB=0       followed by 1
AC=1        autocalibrate
DFR=5       Data conversion frequency,44.1kH [3]
ST=1        stereo mode
DF=0        dataformat is 16 bit twos complement [2]
MCK=2       16.9344 MHz is the master clock [2]
FSEL=0      frame size, 64 bits [2]
MS=1        master mode (i.e. receive external serial clock)
TXDIS=0     enable serial output
ENL=0       disable loopback testing
ADL=NC      loopback mode analog/digital (disabled)
PIO=NC      paralell I/O bits not used [2]
REVID=NC

Data Timeslot Control Bits:
OM0=1       enable line 0 output
OM1=1       enable line 1 output
LO=0        no attenuation of left channel output [6]
SM=0        mono output is muted
RO=0        no attenuation of right channel output [6]
PIO=NC      paralell I/O bits not used [2]
OVR=NC      overrange INPUT
IS=0        line-level stereo input selected
LG=0        no gain for left channel [4]
MA=15       no monitor mix (i.e. ADC output is not mixed with DAC input
RG=0        no gain for right channel [4]

************************************************************************

This program makes use of the multi-channel mode that is available on the
SPORT0 and it also uses autobuffering to reduce interrupt service overhead.
For a description of the multi-channel and autobuffering features, please
refer to the ADSP-2111 or ADSP-2101 architecture user's manuals.
```

(listing continues on next page)

12 Hardware Interfacing

```
In its current condition, this program can be used “as is” to perform straight
talkthrough (at 44.1 KHz sample rate) on both left and right channels of the
AD1849. The incoming audio data from the 16bit ADCs is placed in a short
buffer and immediately sent out to the 16bit DACs.

The initial setup and handshaking with the AD1849 starts with the “START”
routine. A state machine to perform the handshaking is executed in software.
Once the AD1849 is configured properly, the processor enters the WAIT_DATA
loop and waits for serial port interrupt requests.

This program is booted into the ADSP-2111 on power-up.

*************************************************************************}

.MODULE/ABS=0/BOOT=0    AD1849;
.VAR/CIRC   CTRLIN[4];              {circular buffers for data input and}
.VAR/CIRC   CTRLOUT[4];             {output for data mode and control mode}
.VAR/CIRC   DATAIN[4];
.VAR/CIRC   DATAOUT[8];
.VAR        FIRST_FLG;
.VAR        DCB_FLG;
.VAR        DMODE_FLG;

         JUMP START;nop;nop;nop;          {restart interrupt}
         RTI;nop;nop;nop;                 {IRQ2 int, not used}
         RTI;nop;nop;nop;                 {HIP write int, not used}
         RTI;nop;nop;nop;                 {HIP read int, not used}
         JUMP SETUPCONTROL;nop;nop;nop;   {SPORT0 transmit int}
         JUMP NEWDATA;nop;nop;nop;        {SPORT0 receive int}
         RTI;nop;nop;nop;                 {SPORT1 tx or IRQ1 int, not used}
         RTI;nop;nop;nop;                 {SPORT1 rx or IRQ0 int, not used}
         nop;nop;nop;nop;                 {timer int,not used}

START:   RESET FL0;                       {set the AD1849 D/C pin low}
         L0=%CTRLIN;
         M0=1;
         I0=^CTRLIN;
         L1=%CTRLOUT;
         M1=1;
         I1=^CTRLOUT;
         AX0=1;
         DM(FIRST_FLG)=AX0;
         DM(DCB_FLG)=AX0;
         AX0=0;
         DM(DMODE_FLG)=AX0;
         AY0=B#0010000100101100;
```

```
{Initialize contol mode output buffer}

         DM(I1,M1)= B#0010000100101100;   {DCB=0,AC=1,DFR=05,ST=1,DF=00}
         DM(I1,M1)= B#0010001000000000;   {MCK=02,FSEL=0,MS=0,TXDIS=0}
                                          {ENL=0,ADL=0}
         DM(I1,M1)= B#0000000000000000;   {PIO = 00}
         DM(I1,M1)= B#0000000000000000;   {REVID=NC}
         L2 = 0;                          {linear addressing for register }
         I2 = 0x3fef;                     {pt. to last DM cntrl reg}
{3FEF}   DM(I2,M1) = 0x0000;              {sport1 not used}
{3FF0}   DM(I2,M1) = 0x0000;
{3FF1}   DM(I2,M1) = 0x0000;
{3FF2}   DM(I2,M1) = 0x0000;
{3FF3}   DM(I2,M1) = 0X0283;         {autobuf. rx:i0, m0 tx:i1,m1}
{3FF4}   DM(I2,M1) = 383;            {sport rfsdiv, sets up FRAME sync freq}
{3FF5}   DM(I2,M1) = 849;            {sport0 sclkdiv, SCLK will be less than}
                                     {8KHz from control mode}
{3FF6}   DM(I2,M1) = B#1100010100011111;
                                     {sport0 control register:
                                     multi-channel mode, 24 channels
                                     internal sclk & rfs
                                     normal framing mode
                                     frame sync not inverted
                                     16 bit word length}
{3FF7}   DM(I2,M1) = 0x000F;         {first 4 xmit multichannels used}
{3FF8}   DM(I2,M1) = 0x0000;
{3FF9}   DM(I2,M1) = 0x000F;         {first 4 rx multichannels used}
{3FFA}   DM(I2,M1) = 0x0000;
{3FFB}   DM(I2,M1) = 0x0000;         {TSCALE register}
{3FFC}   DM(I2,M1) = 0x0000;         {TCOUNT register}
{3FFD}   DM(I2,M1) = 0x0000;         {TPERIOD register,initializing value
                                     for TCOUNT after every interrupt}
{3FFE}   DM(I2,M1) = 0x0000;         {external data memory waits=0}

         ICNTL=0x00;
         IMASK=B#00010000;           {only SPORT transmit intrpt enabled
                                     initially while in control mode}

         AX0=DM(I1,M1);              {send first 16bits of ctrl word}
         TX0=AX0;
{3FFF}   DM(I2,M1) = 0x1418;         {system control reg: sport0 enabled}
```

(listing continues on next page)

```
{Wait for an interrupt indicating that transmit register is ready for
 new data and that the 2111 has received a 16bit word}

WAIT1:   AX1=DM(DMODE_FLG);
         AR=PASS AX1;
         IF GT JUMP GO_DMODE;
         JUMP WAIT1;

SETUPCONTROL:
         AX0=DM(FIRST_FLG);
         AF=PASS AX0;
         IF NE JUMP DECR_FIRST;
         AX0=DM(DCB_FLG);
         AR=PASS AX0;
         IF EQ JUMP DCBFLG_SET;
         AX0=DM(CTRLIN);            {DCB_FLG has not been set yet}
         AR=AX0 XOR AY0;            {check all incoming bits including DCB bit}
         IF EQ JUMP SET_DCB;        {set flag if DCB was 0}
         RTI;

DCBFLG_SET:
         AX0=DM(CTRLIN);            {DCB_FLG was set}
         AR=AX0 AND AY0;            {only check for DCB bit}
         IF NE JUMP SETDMODE;       {if DBC=1 ready for datamode}
         RTI;

SET_DCB: AX0=0;
         DM(DCB_FLG)=AX0;
         AX0=B#0010010100101100;    {DCB was 0, prepare to send DCB=1}
         DM(CTRLOUT)=AX0;
         AY0=B#0000010000000000;
         RTI;

DECR_FIRST:
         AX0=0;
         DM(FIRST_FLG)=AX0;
         RTI;

SETDMODE:
         IMASK=0;
         AX0=0X0818;                {disable sport0}
         DM(0X3FFF)=AX0;

{ At this point we could boot page#1 and do the following data mode setup
  after the reboot occurs. This would free up some PM RAM for other uses}
         I1 = ^DATAOUT;
         L1=0;
         DM(I1,M1) = 0x0000;        {reset output & input data buffers}
         DM(I1,M1) = 0x0000;        {initialize embedded control bits}
         DM(I1,M1) = B#1100000000000000;  {OM1=1,OM1=1,LO=0,SM=1,RO=0}
         DM(I1,M1) = B#0000000011110000;
                                    {PIO=00,OVR=1,IS=0,LG=0,MA=15,RG=0}
```

```
          DM(I1,M1) = 0x0000;         {reset output & input data buffers}
          DM(I1,M1) = 0x0000;         {initialize embedded control bits}
          DM(I1,M1) = B#1100000000000000;  {OM1=1,OM1=1,LO=0,SM=1,RO=0}
          DM(I1,M1) = B#0000000011110000;  {PIO=00,OVR=1,IS=0,LG=0,MA=15,RG=0}
          AX0=0X861F;
          DM(0X3FF6)=AX0;             {sport0 control:
                                      multi-channel mode
                                      external sclk & rfs
                                      32 word frames, 16bit words}
          SET FL0;                    {set D/C high}
          AX0=1;
          DM(DMODE_FLG)=AX0;
          RTI;

GO_DMODE:
          L0=%DATAIN;
          I0=^DATAIN;
          L1=%DATAOUT;
          I1=^DATAOUT;
          M2=3;
          L3=L1;
          AX0=0XFFFF;
          DM(0X3FF7)=AX0;             {enable all multi-channel words}
          DM(0X3FF8)=AX0;
          DM(0X3FF9)=AX0;
          DM(0X3FFA)=AX0;
          AX0=DM(I1,M1);              {send first 16bits of data}
          TX0=AX0;
          AX0=0X1418;
          DM(0X3FFF)=AX0;             {turn on sport0}
          IFC=B#000000111111;         {clear all pending interrupts}
          IMASK=B#00001000;           {sport0 rx interrupt on}

WAIT_DATA:
          JUMP WAIT_DATA;             {wait for sport0 rx autobuffer interrupt}

NEWDATA:
          I3=I1;
          MODIFY(I3,M2);
          AX0=DM(DATAIN);             {this routine sends the incoming}
          DM(I3,M1)=AX0;              {a/d data straight to the d/a by}
          AX0=DM(DATAIN+1);           {copying newly arrived data words from}
          DM(I3,M1)=AX0;              {DATAIN into the DATAOUT buffer}
          RTI;

{The DATAOUT buffer is twice as long as the DATAIN buffer, since it contains}
{control words, as well as output data. This program ignores the control}
{words arriving back from the AD1849 during the data mode}

.ENDMOD;
```

Listing 12.1 ADSP-2111/AD1849 Talk-Through Routine (18492111.DSP)

12.2.2 ADSP-2105/AD1849 SoundPort Interface

This section contains a program that provides the initialization of control functions needed to interface the AD1849 SoundPort Stereo Codec to the ADSP-2105. Listing 12.2 is an ADSP-2105/AD1849 talk-through routine: incoming data is echoed immediately to the output.

You should consider the following points when using this program:

- Multichannel mode is not available on SPORT1, so SPORT1 is run in unframed mode. Any disruption in the serial data stream can cause the AD1849 to lose synchronization. The interface code checks incoming control words, and if there is a discrepancy, the code resets SPORT1. This could cause a loss of data if the data stream is corrupted. In a stable, electrically clean environment you should not have any significant problems. The SPORT receive interrupt routine checks for errors, and this routine can be removed if your environment does not require it.

- Control for the AD1849's D/C line is derived from a latched external memory-mapped port (the *mode_sel* port definition). The program uses data bit D8 (the LSB) as the mode select flag.

- RFS1 and TFS1 must be tied together for codec initialization and operation.

```
{*************************************************************************

ADSP-2105 - AD1849(SOUNDPORT)  INTERFACE PROGRAM
Analog Devices Jan, 1992

Revised 12/10/92: Equal length Tx and Rx buffers. Direct addressing
                  for Tx autobuffer writes (NEWDATA routine)

This program is written to run on an ADSP-2105.

For the hardware connections between the ADSP-2105 and AD1849, please refer to
the diagrams shown on the AD1849 data sheet.

Note:RFS1 and TFS1 MUST be tied together in order to initialize and operate
     the codec.

The AD1849 will be set up as follows:
(NC indicates a no care state, all control reg are 1 bit unless indicated
by [#bits])

Control time slot control bits:
DCB=0       followed by 1
AC=1        autocalibrate
DFR=5       Data conversion frequency,44.1kH [3]
ST=1        stereo mode
DF=0        dataformat is 16 bit twos complement [2]
MCK=2       16.9344 MHz is the master clock [2]
FSEL=0      frame size, 64 bits [2]
MS=1        master mode (i.e. receive external serial clock)
TXDIS=0     enable serial output
ENL=0       disable loopback testing
ADL=NC      loopback mode analog/digital (disabled)
PIO=NC      parallel I/O bits not used [2]
REVID=NC

Data Timeslot Control Bits:
OM0=1       enable line 0 output
OM1=1       enable line 1 output
LO=0        no attenuation of left channel output [6]
SM=0        mono output is muted
RO=0        no attenuation of right channel output [6]
PIO=NC      parallel I/O bits not used [2]
OVR=NC      overrange INPUT
IS=0        line-level stereo input selected
LG=0        no gain for left channel [4]
MA=15       no monitor mix (i.e. ADC output is not mixed with DAC input)
RG=0        no gain for right channel [4]

*************************************************************************
```

(listing continues on next page)

12 Hardware Interfacing

In its current condition, this program can be used “as is” to perform straight talkthrough (at 44.1 KHz sample rate) on both left and right channels of the AD1849. The incoming audio data from the 16bit ADCs is placed in a short buffer and immediately sent out to the 16bit DACs.

The initial setup and handshaking with the AD1849 starts with the “START” routine. A state machine to perform the handshaking is executed in software. Once the AD1849 is configured properly, the processor enters the WAIT_DATA loop and waits for serial port interrupt requests.
This program is booted into the ADSP-2105 on power-up.

```
*************************************************************************}
.
MODULE/ABS=0/BOOT=0  AD1849;
.VAR/CIRC   CTRLIN[4];              {circular buffers for data input and}
.VAR/CIRC   CTRLOUT[4];             {output for data mode and control mode}
.VAR/CIRC   DATAIN[4];
.VAR/CIRC   DATAOUT[4];
.VAR        FIRST_FLG;
.VAR        DCB_FLG;
.VAR        DMODE_FLG;
.var        sync_flag;

.port mode_sel;                     {latched control for Control/Data line}

         JUMP START;nop;nop;nop;          {restart interrupt}
         RTI;nop;nop;nop;                 {IRQ2 int, not used}
         RTI;nop;nop;nop;                 {SPORT0 tx not used}
         RTI;nop;nop;nop;                 {SPORT0 rx not used}
         JUMP SETUPCONTROL;nop;nop;nop;   {SPORT1 transmit int}
         JUMP NEWDATA;nop;nop;nop;        {SPORT1 receive int}
         RTI;nop;nop;nop;                 {timer int,not used}

START:   AX0=0;
         dm(mode_sel)=AX0;                {set AD1849 D/C pin low}
         L0=%CTRLIN;
         M0=1;
         I0=^CTRLIN;
         L1=%CTRLOUT;
         M1=1;
         I1=^CTRLOUT;
         AX0=1;
         DM(FIRST_FLG)=AX0;
         DM(DCB_FLG)=AX0;
         AX0=0;
         DM(DMODE_FLG)=AX0;               {in control mode}
         AY0=B#0010000100101100;
```

```
{Initialize contol mode output buffer}

         DM(I1,M1)= B#0010000100101100;   {DCB=0,AC=1,DFR=05,ST=1,DF=00}
         DM(I1,M1)= B#0010001000000000;   {MCK=02,FSEL=0,MS=1,TXDIS=0}
                                          {ENL=0,ADL=0}
         DM(I1,M1)= B#0000000000000000;   {PIO = 00}
         DM(I1,M1)= B#0000000000000000;   {REVID=NC}
         L2 = 0;                          {linear addressing for register}
         I2 = 0x3fef;                     {point to last DM cntrl reg}
{3FEF}   DM(I2,M1) = 0x0283;              {sport1 autobuffer register}
{3FF0}   DM(I2,M1) = 383;                 {rfsdiv1, not really used}
{3FF1}   DM(I2,M1) = 849;                 {sclkdiv1}
{3FF2}   DM(I2,M1) = B#0100000100011111;  {sport1 control register:
                                          internal sclk & rfs
                                          normal framing mode
                                          frame sync not inverted
                                          16-bit word length      }
         I2=0x3ffb;
{3FFB}   DM(I2,M1) = 0x0000;              {TSCALE register}
{3FFC}   DM(I2,M1) = 0x0000;              {TCOUNT register}
{3FFD}   DM(I2,M1) = 0x0000;              {TPERIOD register,initializing value
                                          for TCOUNT after every interrupt }
{3FFE}   DM(I2,M1) = 0x0000;              {external data memory waits=0}

         ICNTL=0x00 ;
         IMASK=B#000100;                  {only SPORT1 tx interurpt enabled
                                          initially while in control mode }

{.... Set bit test mask for DCB bit, used in tx interrupt state machine ....}
         AY0=B#0010000100101100;          {test mask for DCB bit}

{....       send first control word to switch codec to data mode        ....}
         AX0=DM(I1,M1);                   {send first 16bits of ctrl word}
         TX1=AX0;
{3FFF}   DM(I2,M1) = 0x0c18;              {system control reg: sport1 enabled}

{.... Wait for an interrupt indicating that transmit register is ready for
   new data and that the 2105 has received a 16bit word                ....}

WAIT1:   AX1=DM(DMODE_FLG);               {check dmode flag}
         AR=PASS AX1;
         IF GT JUMP GO_DMODE;             {if set, in data mode}
         JUMP WAIT1;                      {else, wait for initialization to
                                          be completed from tx interrupt
                                          routine           }

GO_DMODE:   L0=%DATAIN;                   {init I0, L0 for rx autobuffer}
            I0=^DATAIN;
            L1=%DATAOUT;                  {init I1, L1 for tx autobuffer}
            I1=^DATAOUT;
```

(listing continues on next page)

```
         M2=3;
         AX0=DM(I1,M1);             {send first 16bits of data}
         TX1=AX0;
         AX0=0X0c18;
         DM(0X3FFF)=AX0;            {turn on sport1}
         IFC=B#000000111111;        {clear all pending interrupts}
         nop;                       {cycle for IFC latency}
         IMASK=B#000010;            {sport1 rx interrupt on}

WAIT_DATA:
         JUMP WAIT_DATA;            {wait for sport1 rx autobuffer interrupt}

{sport1 tx interrupt routine}

{This routine initializes the AD1849 control mode and then waits to     }
{see if the codec is ready to be switched to data mode. The routine     }
{also initializes the transmit autobuffer with the appropriate          }
{data-mode control words. See the AD1849 data sheet for a complete      }
{explanation of control word bits.                                      }
{                                                                    }
{Note: AY0 contains a bit mask and must NOT be modified elsewhere       }
{                                                                    }
{                                                                    }

SETUPCONTROL:
         AX0=DM(FIRST_FLG);         {first time through?}
         AF=PASS AX0;
         IF NE JUMP DECR_FIRST;     {if so, wait until next word transmitted}
         AX0=DM(DCB_FLG);
         AR=PASS AX0;
         IF EQ JUMP DCBFLG_SET;

         AX0=DM(CTRLIN);            {DCB_FLG has not been set yet}
         AR=AX0 XOR AY0;            {check all incoming bits including DCB bit}
         IF EQ JUMP SET_DCB;        {set flag if DCB was 0}
         RTI;

DCBFLG_SET:
         AX0=DM(CTRLIN);            {DCB_FLG was set}
         AR=AX0 AND AY0;            {only check for DCB bit}
         IF NE JUMP SETDMODE;       {if DBC=1 ready for datamode}
         RTI

SET_DCB:
         AX0=0;
         DM(DCB_FLG)=AX0;
         AX0=B#0010010100101100;    {DCB was 0, prepare to send DCB=1}
         DM(CTRLOUT)=AX0;
         AY0=B#0000010000000000;
         RTI;
```

```
DECR_FIRST: AX0=0;
            DM(FIRST_FLG)=AX0;            {if first time, set flag=0}
            RTI;

SETDMODE:   IMASK=0;
            AX0=0X0418;                   {disable sport1}
            DM(0X3FFF)=AX0;

{  At this point we could boot page#1 and do the following data mode setup
   after the reboot occurs. This would free up some PM RAM for other uses.

   The data mode setup initializes the transmit autobuffer control word
   elements. These values are not changed in this talkthrough application.    }

          I1 = ^DATAOUT;
          L1=0;
          DM(I1,M1) = 0x0000;             {reset output & input data buffers}
          DM(I1,M1) = 0x0000;             {initialize embedded control bits}
          DM(I1,M1) = B#1100000000000000; {OM1=1,OM1=1,LO=0,SM=0,RO=0}
          DM(I1,M1) = B#0000000011110000; {PIO=00,OVR=0,IS=0,LG=0,MA=15,RG=0}
          DM(I1,M1) = 0x0000;             {reset output & input data buffers}
          DM(I1,M1) = 0x0000;             {initialize embedded control bits}
          DM(I1,M1) = B#1100000000000000; {OM1=1,OM1=1,LO=0,SM=0,RO=0}
          DM(I1,M1) = B#0000000011110000; {PIO=00,OVR=0,IS=0,LG=0,MA=15,RG=0}
          AX0=0X001F;
          DM(0X3FF2)=AX0;                 {  sport1 control:}
                                          internal tfs
                                          external sclk & rfs
                                          16bit words  }
          AX0=1;
          dm(mode_sel)=AX0;               {set D/C high}
          DM(DMODE_FLG)=AX0;              {set data mode flag high}
          RTI;

{                                                                          }
{                                                                          }
{ sport1 rx interrupt routine                                                  }
{                                                                          }
{ This routine sends the incoming a/d data straight to the d/a by copying     }
{ newly arrived data words from the DATAIN buffer into the DATAOUT buffer     }
{                                                                          }
{                                                                          }

NEWDATA: AX0=DM(DATAIN);                  {get LEFT channel data}
         DM(dataout)=AX0;                 {output LEFT channel data}
         AX0=DM(DATAIN+1);                {get RIGHT channel data}
         DM(dataout+1)=AX0;               {output RIGHT channel data}
```

(listing continues on next page)

```
{                                                                          }
{ Error checking section (optional). Incoming control words are compared   }
{ to the outgoing control words.                                           }
{                                                                          }
         ax0=dm(dataout+2);          {read known output control word}
         ay0=dm(datain+2);           {read newly received control word}
         ar=ax0-ay0;                 {compare, they should be the same}
         if eq jump no_error;        {if same, no error}
         ax0=dm(sync_flag);          {if not, read SYNC_FLAG}
         ar=pass ax0;                {test for 0 or 1}
         if ne jump reset_sport;     {if 1, second failure, reset SPORT1}
         ar=pass 1;                  {else, first failure, set SYNC_FLAG}
         dm(sync_flag)=ar;
         rti;                        {return}

reset_sport:
         si=0x418;                   {disable SPORT1}
         dm(0x3fff)=si;
         si=0xc18;                   {re-enable SPORT1}
         dm(0x3fff)=si;
no_error:
         ar=pass 0;
         dm(sync_flag)=ar;           {reset SYNC_FLAG}
         rti;                        {return}
{                                                                          }
{ The DATAOUT buffer is twice as long as the DATAIN buffer, since it       }
{ contains control words, as well as output data. This program ignores the }
{ control words arriving back from the AD1849 during the data mode, except }
{ for the purpose of error checking.                                       }
{                                                                          }

next:    ax0=dm(0x3fff);
         ay0=b#0000001001000000;
         ar=ax0 OR ay0;
         dm(0x3fff)=ar;
         rti;
.ENDMOD;
```

Listing 12.2 ADSP-2105/AD1849 Talk-Through Routine (18492105.DSP)

12.2.3 ADSP-2101/AD1847 SoundPort Interface

This section contains a program that provides the initialization of control functions needed to interface the AD1847 SoundPort Stereo Codec to the ADSP-2101.

The AD1847 has a TDM serial interface. The code provided should work without modifications for the ADSP-2101, ADSP-2103, and ADSP-2115. The code requires minor modifications to the interrupt vector table to use it with the ADSP-2111, ADSP-2171, and ADSP-21msp5x processors. Also, you can probably use the additional flags available on these processors to optimize the code.

Analog Devices does not recommend using this code on the ADSP-2105 because the ADSP-2105 does not have a TDM serial port. For additional information about AD1847 interfacing, refer to the “README” file included with the other files for this chapter.

Listing 12.3 is an ADSP-2101 / AD1847 talk-through routine.

You should consider the following points when using this program:

- Although the AD1847 and the ADSP-2101 are set for 32-word blocks, the AD1847 works on 16-word cycles. Therefore, control words are sent on time slots 0 and 16, left channel data is sent on slots 1 and 17, and right channel data is sent on slots 2 and 18. Receive data follows a similar pattern.

- The AD1847 has a two-sample buffer to allow for slower sampling rates. Since the AD1847’s serial bit clock rate is fixed, the interval between time slot 0 and time slot 16 is less than the sample period. For example, for an 8 kHz sampling rate, data is expected every 125 µsec. At that sampling rate, the serial bit clock produced by the AD1847 is 12.288 MHz, yielding a time span between data samples of 20.83 µsec (81.38 ns x 16 bits / word x 16 words). The actual data received, however, is sampled at the correct time interval and stored in the AD1847’s output buffer. Remember that time is needed to convert the data, and that the AD1847 can wait between generating frame syncs to insure proper sample timing.

12 Hardware Interfacing

```
.module/ram/abs=0       ad1847;

/************************************************************************
ADSP-2101 -> AD1847: Talkthru Interface

This code provides the necessary initializations and setup to allow for
communication between the AD1847 SoundPort codec and the ADSP-2100 family
Serial Port 0.

This interface code was written around an AD1847 connected to an ADSP-2101
EZ-Lab board through the J2 SPORT Connector.

When the board is RESET, the codec is initialized for an
8kHz sampling rate of stereo PCM data.

This module can be used without modification for the ADSP-2101, ADSP-2103,
and ADSP-2115. Use with the ADSP-2111, ADSP-2171, or ADSP-21msp5x processors
would require modification of the interrupt vector table and any instructions
relating to IFC, IMASK, or any other interrupt-related structure.

This interface is not recommended for the ADSP-2105.
************************************************************************/

.var/dm/ram/circ        rx_buf[3];              /* Status + L data + R data */
.var/dm/ram/circ        tx_buf[3];              /* Cmd + L data + R data    */
.init tx_buf:           0xc000, 0x0000, 0x0000; /* Initially set MCE        */

.var/dm/ram/circ        init_cmds[13];
/*********************************************************/
/*       Initial codec setup:                            */
/* 0:  Left Input Control: 0 gain, Line 1 input          */
/* 1:  Right Input Control: 0 gain, Line 1 input         */
/* 2:  Left Aux #1 Input Control: Muted                  */
/* 3:  Right Aux #1 Input Control: Muted                 */
/* 4:  Left Aux #2 Input Control: Muted                  */
/* 5:  Right Aux #2 Input Control: Muted                 */
/* 6:  Left DAC Control: 0 attenuation, muted            */
/* 7:  Right DAC Control: 0 attenuation, muted           */
/* 8:  Data Format: XTAL1, 8kHz sampling, stereo,        */
/*         16-bit linear PCM                             */
/* 9:  Interface Config: Playback enabled, ACAL allowed  */
/* 10: Pin Control: CLKOUT active, XCTL1/0 LO            */
/* 12: Misc. Info: Transmit on 0,1,2, 32-word frame      */
/* 13: Digital Mix Control: DME Disabled, 0 attenuation  */
/*                                                       */
/* To start-up the codec with any other properties,      */
/* change the appropriate initialization value.          */
/* Refer to the AD1847 data sheet for detailed register  */
/* descriptions.                                         */
/*********************************************************/
```

```
.init init_cmds:
            0xc000,
            0xc100,
            0xc280,
            0xc380,
            0xc480,
            0xc580,
            0xc680,
            0xc780,
            0xc850,
            0xc909,
            0xca00,
            0xcc40,
            0xcd00;

.var/dm     stat_flag;

reset_vect: jump sys_init; nop; nop; nop;

irq2_svc:   rti; nop; nop; nop;

tx0_irq:    ar = dm(stat_flag);
            ar = pass ar;
            if eq rti;
            jump next_cmd;

rx0_irq:    jump input_samples;
            rti; nop; nop;

            rti; nop; nop; nop;
            rti; nop; nop; nop;
            rti; nop; nop; nop;

/***************
 Code Start:
****************/
sys_init:
            i0 = ^rx_buf;
            l0 = %rx_buf;
            i1 = ^tx_buf;
            l1 = %tx_buf;
            i3 = ^init_cmds;
            l3 = %init_cmds;
            m1 = 1;
```

(listing continues on next page)

```
/**********************
   sport0 setup:
         multichannel enable, ext. sclk, MFD=1,
         32 words, ext. rfs, DTYPE 00, 16 bits
**********************/
         ax0 = b#1000011000001111;
         dm(0x3ff6) = ax0;

/**********************
   Multichannel enable setup:
         Rx: 3, 4, 5, 19, 20, 21
         Tx: 0, 1, 2, 16, 17, 18
**********************/
         ax0 = b#0000000000000111;
         dm(0x3ff9) = ax0;
         dm(0x3ffa) = ax0;

         ax0 = b#0000000000000111;
         dm(0x3ff7) = ax0;
         dm(0x3ff8) = ax0;

/**********************
   SPORT Autobuffer Setup
         Rx: I0, M1
         Tx: I1, M1
**********************/
         ax0 = b#0000001010000111;
         dm(0x3ff3) = ax0;

         ax0 = b#0001000000000000;
         dm(0x3fff) = ax0;          /* SPORT0 Enabled,
                                    SPORT1 = FI, FO, IRQs */
start_setup:
         ifc = b#000000111111;
         nop;
         ax0 = 1;
         dm(stat_flag) = ax0;
         imask = b#010000;          /* enable tx0 interrupt */
         ax0 = dm(i1,m1);
         tx0 = ax0;
check_init:
         ax0 = dm(stat_flag);       /* wait for entire init */
         af = pass ax0;             /* buffer to be sent to */
         if ne jump check_init;     /* the codec            */
check_aci:
         ax0 = dm(rx_buf);          /* once initialized, wait */
         ay0 = b#0000000000000010;  /* for codec to come out  */
         ar = ax0 and ay0;          /* of autocalibration     */
         if ne jump check_aci;
```

```
        idle;

        ay0 = 0xbf3f;
        ax0 = dm(init_cmds+6);
        ar = ax0 AND ay0;
        dm(tx_buf) = ar;                /* unmute left DAC */
        idle;

        ax0 = dm(init_cmds+7);
        ar = ax0 AND ay0;
        dm(tx_buf) = ar;                /* unmute right DAC */
        idle;

        ifc = b#000000111111;
        nop;
        imask = b#001000;               /* enable rx0 interrupt */

/************************************************************************/
/* Main Loop: talkthru                                                  */
/************************************************************************/
talkthru:idle;
        jump talkthru;

/************************************************************************/
input_samples:
        ena sec_reg;
        ax1 = dm(rx_buf+1);             /* L channel input */
        mx1 = dm(rx_buf+2);             /* R channel input */
        dm(tx_buf+2) = mx1;
        dm(tx_buf+1) = ax1;
        rti;

next_cmd:
        ena sec_reg;
        ax0 = dm(i3,m1);
        dm(tx_buf) = ax0;
        ax0 = i3;
        ay0 = ^init_cmds;
        ar = ax0 - ay0;
        if gt rti;
        ax0 = 0x8000;
        dm(tx_buf) = ax0;
        ax0 = 0;                    /* remove MCE if done initialization */
        dm(stat_flag) = ax0;        /* reset status flag */
        rti;

.endmod;
```

Listing 12.3 ADSP-2101/AD1847 Talk-Through Routine (TALK_47.DSP)

12 Hardware Interfacing

Listing 12.4 is an ADSP-2101 / AD1847 demonstration routine. This program was written to demonstrate several features of the AD1847 when it is connected to the ADSP-2101 EZ-LAB® Demonstration Board.

```
.module/ram/abs=0       ad1847;

/************************************************************************
 ADSP-2101 -> AD1847: Talkthru Interface

This code provides the necessary initializations and setup to allow for
communication between the AD1847 SoundPort codec and the ADSP-2100 family
Serial Port 0.

This interface code was written around an AD1847 connected to an ADSP-2101
EZ-Lab board through the J2 SPORT Connector.

There are two modes of operation of this interface:

1) If the board is RESET, the codec is initialized for an
   8kHz sampling rate of stereo PCM data.

2) If the FLAG_IN button is held at reset, then the code enters its AD1847
   configuration mode.

Codec attributes that can be changed are: line or microphone input source,
input gain, and the sample rate. Do not press the IRQ2 button too fast when
changing the sample rate as the codec needs time to perform its autoclibration
procedure for each sampling rate.

Initialization:
         SPORT1 must be configured as flags and interrupts.
         irq2 must be enabled.

Operation:
         Press and hold       <FLAG> button
         Press and release    <IRQ2> button
         Release              <FLAG> button to enter the setup routine

         Push <IRQ2> button to toggle between Line_1 and Line_2 input
         Push <FLAG> button to go to the next state

         Push <IRQ2> button to change the input gain, (8 levels)
         Push <FLAG> button to go to the next state

         Push <IRQ2> button to change the sample rate, (16 steps)
         Push <FLAG0> button to exit the setup routine
```

```
 Line Input Gain Default:     level 0 = 0dB
 Sample Rates:
        (1)      8 (default)       (9)   5.5125
        (2)      16                (10)  11.025
        (3)      27.42857          (11)  18.9
        (4)      32                (12)  22.05
        (5)      N/A               (13)  37.8
        (6)      N/A               (14)  44.1
        (7)      48                (15)  33.075
        (8)      9.6               (16)  6.615

This module can be used without modification for the ADSP-2101, ADSP-2103,
and ADSP-2115. Use with the ADSP-2111, ADSP-2171, or ADSP-21msp5x processors
would require modification of the interrupt vector table and any instructions
relating to IFC, IMASK, or any other interrupt-related structure.

This interface is not recommended for the ADSP-2105.
************************************************************************/

.var/dm/ram/circ        rx_buf[3];              /* Status + L data + R data */
.var/dm/ram/circ        tx_buf[3];              /* Cmd + L data + R data    */
.init tx_buf:           0xc000, 0x0000, 0x0000; /* Initially set MCE        */

.var/dm/ram/circ        init_cmds[13]; /
*************************************************************/
/*       Initial codec setup:                                */
/* 0:  Left Input Control: 0 gain, Line 1 input              */
/* 1:  Right Input Control: 0 gain, Line 1 input             */
/* 2:  Left Aux #1 Input Control: Muted                      */
/* 3:  Right Aux #1 Input Control: Muted                     */
/* 4:  Left Aux #2 Input Control: Muted                      */
/* 5:  Right Aux #2 Input Control: Muted                     */
/* 6:  Left DAC Control: 0 attenuation, muted                */
/* 7:  Right DAC Control: 0 attenuation, muted               */
/* 8:  Data Format: XTAL1, 8kHz sampling, stereo,            */
/*            16-bit linear PCM                              */
/* 9:  Interface Config: Playback enabled, ACAL allowed      */
/* 10: Pin Control: CLKOUT active, XCTL1/0 LO                */
/* 12: Misc. Info: Transmit on 0,1,2, 32-word frame          */
/* 13: Digital Mix Control: DME Disabled, 0 attenuation      */
/*************************************************************/
```

(listing continues on next page)

```
.init init_cmds:
            0xc000,
            0xc100,
            0xc280,
            0xc380,
            0xc480,
            0xc580,
            0xc680,
            0xc780,
            0xc850,
            0xc909,
            0xca00,
            0xcc40,
            0xcd00;

.var/dm     stat_flag;
.var/dm     gain_state;
.var/dm     sampling_state;
.var/dm     chng_state;

reset_vect: jump setup_mode; nop; nop; nop;

irq2_svc:   toggle FLAG_OUT; si = 1; dm(chng_state) = si; rti;

tx0_irq:    ar = dm(stat_flag);
            ar = pass ar;
            if eq rti;
            jump next_cmd;

rx0_irq:    jump input_samples;
            rti; nop; nop;

            rti; nop; nop; nop;
            rti; nop; nop; nop;
            rti; nop; nop; nop;

/********************
   Code Start:
*********************/
setup_mode:
            set flag_out;
            icntl = b#00100;              /* edge sensitive IRQ2 */
            ax0 = 0x0018;
            dm(0x3fff) = ax0;
            nop;
            nop;
            if FLAG_IN jump sys_init;

            ifc = b#000000111111;
            ax0 = 0;
            dm(chng_state) = ax0;

            imask = b#100000;
```

```
wait_irq:   ar = dm(chng_state);                /* wait for IRQ2 */
            ar = pass ar;
            if eq jump wait_irq;
            imask = 0x0000;

wait_rel_FO:if NOT FLAG_IN jump wait_rel_FO;    /* wait for FLAG_IN release */

            ax0 = 0;
            dm(chng_state) = ax0;

            jump set_codec;

sys_init:
            i0 = ^rx_buf;
            l0 = %rx_buf;
            i1 = ^tx_buf;
            l1 = %tx_buf;
            i3 = ^init_cmds;
            l3 = %init_cmds;

            m1 = 1;

/**********************
   sport0 setup: multichannel enable, ext. sclk, MFD=1,
            32 words, ext. rfs, DTYPE 00, 16 bits
**********************/
            ax0 = b#1000011000001111;
            dm(0x3ff6) = ax0;

/*********************
   Multichannel enable setup:
            Rx: 3, 4, 5, 19, 20, 21
            Tx: 0, 1, 2, 16, 17, 18
*********************/
            ax0 = b#0000000000000111;
            dm(0x3ff9) = ax0;
            dm(0x3ffa) = ax0;

            ax0 = b#0000000000000111;
            dm(0x3ff7) = ax0;
            dm(0x3ff8) = ax0;
```

(listing continues on next page)

```
/*********************
   SPORT Autobuffer Setup
         Rx: I0, M1
         Tx: I1, M1
**********************/
         ax0 = b#0000001010000111;
         dm(0x3ff3) = ax0;

         ax0 = b#0001000000000000;
         dm(0x3fff) = ax0;               /* SPORT0 Enabled,
                                            SPORT1 = FI, FO, IRQs */

start_setup:
         ifc = b#000000111111;
         nop;

         ax0 = 1;
         dm(stat_flag) = ax0;

         imask = b#010000;               /* enable tx0 interrupt */

         ax0 = dm(i1,m1);
         tx0 = ax0;

check_init:
         ax0 = dm(stat_flag);            /* wait for entire init */
         af = pass ax0;                  /* buffer to be sent to */
         if ne jump check_init;          /* the codec            */

check_aci:
         ax0 = dm(rx_buf);               /* once initialized, wait */
         ay0 = b#0000000000000010;       /* for codec to come out  */
         ar = ax0 and ay0;               /* of autocalibration     */
         if ne jump check_aci;

         idle;

         ay0 = 0xbf3f;
         ax0 = dm(init_cmds+6);
         ar = ax0 AND ay0;
         dm(tx_buf) = ar;                /* unmute left DAC */
         idle;

         ax0 = dm(init_cmds+7);
         ar = ax0 AND ay0;
         dm(tx_buf) = ar;                /* unmute right DAC */
         idle;

         ifc = b#000000111111;
         nop;
         imask = b#001000;               /* enable rx0 interrupt */
```

```
/************************************************************************/
/* Main Loop: talkthru                                                  */
/************************************************************************/

talkthru:
         idle;
         jump talkthru;

/************************************************************************/

input_samples:
         ena sec_reg;
         ax1 = dm(rx_buf+1);        /* L channel input */
         mx1 = dm(rx_buf+2);        /* R channel input */
         dm(tx_buf+2) = mx1;
         dm(tx_buf+1) = ax1;
         rti;

next_cmd:
         ena sec_reg;
         ax0 = dm(i3,m1);
         dm(tx_buf) = ax0;
         ax0 = i3;
         ay0 = ^init_cmds;
         ar = ax0 - ay0;
         if gt rti;
         ax0 = 0x8000;
         dm(tx_buf) = ax0;
         ax0 = 0;                   /* remove MCE if done initialization */
         dm(stat_flag) = ax0;       /* reset status flag */
         rti;

/************************************************************************
 Codec configuration section
************************************************************************/
set_codec:
         set FLAG_OUT;
         ax0 = 0;
         dm(gain_state) = ax0;
         dm(sampling_state) = ax0;

         IFC = b#000000111111;
         nop;
         imask = b#100000;          /* enable IRQ2 interrupt    */
```

(listing continues on next page)

```
/****************
 Input Selection
*****************/

set_input:
         if NOT FLAG_IN jump gain_wait;
         sr0 = dm(chng_state);
         af = pass sr0;
         if eq jump set_input;

         ax0 = dm(init_cmds);
         ay0 = 0xc080;
         ar = ax0 XOR ay0;          /* if already set for line 1 in, */
         if ne jump set_to_2;       /* switch to line 2              */
         ax0 = 0xc000;
         dm(init_cmds) = ax0;
         ax0 = 0xc100;
         dm(init_cmds+1) = ax0;
         ax0 = 0;                   /* reset flag for IRQ2 */
         dm(chng_state) = ax0;
         jump set_input;

set_to_2:
         ax0 = 0xc080;
         dm(init_cmds) = ax0;
         ax0 = 0xc180;
         dm(init_cmds+1) = ax0;
         ax0 = 0;                   /* reset flag for IRQ2 */
         dm(chng_state) = ax0;
         jump set_input;

/****************
 Gain Selection
*****************/

gain_wait:
         if NOT FLAG_IN jump gain_wait;     /* software switch de-bounce */

set_gain:
         if NOT FLAG_IN jump sampling_wait;
         sr0 = dm(chng_state);
         af = pass sr0;
         if eq jump set_gain;

         ax1 = dm(gain_state);
         ar = pass ax1;
         if gt jump next1;
```

```
          ax0 = 0xfff0;              /* set gain = 0 for both inputs */
          ay0 = dm(init_cmds);
          ar = ax0 AND ay0;
          dm(init_cmds) = ar;
          ay0 = dm(init_cmds+1);
          ar = ax0 AND ay0;
          dm(init_cmds+1) = ar;
          ax0 = 1;
          dm(gain_state) = ax0;
          jump set_gain;

next1:    ax1 = dm(gain_state);
          ay0 = 1;
          ar = ax1 - ay0;
          if gt jump next2;

          ax0 = 0x0002;              /* set gain = 2 = 3dB */
          call gain_adjust;
          jump set_gain;

next2:    ax1 = dm(gain_state);
          ay0 = 2;
          ar = ax1 - ay0;
          if gt jump next3;

          ax0 = 0x0006;              /* set gain = 4 = 6dB */
          call gain_adjust;
          jump set_gain;

next3:    ax1 = dm(gain_state);
          ay0 = 3;
          ar = ax1 - ay0;
          if gt jump next4;

          ax0 = 0x0002;              /* set gain = 6 = 9dB */
          call gain_adjust;
          jump set_gain;

next4:    ax1 = dm(gain_state);
          ay0 = 4;
          ar = ax1 - ay0;
          if gt jump next5;

          ax0 = 0x000e;              /* set gain = 8 = 12dB */
          call gain_adjust;
          jump set_gain;
```

(listing continues on next page)

```
next5:   ax1 = dm(gain_state);
         ay0 = 5;
         ar = ax1 - ay0;
         if gt jump next6;

         ax0 = 0x0002;              /* set gain = 10 = 15dB */
         call gain_adjust;
         jump set_gain;

next6:   ax1 = dm(gain_state);
         ay0 = 6;
         ar = ax1 - ay0;
         if gt jump next7;

         ax0 = 0x0006;              /* set gain = 12 = 18dB */
         call gain_adjust;
         jump set_gain;

next7:   ax0 = 0x0002;              /* set gain = 13 = 21dB */
         call gain_adjust;
         ax0 = 0;
         dm(gain_state) = ax0;
         jump set_gain;

/*______
 gain adjust
______*/
gain_adjust:
         ay0 = dm(init_cmds);
         ar = ax0 XOR ay0;
         dm(init_cmds) = ar;
         ay0 = dm(init_cmds+1);
         ar = ax0 XOR ay0;
         dm(init_cmds+1) = ar;
         ay0 = dm(gain_state);
         ar = ay0+1;
         dm(gain_state) = ar;
         ax0 = 0;
         dm(chng_state) = ax0;
         rts;
```

```
/************************
 Sampling Rate Selection
************************/
sampling_wait:
         if NOT FLAG_IN jump sampling_wait;

set_sampling:
         if NOT FLAG_IN jump done_config;
         sr0 = dm(chng_state);
         af = pass sr0;
         if eq jump set_sampling;

         ax0 = 0;
         dm(chng_state) = ax0;

set_rate:
         ar = dm(sampling_state);
         ar = pass ar;
         if gt jump to_16;

to_8:    ar = 1;                    /* buffer initialized for 8 kHz */
         dm(sampling_state) = ar;
         jump set_sampling;

to_16:   ax0 = dm(sampling_state);
         ay0 = 1;
         ar = ax0 - ay0;
         if gt jump to_27;

         ax1 = 0xc852;              /* set for 16.0 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 2;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_27:   ax0 = dm(sampling_state);
         ay0 = 2;
         ar = ax0 - ay0;
         if gt jump to_32;

         ax1 = 0xc854;              /* set for 27.42857 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 3;
         dm(sampling_state) = ax0;
         jump set_sampling;
```

(listing continues on next page)

```
to_32:   ax0 = dm(sampling_state);
         ay0 = 3;
         ar = ax0 - ay0;
         if gt jump to_na;

         ax1 = 0xc856;              /* set for 32.0 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 4;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_na:   ax0 = dm(sampling_state);
         ay0 = 4;
         ar = ax0 - ay0;
         if gt jump to_na2;

         ax0 = 5;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_na2:  ax0 = dm(sampling_state);
         ay0 = 5;
         ar = ax0 - ay0;
         if gt jump to_48;

         ax0 = 6;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_48:   ax0 = dm(sampling_state);
         ay0 = 6;
         ar = ax0 - ay0;
         if gt jump to_96;

         ax1 = 0xc85c;              /* set for 48.0 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 7;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_96:   ax0 = dm(sampling_state);
         ay0 = 7;
         ar = ax0 - ay0;
         if gt jump to_55;

         ax1 = 0xc85e;              /* set for 9.6 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 8;
         dm(sampling_state) = ax0;
         jump set_sampling;
```

```
to_55:   ax0 = dm(sampling_state);
         ay0 = 8;
         ar = ax0 - ay0;
         if gt jump to_11;

         ax1 = 0xc851;              /* set for 5.5125 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 9;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_11:   ax0 = dm(sampling_state);
         ay0 = 9;
         ar = ax0 - ay0;
         if gt jump to_18;

         ax1 = 0xc853;              /* set for 11.025 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 10;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_18:   ax0 = dm(sampling_state);
         ay0 = 10;
         ar = ax0 - ay0;
         if gt jump to_22;

         ax1 = 0xc855;              /* set for 18.9 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 11;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_22:   ax0 = dm(sampling_state);
         ay0 = 11;
         ar = ax0 - ay0;
         if gt jump to_37;

         ax1 = 0xc857;              /* set for 22.05 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 12;
         dm(sampling_state) = ax0;
         jump set_sampling;
```

(listing continues on next page)

```
to_37:   ax0 = dm(sampling_state);
         ay0 = 12;
         ar = ax0 - ay0;
         if gt jump to_44;

         ax1 = 0xc859;              /* set for 37.8 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 13;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_44:   ax0 = dm(sampling_state);
         ay0 = 13;
         ar = ax0 - ay0;
         if gt jump to_33;

         ax1 = 0xc85b;              /* set for 44.1 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 14;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_33:   ax0 = dm(sampling_state);
         ay0 = 14;
         ar = ax0 - ay0;
         if gt jump to_66;

         ax1 = 0xc85d;              /* set for 33.075 kHz */
         dm(init_cmds+8) = ax1;
         ax0 = 15;
         dm(sampling_state) = ax0;
         jump set_sampling;

to_66:   ax0 = 0;
         dm(sampling_state) = ax0;

         ax1 = 0xc85f;              /* set for 6.615 kHz */
         dm(init_cmds+8) = ax1;
         jump set_sampling;


done_config:
         toggle FLAG_OUT;
         jump sys_init;
.endmod;
```

Listing 12.4 ADSP-2101/AD1847 Demonstration Routine (DEMO_47.DSP)

12.3 INTERFACING DRAMS WITH THE ADSP-2100 FAMILY

As new algorithms for digital signal processors are developed, the memory requirements for these applications will continue to grow. Not only will these applications require more memory, but they will require increased efficiency for data storage and retrieval. Examples of these new applications include digital processing for two- and three-dimensional graphic images and speech storage for voice mail systems and digital telephone answering machines.

Two common storage options for these applications are Static Random Access Memories (SRAMs) and Dynamic Random Access Memories (DRAMs).

The functional difference between an SRAM and a DRAM is how the devices store data. In an SRAM, data is stored in transistors that hold their value until you overwrite them with new data. The DRAM stores data in capacitors that gradually loose their charge and, without refreshing, will loose the data.

Size is another important difference. Capacitors are significantly smaller than transistors, so DRAMs have a higher bit density.

Although a DRAM package is smaller than the equivalent memory size in an SRAM package, you should consider the following factors when you decide which memory device to use:

- Capacitor leakage.
  Over time, DRAM capacitors “leak” or lose current and they must be “refreshed” periodically.

- Multiplexed-addressing.
  To take advantage of the higher bit density of the DRAM, the capacitor cells are accessed using multiplexed-addressing. This reduces the required number of address pins because you can use each pin twice: once to address a memory row and a second time to address a memory column. Although this makes the package smaller, it complicates memory addressing.

- Availability, price, and performance.
  DRAMs are readily available in 1 Mbit, 4 Mbit, and larger capacities, while SRAMs are typically available in 64 Kbyte (512 Kbit) and 128 Kbyte (1 Mbit) ranges. DRAMs are usually less expensive than SRAMs, but DRAM access times are significantly slower than SRAM access times.

12 Hardware Interfacing

- Program and hardware simplicity.
  The SRAM has a simple microprocessor interface. With N address lines, you can address 2^N sequential locations in SRAM. You do not need additional software overhead (to refresh) when your program reads or writes to memory. Each read or write requires only a single DSP instruction. When using the ADSP-2100 family, each instruction takes one clock cycle. If necessary, wait states can be programmed for addressing slower memories.

  When you use a DSP to address DRAM, you need additional hardware or software to handle the multiplexed row and column addressing and the refresh or precharge requirements. Many systems with DRAM use a hardware-intensive solution, like a DRAM controller.

  As an alternative, you can move the control functions into the DSP software. This is a good implementation for systems requiring large memory spaces and cost efficient solutions. The chip count is reduced at the expense of decreased software throughput.

Although DSPs were designed to interface with SRAMs, in the right application, DRAMs can provide a larger, more cost-effective storage medium.

This chapter presents a software solution for addressing DRAM with the ADSP-2100 family. To implement this solution, the ADSP-2101 EZ-LAB® Evaluation Board was used as the development platform. A simple DRAM interface board was designed for this application. The DRAM interface board connects to the EZ-LAB through the expansion connector to let the EZ-LAB access the interface board's bank of memory. Figure 12.1 is a block diagram of the test system. The subroutines included at the end of this chapter were verified on this interface board and the ADSP-2101 EZ-LAB®.

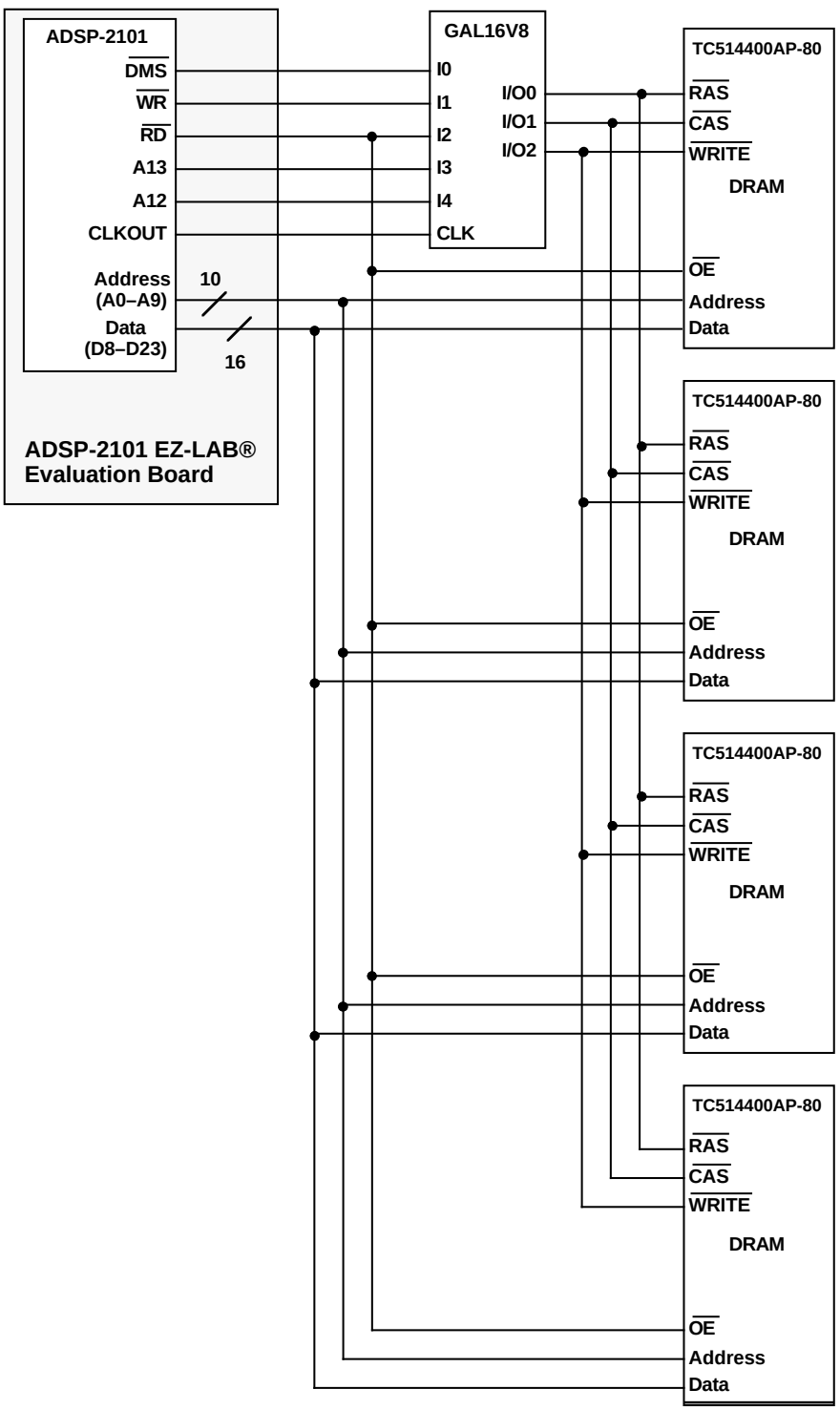

Figure 12.1 Functional Block Diagram Of DRAM Interface Test System

12 Hardware Interfacing

Depending on your application and DRAM selection, you may have to modify the subroutines and the PAL equations included in this chapter.

Table 12.1 lists the components used in the test system described above.

ADSP-2101 EZ-LAB Evaluation Board	1
1 M x 4 DRAM, PN TC514400AP-80 DRAM, 80 ns access time	4
Programmable Logic Device, PN GAL16V8-15	1
Ribbon Cable Connector, 60-Pin, Straight wire-wrap	1
Wire-wrap board	1

Table 12.1 Test System Components

When you design a DRAM memory system for an ADSP-2100 family DSP, you must consider the following points:

- DRAM configuration
- Multiplexed memory addressing
- DSP and DRAM control signals
- DSP to DRAM Interface timing
- DRAM Memory access modes
- DRAM Refresh

12.3.1 DRAM Configuration

Originally, DRAMs were organized in one-bit widths, such as 256 K x 1 or 1 M x 1. Since ADSP-2100 family DSPs are 16-bit fixed point processors, it would have taken 16 DRAMs to store data memory. As DRAM chip organization evolves, larger memories and wider widths are becoming available. DRAMs are available up to 4 M and larger with 4-bit and 16-bit widths.

You must make a trade-off between system cost, power consumption, and available board space when you choose DRAMs. Larger and wider DRAMs are more expensive and consume more power than smaller DRAMs, but you need fewer chips for the same amount of memory.

For this application, 1 M x 4 DRAMs (available from several manufactures) were chosen. These four DRAMs provide 1 M x 16 data memory.

12.3.2 Multiplexed Memory Addressing

To address a 1 M x 4 DRAM, you need ten address lines. With ten multiplexed address lines, you can address each row (2^{10}, or 1024 rows) and each column (2^{10}, or 1024 columns) in the DRAM. This is equal to 1024 x 1024, or 1 M, of addressable, 4-bit locations.

The ADSP-2100 family processors have 14 address lines for data memory. In an SRAM this can provide up to 2^{14}, or 16,384 (16 K) addressable data memory locations. In this application, you only need ten address lines to address 4, 1 M x 4 DRAMs; this system uses A0–A9 of the ADSP-2101 (see Figure 12.1). Since the remaining address lines are not used to address memory, they could be left unconnected, but this system uses two available address lines (A12 and A13) as control lines for the DRAMs.

12.3.3 DSP & DRAM Control Signals

To interface a DRAM to the DSP, compare the control signals and timing diagrams for both parts. Although this section has a brief overview of DSP and DRAM timing, you must determine the most effective use of the ADSP-2100 family control signals to control the reads and writes to DRAM in your application.

The DSP uses several control lines to access data memory: RD, WR, DMS. This application uses DMS (Data Memory Select) to differentiate between a program and data memory access because the program memory (PM) and data memory (DM) address and data lines are multiplexed off-chip.

12.3.3.1 DSP Read/Write Timing

Figure 12.2 shows the read and write timing for the DSP. The read cycle (or write cycle) begins when the processor puts the address on the data memory address (DMA) bus and asserts DMS. The RD (or WR) signal is then asserted. Data is placed on the data bus within a specified time, then RD (or WR) is deasserted. Finally, DMS is deasserted, ending the memory access.

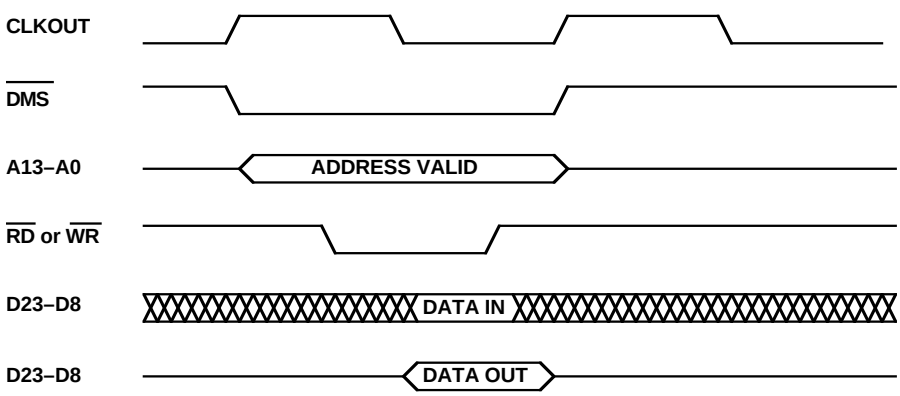

Figure 12.2 DSP Read/Write Timing

12.3.3.2 DRAM Read/Write Timing

The DRAM uses the control signals RAS (Row Address Strobe), CAS (Column Address Strobe), WRITE, and OE (Output Enable).

A DRAM read cycle (see Figure 12.3 for the read cycle timing) starts when the falling edge of RAS strobes the row address into the DRAM. The falling edge of CAS strobes the column address into the DRAM and, after an access delay, enables the output buffer. The WRITE signal must stay high before and after the falling edge of CAS. The read cycle ends when the RAS and CAS lines are brought high.

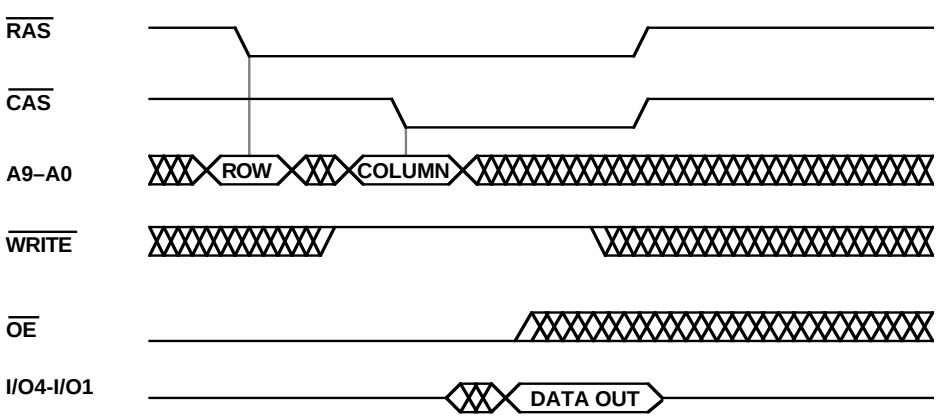

Figure 12.3 DRAM Read Cycle Timing

Another read cannot occur until the precharge time is met after RAS and CAS are brought high. There are minimum pulse-widths and setup and hold times associated with all control lines.

The DRAM write cycle is similar to the read cycle, except the WRITE line is held low. There are two basic write cycles for DRAMs: the early-write cycle and the delayed-write cycle. In the early-write cycle, the WRITE line is asserted before the assertion of CAS; in the delayed-write cycle, it is asserted after. For this DSP interface, use the delayed-write (or output enabled write) cycle.

A DRAM delayed-write cycle (write cycle timing shown in Figure 12.4) also starts when the falling edge of RAS strobes the row address into the DRAM. The falling edge of CAS strobes the column address into the DRAM. Then, the falling edge of WRITE latches the data into the DRAM. There are minimum pulse-widths and setup and hold times associated with these control lines as well.

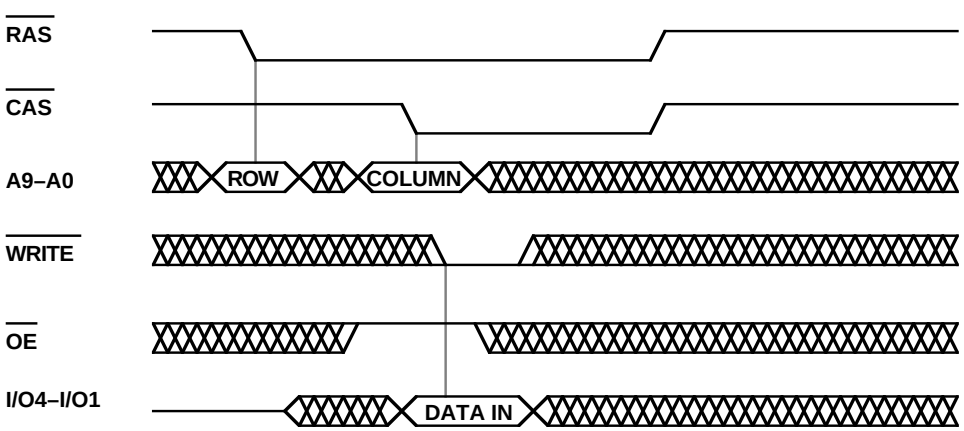

Figure 12.4 DRAM Delayed-Write (Output Enabled) Cycle Timing

12.3.3.3 RAS Generation

Because the falling edge of RAS latches the row address into the DRAM, the address must be valid before RAS is deasserted.

To drive RAS, use a combination of the DMS, RD, and address lines A13 and A12. A read from data memory locations 0x1000 to 0x13FF drives RAS low and latches the row address from 0x000 to 0x3FF. To drive RAS high again, read from any data memory address in the range 0x2000 to 0x27FF.

Use the following logic to drive RAS:

When (A13 = 0), (A12 = 1), (DMS = 0), and (RD = 0),
RAS = 0;

or, when (A13 = 1), (A12 = 0), (DMS = 0), and (RD = 0),
RAS =1;

otherwise, RAS = RAS;

You can implement the above logic discretely, or in a PAL, such as the GAL16V8 used in this application.

12.3.3.4 CAS Generation

The generation of CAS must occur after the generation of RAS. Since the negative transition of CAS latches the column address into the DRAM, the address must be present before CAS goes low. To generate CAS, use the same control pins (DMS, RD, WR, A13, and A12). A read or write to the data memory addresses 0x3000 to 0x33FF drives CAS low and latches the column address from 0x000 to 0x3FF. (Note a read from data memory 0x3800 or higher reads from the DSP's internal data memory and does not assert the external RD or address lines.) CAS returns to logic high at the completion of the read or write (when DMS returns high).

Use the following logic to drive CAS:

When (A13 = 1), (A12 = 1), (DMS = 0), ((RD = 0)#(WR =0))),
then CAS = 0;

otherwise CAS = 1;

The actual read or write from the DSP occurs when CAS is low. To meet the write timing requirements of the DRAM and the DSP, CAS must be held low for two clock cycles. To accomplish this, use the external data memory wait states of the processor.

Data memory has several configurable wait states. The address range for CAS (0x3000–0x33FF) is configured by setting DWAIT3 in the data memory wait state control register. For this application, set DWAIT3 to 1.

12.3.3.5 WRITE & OE Generation

The DRAM latches data on the falling edge of WRITE, while the DSP latches data on the rising edge. To ensure that the data from the DSP is still valid during the high-to-low transition of WRITE, the WR signal must be delayed one clock cycle. By delaying WR one cycle to create WRITE, you create a falling edge to latch valid data into the DRAM (see Figure 12.6). You can use a D flip-flop or PAL with registers to delay WR.

The RD output of the DSP can be run directly into the OE input of the DRAM.

12.3.4 DSP To DRAM Interface Timing

This section describes the specific interface timing required between the DSP and DRAM.

12.3.4.1 DRAM Read Timing

To read from the DRAM, the OE signal of the DRAM is tied to the DSP's RD line. Data is latched into the DSP on the rising edge of the RD signal and this matches the availability of data from the DRAM. Figure 12.5 shows the timing for generating the RAS and CAS signals to read from DRAM.

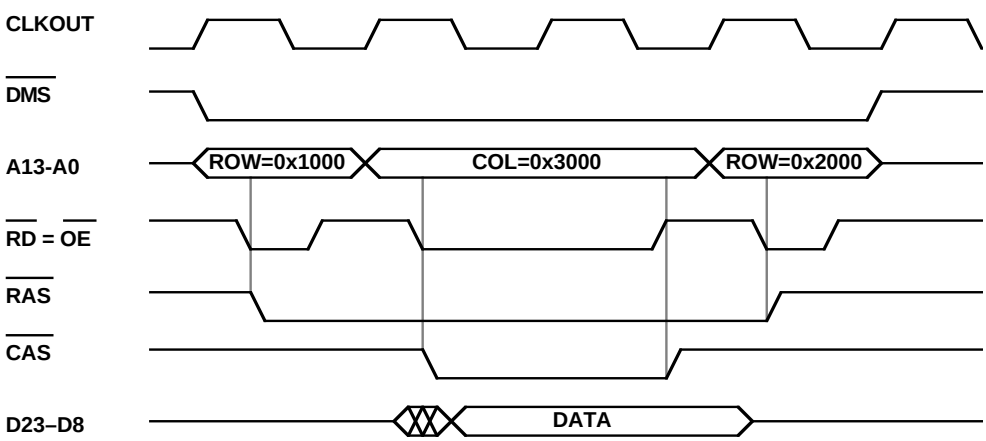

Figure 12.5 RAS & CAS Timing For DRAM Read

To accomplish the read cycle, execute a dummy read to a DM location (0x1000 to 0x13FF) to select a row and generate RAS. Follow this cycle with a read from external DM (0x3000 to 0x33FF), which selects a column, makes data available on bus lines, and latches it on the DSP. To finish the read cycle, perform a dummy read to a DM location (0x2000) to drive the RAS line high again.

Implemented in ADSP-2100 family assembly code, a DRAM read requires four DSP cycles. A fifth cycle (nop) is needed when consecutive reads are performed to assure RAS precharge timing is met. The following example illustrates a read from row address 0xABC and column address 0xDEF:

```
DRAM_read:
      ax0=DM(0x1ABC);          /* dummy read sets !RAS, ROW addr */
      ay0=DM(0x3DEF);          /* !CAS set, ay0=DRAM data */
      ax0=DM(0x2000);          /* dummy read to deselect !RAS */
      nop;                     /* necessary for precharge time */
```

This may appear to be excessive overhead for a single memory read. You can achieve faster reads by using a DRAM with Fast-Page addressing (described later in this chapter).

12.3.4.2 DRAM Write Timing

A write cycle is similar to a read cycle. For a write to the DRAM, the DSP's WR line must be delayed to create the DRAM WRITE signal. The DRAM WRITE signal is generated from a clocked D-flip flop within the GAL16V8 with the ADSP-2101 WR signal as input. The clock is obtained from DSP's CLKOUT signal. Figure 12.6 shows the timing for generating the RAS and CAS signals to write to DRAM.

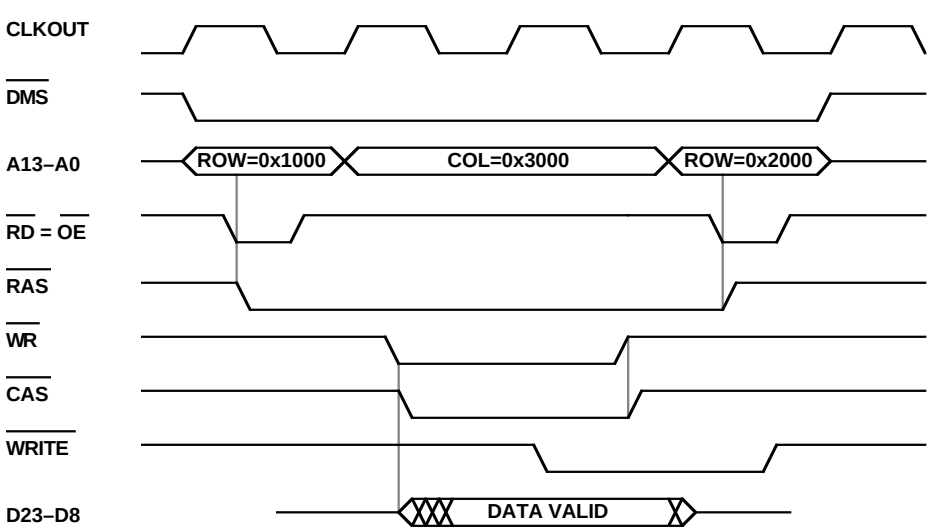

Figure 12.6 RAS & CAS Timing For DRAM Write

For a delayed-write (or output-enable write), the falling edge of WRITE latches the data into the DRAM.

In ADSP-2100 family assembly code, a DRAM write requires four DSP cycles. A fifth cycle (nop) is needed when consecutive writes are performed to assure RAS precharge timing is met. The following example shows you how to write to row address 0xABC and column address 0xDEF:

```
DRAM_write:
      ax0=DM(0x1ABC);          /* dummy read sets !RAS, ROW addr */
      DM(0x3DEF)=ay0;          /* !CAS set, ay0 written to DRAM */
      ax0=DM(0x2000);          /* dummy read to deselect !RAS */
      nop;                     /* necessary for precharge time */
```

12.3.5 Memory Access Modes

DRAMs support several memory access modes to help reduce memory access times. The access mode that you select depends on the modes supported by the particular DRAM used in your application. Memory access modes include: Page Mode, Enhanced or Fast Page Mode, Static Column Mode, and Nibble or Ripple Mode. This section describes two modes: page mode and fast page mode.

12.3.5.1 Page Mode

Page mode is the simplest memory access for a DRAM, but it also takes the longest time. To read one memory location requires a row and then a column access. A “page” is equivalent to a row.

Page-access provides quick access to memory locations in a page and can be accomplished by any DRAM. Page mode starts as a normal access when RAS is driven low. A memory access is accomplished by asserting CAS and performing a read or write. While keeping RAS low, you can access any other column location in the page by again asserting CAS. Since RAS remains low, the RAS precharge time is saved, which results in a faster access speed.

12.3.5.2 Enhanced Or Fast Page Mode

Fast page mode requires a DRAM that supports this feature. It is similar to page-mode access because you can access multiple columns within one page. However, it is faster than normal page mode because the column access is started as soon as the new address is placed on the DRAM address input. This saves the CAS precharge time.

Memory access in fast page mode is possible using the available signals. In this mode, memory reads or writes to dynamic memory are executed by leaving the RAS signal low after the row latch occurs. You can continuously fill columns in the same row. The only limitation is that RAS has a maximum pulse width timing requirement (for example 200 µs).

Multiple reads or writes are performed much faster if done within the same page. The following example shows you how to read three locations in one row;

```
DRAM_page_reads:
      ax0=DM(0x1ABC);          /* dummy read sets !RAS, ROW addr */
      ay0=DM(0x3123);          /* !CAS set, ay0=DRAM data */
      ay1=DM(0x3456);          /* !CAS set, ay0=DRAM data */
      ax1=DM(0x3789);          /* !CAS set, ay0=DRAM data */
      ax0=DM(0x2000);          /* dummy read to deselect !RAS */
      nop;                     /* necessary for precharge time */
```

12.3.6 DRAM Refresh

The DRAM stores data in capacitor cells. Since capacitors lose their charge, or “leak”, over time, each cell of the DRAM must be refreshed periodically to maintain adequate voltage levels. Each memory read actually causes the capacitors to discharge slightly, so DRAMs have built-in write-back features that follow a read. This write-back feature is called a “precharge”. The precharge occurs after the read when both RAS and CAS have returned high. It is part of the read cycle timing and you must account for it in timing analysis.

The read / write-back of the DRAM recharges the capacitor cells. The DRAM architecture refreshes every column cell when a row is accessed. Therefore, to accomplish a refresh of all memory cells, it is only necessary to read every DRAM row within the specified refresh period of your DRAM.

The RAS-only refresh is the simplest type of refresh operation and is supported by all DRAMs. To accomplish an RAS-only refresh, the RAS line is brought low one cycle for each row of the DRAM while the CAS remains high.

On the DRAM interface, this is done by reading from external data memory twice. One read sends RAS low and latches the row address, the other read brings RAS back high. CAS (DMS) remains high through the entire RAS cycle. The following example shows you one method to refresh DRAM.

```
DRAM_refresh:
      ax0=DM(0x1ABC);         /* dummy read sets !RAS, ROW addr */
      ax0=DM(0x2000);         /* dummy read to deselect !RAS */
```

Note: You may have to use a NOP instruction at the end of the refresh loop if the processor's instruction rate is less than 100 ns. The NOP instruction is necessary to meet the RAS precharge time and varies with DRAM access time (the DRAMs used in this application need a minimum RAS precharge time of 60 ns).

Some DRAMs have other refresh methods built-in. These include the hidden refresh, the CAS-before-RAS refresh, and refresh with scrubbing. For simplicity, this application concentrates on the RAS-only refresh.

12.3.7 DRAM Refresh Timing

Typical DRAM refresh periods are 4–16 ms. Depending on your application, you can refresh all the rows at one time, called a burst refresh, or interleave refreshes with normal memory accesses, called an interleaved refresh.

If you use a 1 M deep DRAM with a 16 ms refresh period, you must access each of the 1024 rows during the 16 ms interval. The overall time needed to refresh the DRAM is the same regardless of whether you choose burst or interleaved refreshes. To refresh 1024 rows, you need 1024 x 2 cycles since it takes one cycle to assert RAS and one cycle to deassert RAS. For a DSP running at 10 MHz, there is a 100 ns cycle time. So, your 2048 cycles will take 2048 x 100 ns, or about 0.2 ms.

If your application can afford to pause for 0.2 ms every 16 ms, then the burst method is the simplest. If not, consider breaking the 0.2 ms into more manageable pieces, for example ten refreshes of 0.02 ms each.

Typical real-time applications are based on some periodic input or cycle. You can use this timing to determine when to perform the refresh. Otherwise, you must generate the period interrupt yourself to ensure that refresh occurs to prevent data loss. With ADSP-2100 family DSPs, you can use the internal timer to generate this periodic input. If the timer is not available, use an external interrupt through IRQ2. If the external interrupt is not available, use a serial port to create a periodic interrupt even if the port is not being used for communication.

The following code example refreshes all 1M DRAM locations within 16 ms. Since a 1M DRAM consists of 1024 rows x 1024 columns, you need to access 1024 rows within the 16 ms. The code uses the DSP's internal timer to create a periodic interrupt that refreshes 256 rows every 4 ms. For this example, assume the DSP is running at 16.67 MHz (60 ns clock cycle).

To refresh every 4 ms using 60 ns cycles, you need to create a timer interrupt every 4 ms / 60 ns, or 66,667 clock cycles. If you set TSCALE to 2, TCOUNT is decremented every third clock cycle. Setting TCOUNT and TPERIOD to 22,222 or 0x56CE generates a timer interrupt every 66,666 cycles.

```
Timer_initialization:
      ax0=0x2                       /* TSCALE=2,  */
      dm(0x3ffb)=ax0;
      ax0=0x56CE                    /* TCOUNT=22,222 */
      dm(0x3ffc)=ax0;
      ax0=0x56CE                    /* TPERIOD=22,222 */
      dm(0x3ffd)=ax0;
```

During a timer interrupt, the timer interrupt service routine calls the following refresh subroutine.

```
/* Refresh for 256 rows

bank1: 000-0FF
bank2: 100-1FF
bank3: 200-2FF
bank4: 300-3FF    */

DRAM_timer_refresh:
      ax0=DM(bank);           /* determine which bank to refresh */
      ay0=0x3;                /* set ay0 for masking */
      ay1=1;                  /* set ay1 for incrementing */
      ar=ax0+ay0;             /* update for next bank */
      ar=ar and ay1;          /* mask upper bits (bank = 0-3) */
      DM(bank)=ar;            /* store new bank for next refresh */
      sr=lshift ar by 8 (lo); /* left shift bank by 8 */
      M0=SR0;
      I0=0x1000;              /* start of RAS addresses */
      modify(I0,M0);          /* offset to start of bank */
      M1=1;
      CNTR=256;
```

```
do refresh until ce;
      ax0=DM(I0,M1);    /* dummy read sets !RAS, ROW addr */
      ax0=DM(0x2000);   /* dummy read to deselect !RAS */
      refresh: nop;     /* end of loop (nop for precharge) */
rts;
```

12.3.8 EZ-LAB Implementation

To illustrate a DRAM interface to a DSP, this application uses the ADSP-2101 EZ-LAB evaluation board and a DRAM expansion card. The 2101 EZ-LAB has an ADSP-2101 processor and 64k x 8 EPROM. The EPROM is used only for booting the internal program RAM of the DSP. No additional data memory is on the board, however all data, address, and control lines are available through the connector.

To add DRAM to the EZ-LAB, the expansion card is connected to the EZ-LAB with a ribbon cable (see Figure 12.7). The DRAM interface card has one programmable logic device (GAL16V8), and four 1M x 4 DRAMs, (TC514400). Since the ADSP-2101 is a 16-bit fixed point DSP, the application needs four 1M x 4 DRAMs for 16-bit wide memory. The 16V8 PAL provides the glue logic to create the DRAM control signals, RAS, CAS, and WRITE.

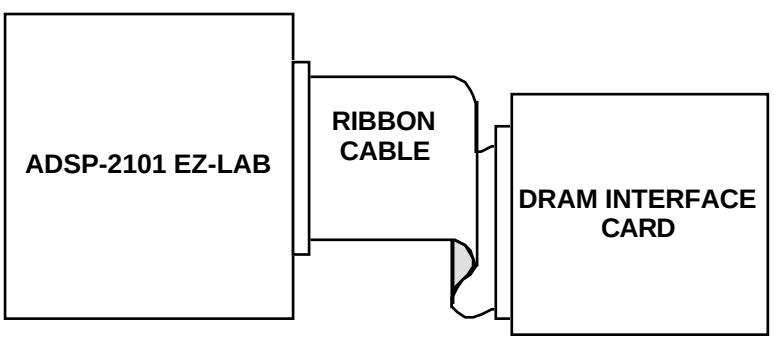

Figure 12.7 EZ-LAB/DRAM Interface Board Connection

12.3.9 DRAM Program Listings

This section contains several program listings that were tested on the application described in this chapter.

```
.module/ram/boot=0      DRAM_read;
/*
File Name:  DRAMRD.DSP
Version:    Version 0.00
Purpose:    Conducts a random read from DRAM
Calling parameters:
            setras: points to read location for RAS deassertion
            i3: points to the row being accessed
            i2: points to the column being accessed
Return Values
            ax1: contains data stored in location (row = i3, column = i2)
Registers affected:
            ax0, ax1, i2
Computation Time:
            5 cycles
            Random read from DRAM = 4 cycles ( dwait3 = 1)
*/

.external setras;
.entry read;

read: ax0 = dm(i3, m3);                  /* select row for DRAM access */
      ax1 = dm(i2, m2);                  /* read from column pointed by i2 */
      ax0 = dm(setras);                  /* deassert RAS */
      rts;

.endmod;
```

Listing 12.5 DRAM Read Program

```
.module/ram/boot=0      DRAM_write;
/*
File Name:  DRAMWR.DSP
Version:    Version 0.00
Purpose:    Conducts a random write to DRAM
Calling parameters:
            setras: points to read location for RAS deassertion
            i3: points to the row being accessed
            i2: points to the column being accessed
            mx0: contains the data that requires to be written
Return Values:
      data in mx0 stored in location (row = i3, column = i2)
Registers affected:
            ax0, i2
Computation Time:
            5 cycles
            Random (OE controlled write) to DRAM = 4 cycles ( dwait3 = 1)
*/

.external setras;
.entry write;

write: ax0 = dm(i3, m3);            /* read selects row to be accessed */
      dm(i2, m2) = mx0;             /* write to column pointed by i2 */
      ax0 = dm(setras);             /* deassert RAS */
      rts;

.endmod;
```

Listing 12.6 DRAM Write Program

```
.module/ram/boot=0      DRAM_refresh;
/*
File Name:  DRAMREF.DSP
Version:    Version 0.00
Purpose:    Conduct memory accesses to DRAM for refresh
Calling Parameters:
            refcntr: has the number of rows to refresh
            setras: points to the read location for RAS deassertion
            i6: contains the address of the first row to refresh
Return Values:
            Refresh of the number of rows specified by refcntr
Registers Affected:
            ax0, i6
Computation Time:
            Total execution time = 3 + cntr(3) cycles;
            RAS-only refresh per row = 3 cycles
*/

.external refcntr;
.external setras;
.external row;
.entry refresh;

refresh: cntr = dm(refcntr);              /* number of rows to refresh */
      do rasonly until ce;
      ax0 = dm(i6, m6);                   /* refresh row i6 */
      ax0 = dm(setras);                   /* deassert RAS */
rasonly: nop;                             /* inserted to meet RAS precharge */
      rts;

.endmod;
```

Listing 12.7 DRAM Refresh Program

```
.module/boot=0/abs=0    DRAM_test;
/*
File Name:  DRAMTST.DSP
Version:    Version 0.00
Purpose:    Assembly source code for Dynamic memory test. This code was
            written for the purpose of testing the DRAM interface board using
            the EZ-ICE emulator. It accomodates testing of refresh as well as
            independence of address lines.

            It is designed with two sections, the first fills all of the
            external DRAM and the second section reads and tests stored values
            for errors.

            This routine also sustains the DRAM storage using a timer
            implemented burst refresh every 16ms.
Return Values:
            buffer of 256 locations in internal data memory of error values
            followed by the actual data values that should have been read back
            from the DRAMs.
*/

.const rows       = 1024;                /* number of rows */
.const cols       = 1024;                /* number of columns */
.const ref_rows   = 1024;
.const maxfill    = 0xffff;              /* maximum fill value */
.var/dm/ram/seg=int_dm nbr_err, err_row, err_col, err_ov, buffer, refcntr;
.var/dm/ram/seg=row_deassert setras;
.var/dm/ram/seg=row_range row;
.var/dm/ram/seg=col_range col;
.global row, setras, refcntr;

/* _____________ Interrupt vectors _____________ */
      jump main; nop; nop; nop;                 /* reset interrupt */
      rti; nop; nop; nop;                       /* irq2 */
      rti; nop; nop; nop;                       /* sport0 transmit */
      rti; nop; nop; nop;                       /* sport0 receive */
      rti; nop; nop; nop;                       /* sport1 transmit or irq1 */
      rti; nop; nop; nop;                       /* sport1 receive or irq0 */
      i6 = ^row; m6 = 0;
      call refresh; rti;                        /* timer expired */

/* _____________ External functions _____________ */
.external refresh, write, read;
```

(listing continues on next page)

```
/* _________________ M a i n ____________________ */
main: call setup;
      ax0 = 0;                      /* initialize DM variables */
      dm(nbr_err) = ax0;            /* number of errors encountered */
      dm(err_ov) = ax0;             /* number of errors plus nbr_err */
      dm(err_row) = ax0;            /* saves row location of one error */
      dm(err_col) = ax0;            /* saves col loc. of one error */
      ifc = 0x3f; nop;              /* clear all pending interrupts */
      icntl = b#00111;              /* interrupts are edge sensitive*/
      ax0 = ref_rows;
      dm(refcntr) = ax0;            /* initialize refresh counter */
      i3 = ^row;
      m3 = 1;
      l3 = 0;                       /* init. row pointer */
      i2 = ^col;
      m2 = 1;
      l2 = 0;                       /* init. column pointer */
      i4 = ^buffer;
      m4 = 1;
      l4 = 0x200;                   /* init. error buffer */
      imask = b#000001;             /* enable timer irq*/
      mstat = b#0100000;            /* begin decrementing */

/* Fill routine - This routine fills all of memory with values in the range of
      0 to 0xffff using random writes (non-fast-page mode) so that address
      line errors can be debugged. */

      ax0 = dm(setras);             /* deassert RAS initially */
      cntr = rows;                  /* initialize variables */
      mx0 = 0;                      /* for nested fill loop */
      ay1 = 1;                      /* increment value for fill counter*/
      m3 = 0;
      mr1 = 0;
      do rowfill until ce;
            i2 = ^col;                    /* reset column pointer */
            cntr = cols;                  /* reset cntr to number of colums */
            do colfill until ce;
            imask = 0;                    /* disable timer during write */
                  call write;
                  imask = 1;
                  ax1 = maxfill;
                  ay0 = mx0;              /* mx0 is the fill counter */
                  ar = ay0 + 1;           /* compare with max. fill value */
                  af = ax1 - ay0;
                  if eq ar = pass af;     /* if mx0 = 0xffff, reset to 0 */
            colfill: mx0 = ar;
rowfill: modify(i3,m2);             /* increment row pointer */
```

```
/* Check routine - reads stored values and compares them to actual filled
      values, if a difference is encountered in the read value it is
      accumulated in the error routine, which stores actual and erroneous
      values. */

reread: i3 = ^row;                          /* reset row pointer */
      i2 = ^col;                            /* reset column pointer */
      mr0 = 0;                              /* mr0 = check counter */
      cntr = rows;                          /* cntr = number of rows */
      do rowread until ce;
            i2 = ^col;
            cntr = cols;
            do colread until ce;
                  imask = 0;                /* disable timer during read */
                  call read;
                  imask = 1;
                  ay0 = mr0;
                  ar = dm(nbr_err);
                  af = ax1 - ay0;
                  if ne call error;
                  ax1 = maxfill;
                  ar = ay0 + 1;
                  af = ax1 - ay0;
                  if eq ar = pass af;       /* if mr0 = 0xffff, reset it to 0 */
      colread: mr0 = ar;
      rowread: modify(i3,m2);               /* increment row pointer */

      sr1 = dm(nbr_err);                    /* sr1 + mr0 = number of errors */
      mr0 = dm(err_ov);
      ar = sr1;
      ar = pass ar;                         /* this tests refresh ability */
      if eq jump reread;                    /* continually read until error */

wait: idle;                           /* set breakpoint here in emulation */
      jump wait;
/* _________________ End Main _________________ */

/* _______________ Subroutines _______________ */
error: ay1 = maxfill;                       /* if an error occurs */
      ar = dm(nbr_err);
      af = ar - ay1;                        /* add it to nbr_err or err_ov */
      if ne jump not_ov;
      ay1 = 1;
      ax1 = dm(err_ov);
      ar = ax1 + ay1;
      dm(err_ov) = ar;
      jump cont;
```

(listing continues on next page)

```
not_ov: ay1 = 1;
      ar = ar + ay1;
      dm(nbr_err) = ar;                    /* save 256 error values */

cont: dm(i4, m4) = ax1;                    /* saves error value followed by */
      dm(i4, m4) = ay0;                    /* actual value in internal DM */
      dm(err_row) = i3;                    /* saves last error row and */
      dm(err_col) = i2;                    /* column location */
      rts;

setup: ax0 = 0x0003;
      dm(0x3ffb) =ax0;                     /* TSCALE */
      ax0 = 0xc350;                        /* 50,000 for 16ms per interrupt */
      dm(0x3ffc) =ax0;                     /* TCOUNT */
      dm(0x3ffd) =ax0;                     /* TPERIOD */
      ax0=0x0200;
      dm(0x3ffe)=ax0;                      /* dWait3 = 1 WS */
      ax0=0x0000;
      dm(0x3ffa)=ax0;                      /* No Sport functions enabled */
      dm(0x3ff9)=ax0;
      dm(0x3ff8)=ax0;
      dm(0x3ff7)=ax0;
      dm(0x3ff6)=ax0;
      dm(0x3ff5)=ax0;
      dm(0x3ff4)=ax0;
      dm(0x3ff3)=ax0;
      dm(0x3ff2)=ax0;
      dm(0x3ff1)=ax0;
      dm(0x3ff0)=ax0;
      dm(0x3fef)=ax0;
      dm(0x3fff)=ax0;
      rts;

.endmod;
```

Listing 12.8 DRAM Test Program

```
.module/boot=0/abs=0    DRAM_record;
/*
File Name:  DRAMRCRD.DSP
Version:    Version 0.00
Purpose:    Assembly source code for u-law companded speech sample
            storage and playback. This routine enables approximately
            2 minutes and 11 seconds of storage on the DRAM interface
            board. The maximum storage time possible is twice that
            mentioned above using u-law companding (8-bit word length).
Return Values:
            IRQ2 - switch between record and forward playback
            FLAG_IN - switch between foward and backward playback.
            Note: pressing FLAG_IN in record mode causes backward
            playback.
            record mode - flag_out is low
            playback mode - flag_out is high
*/
.const  rows      = 0x400;    /* # of rows and columns on 1M DRAM */
.const  cols      = 0x400;
.const  lastrow   = 0x13ff;
.const  lastcol   = 0x33ff;
.const  ref_rows  = 0x10;
.var/dm/ram/seg=int_dm        lastr, lastc, sflag, flagin, refcntr; .var/dm/ram/
seg=row_deassert  setras;     /* location for RAS deselection */
.var/dm/ram/seg=row_range     row;        /* row selection range */ .var/dm/ram/
seg=col_range     col;        /* column selection range */
.global row, setras, refcntr;

/* _____________ Interrupt vectors _____________ */
      jump main; nop; nop; nop;           /* reset interrupt */
      jump changestate; nop; nop; nop;    /* irq2 */
      rti; nop; nop; nop;                 /* sport0 transmit */
      jump record; nop; nop; nop;         /* sport0 receive */
      rti; nop; nop; nop;                 /* sport1 transmit or irq1 */
      rti; nop; nop; nop;                 /* sport1 receive or irq0 */
      rti; nop; nop; nop;                 /* timer expired */

/* _____________ External functions _____________ */
.external refresh, read, write;

/* _____________ Interrupt handlers _____________ */
changestate: ax0 = 0;               /* pressing IRQ2 causes state change */
      dm(flagin) = ax0;             /* clear backward playback mode */
      ax0 = dm(sflag);              /* if 0 make 1 = record mode */
      ar = pass ax0;
      if ne jump set_s1;            /* sflag = 0 = playback mode */
      ax0 = 0x0001;                 /* sflag = 1 = record mode */
      dm(sflag) = ax0;
      rti;
```

(listing continues on next page)

```
set_s1: ax0 = 0;                    /* if 1 make 0 = playback mode */
      dm(sflag) = ax0;
      dm(lastr) = i3;               /* save row location for wraparound */
      dm(lastc) = i2;               /* save column location for wraparound */
      rti;

record: ax0 = dm(flagin);           /* if flagin = 1 playback backwards */
      ar = pass ax0;
      if ne jump yalp;
      ax0 = dm(sflag);              /* if sflag = 0 playback forwards */
      ar = pass ax0;
      if eq jump play;
      reset flag_out;               /* flag_out = low */
      mx0 = rx0;
      tx0 = mx0;
      call write;                   /* read a sample and write it to DRAM */
      call inc_loc;                 /* increment column/row pointer */
      ax0 = i3;                     /* compare inc'd location with */
      ay0 = lastrow;                /* ...the last row and column */
      ar = ax0 - ay0;
      if le jump refr;
      ax0 = i2;
      ay0 = ^col;
      ar = ax0 - ay0;
      if ne jump refr;
      ax0 = 0;                      /* if the last row and column available */
      dm(sflag) = ax0;              /* has been reached save the current */
      ax0 = lastrow;                /* row and column location for */
      dm(lastr) = ax0;              /* wraparound */
      ax0 = lastcol;
      dm(lastc) = ax0;
      jump refr;

play: call read;                    /* playback forward */
      tx0 = ax1;                    /* read from DRAM and output sample */
      call inc_loc;
      ax0 = dm(lastr);              /* compare to last column and row, */
      ay0 = i3;                     /* if so set to row and column 0 */
      ar = ax0 - ay0;
      if ne jump refr;
      ax0 = dm(lastc);
      ay0 = i2;
      ar = ax0 - ay0;
      if ne jump refr;
      i3 = ^row;
      i2 = ^col;
      jump refr;

yalp: call read;                    /* playback backward */
```

```
      tx0 = ax1;                    /* read from DRAM and output sample */
      ax0 = i2;                     /* decrement the colunm/row pointer */
      ay0 = ^col;
      ar = ax0 - ay0;
      if ge jump done;
      i2 = lastcol;
      modify(i3, m2);

done: ax0 = i3;                     /* compare to row and column 0, if so */
      ay0 = ^row;                   /* reset to last row and last column */
      ar = ax0 - ay0;
      if ge jump refr;
      ax0 = i2;
      ay0 = lastcol;
      ar = ax0 - ay0;
      if ne jump refr;
      i3 = dm(lastr);
      i2 = dm(lastc);

refr: call refresh;
      ar = i6;
      ay1 = lastrow;                /* if last row to refresh has been */
      af = ar - ay1;                /* reached reset it to row 0 */
      if lt jump cont;
      i6 = ^row;

cont: rti;

/* ___________________ M a i n ____________________ */
main: call setup;                   /* initialize sport registers */
      ax0 = 0;                      /* initialize state variables */
      dm(sflag) = ax0;
      dm(flagin) = ax0;
      ax0 = ref_rows;               /* initialize refresh counter */
      dm(refcntr) = ax0;
      i6 = ^row;
      m6 = 1; l6 = 0;               /* init. refresh row pointer */
      i3 = ^row;
      m3 = 0; l3 = 0;               /* init. row pointer */
      i2 = ^col;
      m2 = 1; l2 = 0;               /* init. column pointer */
      ax0 = dm(setras);             /* initially deassert RAS */
      ifc = 0x3F; NOP;
      icntl = b#00111;              /* interrupts are edge sensitive */
      imask = b#100000;             /* enable irq2 */
      idle;                         /* wait for irq2 */
```

(listing continues on next page)

```
state_1: reset flag_out;            /* flag_out = low (indicate record) */
      ax0 = 0;
      dm(flagin) = ax0;
      i3 = ^row;
      i2 = ^col;
      m2 = 1;
      imask = b#101000;             /* enable irq2 and sp0 receive */

wait1: idle;                        /* receive until flag_in or irq2 */
      ax0 = dm(sflag);              /* or last column and row */
      ar = pass ax0;                /* check mode (record/playback) */
      if eq jump playback;
      if not flag_in jump f1;       /* if not_flagin play backwards */
      jump wait1;

f1:   if not flag_in jump f1;       /* first debounce flag_in */
      dm(lastr) = i3;               /* save row location for wraparound */
      dm(lastc) = i2;               /* save column location for wraparound */
      jump kabyalp;

playback: set flag_out;             /* flag_out = high (indicate playback) */
      ax0 = 0;
      dm(flagin) = ax0;
      m2 = 1;                       /* reset modify value to + 1 */
      i3 = ^row;                    /* reset to row and col 0 */
      i2 = ^col;

wait2: idle;                        /* wait for irq2 or flag_in */
      ax0 = dm(sflag);
      ar = pass ax0;                /* if sflag = 0 = record */
      if ne jump state_1;
      if not flag_in jump f3;       /* if FLAG_IN play backwards */
      jump wait2;

f3:   if not flag_in jump f3;       /* first debounce flag_in */

kabyalp: set flag_out;              /* set up for backward play */
      m2 = -1;                      /* modify for column = -1 */
      ax0 = 0;                      /* signify playback mode */
      dm(sflag) = ax0;
      ax0 = 1;                      /* signify play backwards mode */
      dm(flagin) = ax0;
      i3 = dm(lastr);               /* init. to last row and col. location */
      i2 = dm(lastc);

wait3: idle;                        /* wait for irq2 or flag_in */
      if not flag_in jump f5;       /* switch back to playback */
      ax0 = dm(sflag);
      ar = pass ax0;                /* if sflag = 0, record again */
```

```
      if ne jump state_1;
      jump wait3;

f5:   if not flag_in jump f5;       /* first debounce flag_in */
      jump playback;
/* _________________ End Main ____________________ */

/* _____________ Subroutines ___________________ */
inc_loc:    ax0 = i2;               /* increment column and/or row pointer */
      ay0 = lastcol;
      ar = ax0 - ay0;
      if le jump done1;
      i2 = ^col;
      modify(i3, m2);

done1: rts;

setup: ax0 = 0;
      dm(0x3ffb) =ax0;              /* TSCALE */
      dm(0x3ffc) =ax0;              /* TCOUNT */
      dm(0x3ffd) =ax0;              /* TPERIOD */
      ax0=0x0200;
      dm(0x3ffe)=ax0;               /* Dwait3 = 1 WS */
      ax0=0x0000;
      dm(0x3ff9)=ax0;               /* Disable Receive Multichannels */
      dm(0x3ffa)=ax0;
      dm(0x3ff7)=ax0;               /* Disable Transmit Multichannels */
      dm(0x3ff8)=ax0;
      ax0=0x6b27;                   /* Multichannel disabled */
      dm(0x3ff6)=ax0;               /* Int. gen serial clock */
                                    /* Receive frame sync required, width 0 */
                                    /* Transmit frame sync required, width 0 */
                                    /* Int trans, receive frame sync enabled */
                                    /* u-law companding, 8 bit word length */
      ax0=0x0002;
      dm(0x3ff5)=ax0;               /* Generate 2.048 MHz serial clock */
      ax0=255;
      dm(0x3ff4)=ax0;               /* Divide by 256 for 8KHz sampling rate */
      ax0=0x0000;
      dm(0x3ff3)=ax0;               /* SPORT0 AUTOBUFF disabled */
      dm(0x3ff2)=ax0;               /* SPORT1 CNTL disabled */
      dm(0x3ff1)=ax0;               /* SPORT1 timer not used */
      dm(0x3fef)=ax0;               /* SPORT1 AUTOBUFF disabled */
      ax0 = 0x1000;
      dm(0x3fff)=ax0;               /* SPORT0 enabled, No PM Wait States */
                                    /* BOOT Wait State 0, BOOT page 0  */
      rts;

.endmod;
```

Listing 12.9 DRAM Speech Sample Record/Playback Program

12.3.10 DRAM Interfacing References

Analog Devices:
ADSP-2100 Family User's Manual
ADSP-2100 Family EZ-Tool Manual
ADSP-2100 Family Assembler Tools & Simulator Manual
Digital Signal Processing Laboratory
Using the ADSP-2101 Microcomputer
ADSP-2101 and ADSP-21msp50 data sheets

Data I/O Corp.,
Abel Design Software User's Manual, 1990.

Driscoll, Frederick F.,
"Dynamic Refresh", *Interfacing the 68000 Microprocessor*
pgs. 358-65, 173-177.

Clements, Alan,
"Designing Dynamic Read/Write RAM Systems", Microprocessor Systems Design, pg. 275-93.

Steve Gumm, Carl T. Dreher,
"Unraveling the Intricacies of Dynamic RAMs", Dynamic RAMs Part 1, pg. 162.

Toshiba, Signetics, SGS Thomson data sheets.

12.4 LOADING AN ADSP-2101 PROGRAM VIA THE SERIAL PORT

For many DSP applications, it is desirable to have a DSP processor under the control of a host computer. In these situations, the host computer would download a program for the DSP to execute. The ADSP-2101 provides two serial ports suitable for program download from a host computer. This section note details the ADSP-2101 monitor program for downloading from a serial port. The monitor program itself would be booted from EPROM or other boot memory. While this example uses serial port zero, the principal could be extended to download via a memory-mapped parallel port.

12.4.1 A Monitor

The task of the host computer is to download a series of instructions to the ADSP-2101 for execution. The ADSP-2101 receives the incoming instructions, loads them into program memory and when all instructions have been received, executes them. Prior to and during the download from the host, the ADSP-2101 executes a monitor program. This monitor activates the serial port, receives the instructions and places them in program memory for execution.

The ADSP-2101 instruction is twenty-four bits wide but many hosts, including eight-bit processors, more readily handle byte-wide data. Since the serial port can accommodate serial words from three to sixteen bits in length, byte-length data words are easily received.

Whenever a program memory write occurs, the sixteen most significant bits are supplied by the source register, explicitly named in the instruction, and the eight LSBs are supplied by the PX register. The basic tactic of the monitor program is to assemble the two most significant bytes in a data register (using the Shifter) and load PX explicitly with the least significant byte. A program memory write then writes the correct twenty-four bit instruction.

In addition to the transfer of instructions through the serial port into program memory, the monitor program must also know when the download is complete and execution can begin. A straight forward method is to count the number of instructions sent to the serial port. A count value is sent to the ADSP-2101 before the first instruction. This is the count of the instructions to follow. After each instruction is downloaded, the count can be decremented.

The downloaded program must avoid overwriting the monitor program while the monitor executes. The last instruction of the monitor program is identified by a global label which also identifies the beginning of the available space for downloaded code. The monitor program must be linked with the downloaded program so that the downloaded program makes the correct address references including the reference to this global label.

The indirect addressing capabilities of the Data Address Generators on the ADSP-2101 make it easy to cycle through the correct sequential locations starting with the label.

The final concern is the interrupt table. If the downloaded program is interrupt-driven, the interrupt table (program memory H#0000 to H#001C) must contain valid instructions for servicing expected interrupts.

There are several ways to do this. First, the monitor program itself could contain the valid interrupt table for the program to be downloaded. This assumes that the interrupt structure of the downloaded program is known when the monitor program is created. Second, the interrupt table may be downloaded through the serial port just as the rest of the program is. The DAG can loaded with the start address of the interrupt table and the instructions can be loaded, but you may not overwrite the interrupt being used to receive the data on the serial port until all instructions have been received.

The monitor program example does not load an interrupt table. The best approach is dependent on your application.

12.4.2 Implementation

The first task of the monitor program is to setup and enable the serial port. Serial ports on the ADSP-2101 are extremely flexible in terms of framing options, word lengths and timing. The ADSP-2101 serial ports may receive the frame synch and serial clock from the host processor or generate them internally.

As the program is downloaded from a host computer, the ADSP-2101 looks to the host for serial port information. That is, the serial port frame synchronization and serial port clock are supplied by the host computer. For purposes of illustration, the code that appears at the end of this section uses normal framing and external receive frame synchronization. For externally generated serial clocks the ADSP-2101 can support frequencies up to the processor instruction rate.

The flow for the monitor program is shown in Figure 12.8.

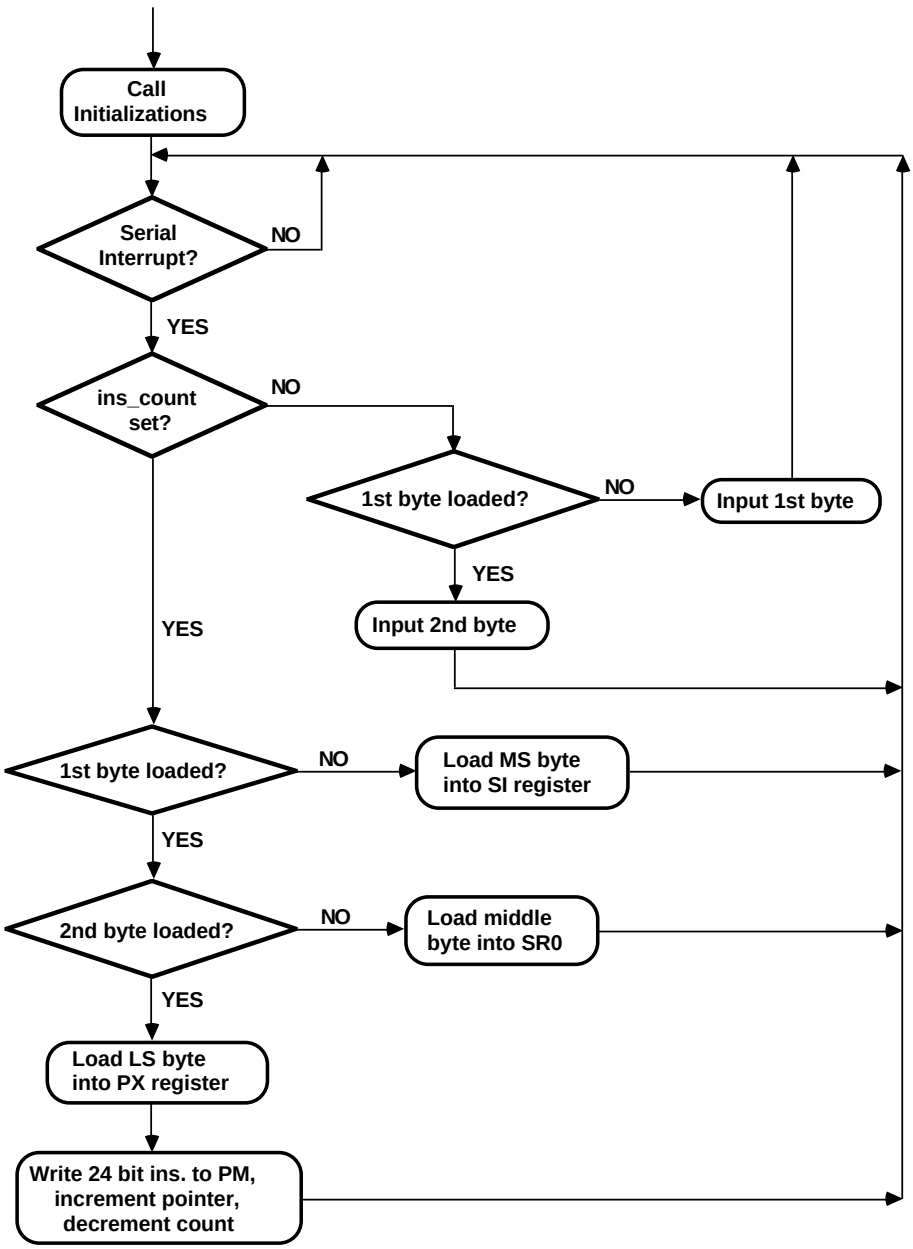

Figure 12.8 Boot Program Flow Diagram

Once the serial ports are enabled, the monitor program waits for a serial port interrupt signifying that a serial word has been received. The first two serial words received are the instruction count. As the serial word is eight bits, two serial port words make up the instruction count. The separate bytes of the instruction count are combined in the shifter and loaded into data memory. This count represents the number of instructions to be downloaded from the host and does not include the interrupt table. The interrupts are handled automatically, as the interrupt table has a fixed length.

With the count downloaded, the ADSP-2101 is ready to accept instructions through the serial port. Instructions are downloaded a byte at a time just as the instruction count was. The most significant byte is first. It is loaded into the SI register and the byte count ("count") is decremented. The middle byte of the instruction is loaded into the SR0 register. These two bytes are combined in the shifter with the results residing in the SR0 register. Once again the byte count is decremented. Finally, the least significant byte is loaded into the PX register. Now that all three bytes are loaded into registers on the ADSP-2101, the downloaded instruction can be written to program memory.

When all is downloaded, a jump to the new downloaded program is all that is necessary to begin execution.

The monitor program is shown in Listing 12.10.

A monitor program initializes the serial port and receives instructions, writing them into program memory, then beginning execution. This method of booting is useful when the ADSP-2101 is under the control of a host computer or controller. Any size program may be downloaded (up to the full addressing capability of the ADSP-2101) with this particular method of implementation. Only the program memory used by the monitor program (eighty-six instructions) cannot be loaded. That space could, however, be used for program memory data storage.

```
.Module/ram/BOOT=0              serial_boot_monitor;
.Var/dm                         count;            {counts bytes}
.VAR/DM                         ins_count;        {counts instructions}
.GLOBAL                         code_start;       {end of monitor space}


                    JUMP restarter; NOP; NOP; NOP;  {restart vector}
                    RTI; NOP; NOP; NOP;             {IRQ2 not used}
                    RTI; NOP; NOP; NOP;             {sport0 TX not used}
                    JUMP serial; NOP; NOP; NOP;     {sport0 RX }
                    RTI; NOP; NOP; NOP;             {sport1 TX not used}
                    RTI; NOP; NOP; NOP;             {sport1 RX not used}
                    RTI; NOP; NOP; NOP;             {no timer used}

restarter:          CALL initializations;
wait_loop:          IDLE;
                    JUMP wait_loop;

initializations:    I4 = H#3ff3;                    {pointer to mem map reg}
                    I5 = PM(^code_start);           {pointer to start of prog}
                    I6 = 0000;                      {pointer to interrupt tab}
                    M4 = 0;
                    M5 = 1;

                    DM(count) = 1;                  {count val for # of bytes}

                    DM(I4,M5) = 0;                  {disable autobuffer}
                    DM(I4,M5) = 0;                  {no frame divide modulus}
                    DM(I4,M5) = 0;                  {no clk divide modulus}
                    DM(I4,M5) = H#2007;             {extrnl RFS & SCLK, no compand}
                                                    {SLEN 8, no multichannel}
                    DM(H#3fff) = H#1000;            {enable sport0}

                    DM(ins_count) = H#FFFF;

                    imask = 8;                      {sport0 rec interrupt only}
                    RTS;

serial:             AY1 = DM(ins_count);
                    AR = PASS AY1;
                    IF GT JUMP next_instruction;    {get next instruction}
                    IF LT JUMP load_word_count;     {get number of instructions}
                    IF EQ IMASK=0;                  {done; turn off interrupts}
                    JUMP code_start;                {start downloaded program}

{load the count, that is, the number of instructions to be downloaded}
{this happens in two bytes  The most significant byte first}
```

(listing continues on next page)

```
load_word_count:    AY0 = DM(count);                {is this 1st or 2nd byte}
                    AR = PASS AY0;
                    IF NE JUMP first_byte;
                    IF EQ JUMP second_byte;

first_byte:         SI = RX0;                       {first byte decrem. count}
                    AR = AY0 - 1;
                    DM(count) = AR;
                    RTI;

second_byte:        SR0 = RX0;                      {second byte...}
                    SR = SR OR LSHIFT SI BY 8 (LO); {put two bytes together}
                    DM(ins_count) = SR0;            {store in ins_count}
                    DM(count) = 3;                  {load count for ins.}
                    RTI;

{load the next instruction.  Instructions are 24 bits long and appear}
{at the serial port in 8 bit fragments.  The most significant byte 1st}

next_instruction:   AX0 = 2;                        {decide which byte is due}
                    AY0 = DM(count);
                    AR = AX0 - AY0;

                    IF LT JUMP most_sig_byte;
                    IF EQ JUMP middle_byte;
                    IF GT JUMP least_sig_byte;

most_sig_byte:      SI = RX0;                       {load MS byte into SI}
                    AR = AY0 - 1;                   {decrement count}
                    DM(count) = AR;
                    RTI;

middle_byte:        SR0 = RX0;                      {load Middle into SR}
                    SR = SR OR LSHIFT SI BY 8 (LO); {put MS and middle together}
                    AR = AY0 - 1;                   {decrement count}
                    DM(count) = AR;
                    RTI;

least_byte:         PX = RX0;                       {put LS byte into PX}
                    PM(I5,M5) = SR0;                {write SR0 into PM}
                                                    {PX provides 8 LS bits}
                    DM(count) = 3;                  {reset byte count}
                    AR = AY1 - 1;                   {decrement ins count}
                    DM(ins_count) = AR;
                    RTI;

code_start:         NOP;

.ENDMOD;
```

Listing 12.10 Monitor Program Listing

12.5 MEMORY INTERFACING FOR THE ADSP-2105

In order to utilize the full potential of the ADSP-2105, several techniques for memory allocation and development tools usage will be discussed. The techniques and examples described here can also be applied to any of the processors in the ADSP-2100 family.

This section will describe several example systems, along with the correct development tool set-ups to implement them. Listings of example architecture descriptions, module declarations, and linker-output map files will be provided for each example. A description of C-compiler use for each case will follow the examples.

12.5.1 Example System1: Using Boot Pages For Program Memory

An ADSP-2105 system may contain up to eight pages of boot memory. ADSP-2105 boot pages are 1K long, as opposed to ADSP-2101/2111 boot pages which are 2K long.

To use boot pages, the following steps should be taken.

Builder	In your .SYS file, use the ADSP2105 directive. Define BOOT ROM pages as needed for your system.
Assembler	Define each module as bootable. (.MODULE / boot=x, where x=0-7)
Linker	Use linker as you normally would.
Splitter	Use the **-bs**[1024] option. This sets the generated boot page size to 1024.

The example system provided uses all eight boot pages. Each module performs a dummy task and then boots in the next page. Note that in the .map file (*ex1.map*), the data memory declared for each overlaps. This is because each boot page is treated as a separate run-time context by the processor. Each boot page is an entirely different program with no relation to any other program; thus memory space is reused for each boot page. If there is a need to share variables between boot pages, the variables must be defined as **static** variables when declared. This forces the linker to place these variables in high memory and not to overwrite them when booting in a new page.

12.5.2 Example System 2: Booting With The -loader Option (RAM Initialization)

Like Example 1, this system uses the concept of boot pages to store the executable code. What is different in this system is the automatic initialization of external PM RAM as well as internal and external DM RAM. This initialization is done through the use of the **-loader** option when using the Prom Splitter (see *ADSP-2100 Family Development Software, Release 3.1 Release Note* for a description of the -loader option operation).

In order to use the **-loader** option, the following steps must be taken.

Builder	In your .SYS file, define internal PM and DM RAM segments, as well as any desired external PM and DM RAM segments. Define BOOT ROM pages (but don't assign any modules to it). Boot page 0 will be filled with the splitter-generated loader code. User code will be placed in subsequent boot pages (1 and higher if needed).
Assembler	DO NOT use the "BOOT = xx" qualifier in any module description. (Modules may, however, be defined to reside in internal PM RAM segments.)
Linker	Use linker as you normally would. Linker will place modules according to their definition and the architecture description.
Splitter	Use **-loader** and **-bs**[1024] (and -bb[ ] if needed) options. The splitter will create a bootable image file which will load program memory with the appropriate modules, and data memory with appropriate values, according to their definitions.

Do not use the -pm, -dm, or -bm switches when using the -loader option.

The boot-page qualifiers are removed from the module declarations because, on chip reset, the **loader** option causes special code to be booted into internal PM RAM which copies the ROM's contents into the appropriate RAM spaces. The executable code will be written into internal PM RAM when the initialization procedure is complete. This operation is transparent to the user and system operation on startup will appear to be the same as for Example 1, except that RAM will be initialized

automatically. The linker output file (.BNM image file) should be used to burn boot memory PROMs. The bootable image file will contain multiple pages. The -loader routine will be placed on boot page 0. User modules will be placed in ROM starting on boot page 1.

12.5.3 Example System 3: Using Internal & External PM RAM For Code

If your particular application requires code that is longer than the 1K internal PM RAM limit of the ADSP-2105, do not despair. The 2100 family PROM splitter **-loader** option will allow these types of programs to be used with the ADSP-2105. As described in the previous section, the **loader** option will initialize external memory. This feature can be used to initialize external PM RAM with executable code while still using the internal PM RAM to its fullest extent.

System development should follow the following steps.

Builder	In your .SYS file, define sufficient internal and external PM RAM space. Define BOOT ROM pages (but don't assign any modules to it) sufficient for the **-loader** code and your code.
Assembler	DO NOT use the "BOOT = xx" qualifier in any module description. (Modules may, however, be defined to reside in internal PM RAM segments.)
Linker	Use linker as you normally would. Linker will place modules according to their definition and the architecture description.
Splitter	Use **-loader** and **-bs**[1024] options. The splitter will create a bootable image file which will load program memory with the appropriate module according to their definitions.

Do not use the -pm, -dm, or -bm switches when using the -loader option.

This strategy is especially useful if your application software can be subdivided into *time-critical* and *non-time-critical modules.* The time-critical modules should be set to reside in the internal PM segment, while non-time-critical modules can be placed in external PM RAM.

12.5.4 Example System 4: Using External PM ROM

Another useful method for partitioning code too big to fit in one page of memory is to use a combination of boot memory and external ROM. To accomplish this both internal and external ROM modules must be defined to reside in the same boot page. Since boot page size is limited to 1K, defining the modules this way forces the linker to place one module externally while keeping it in the same run-time context.

To implement this system, the following steps should be taken.

Builder	In your .SYS file, use the ADSP2105 directive. Define the necessary boot page. Define an appropriate PM ROM segment.
Assembler	Define each module to be bootable from boot page 0 (/boot=0).
Linker	Use linker as you normally would.
Splitter	Run the splitter twice--once to generate the BOOT ROM and once to generate the program memory ROM.

The example system provided uses one page of boot memory along with 1K of internal program ROM. The program ROM module (*ex4_2.dsp*) is declared to reside in program ROM **and** is defined with the /boot=0 directive. This qualifier insures *ex4_2.dsp* is included in the same run-time context as the module defined to reside in the boot ROM (*ex4_1.dsp*).

12.5.5 Hardware Implications

The ADSP-2105 allows several boot EPROM configurations to be used. Refer to the Memory Interface chapter of the *ADSP-2100 Family User's Manual* for an in-depth discussion of the boot memory interface.

For direct plug-in compatibility with an ADSP-2101, use the following connections (for 27512).

```
ADSP-2105 A0 - A13     =>  EPROM A0 - A13
ADSP-2105 D22 - D23    =>  EPROM A14 - A15
```

This allows for eight 1K boot pages on 2K boundaries.

Since the ADSP-2105 only uses 1K boot pages, it is possible to access all eight boot pages while using a smaller EPROM. To accomplish this, boot pages can be placed on 1K boundaries by using the Prom Splitter's **-bb**[1024] option and the following hardware connections (for 27256).

ADSP-2105 A0 - A11	=>	EPROM A0 - A11
ADSP-2105 A12	=>	**no connection**
ADSP-2105 A13	=>	EPROM A12
ADSP-2105 D22 - D23	=>	EPROM A13 - A14

This connection scheme is also useful with a 27512, where code for another processor can utilize the upper half of the EPROM.

Note: Using the -bb[1024] option and the above hardware connections is not compatible with the ADSP-2101. On boot-up, the ADSP-2101 will always load 2K worth of information from the EPROM. Therefore, this configuration cannot be used with any ADSP-2101 In-Circuit Emulator™ (ICE) if booting the ICE from the target system. To use an ICE in these systems, remove the target EPROM and download the executable code into the ICE's overlay memory. The same caveat holds true when simulating the bootable code in the ADSP-2101 simulator using the 'LR' command.

12.5.6 Use Of The C-Compiler With ADSP-2105 Systems

If your application software is derived from C code, the following options and strategies should be employed when using the C-compiler for the above example.

Example 1 Use the **-b#** option when compiling to assign each module to the appropriate boot page. For example, typing **cc21** *filename* **-b0** would assign the module *filename* to boot page 0. Also, modify the *run_hdr* program so that it will be assigned to the correct boot page (ex. MODULE/boot=0 run_hdr;). When re-assembling the modified *run_hdr* program, ALWAYS use the **-c** (case sensitivity) option.

Example 2 DO NOT use the compiler **-b#** option. The **-loader** option of the PROM splitter automatically generates the correct boot image file. Compile the module as you would normally.

Example 3 For memory usage considerations, the **-lpm** option can be used. This places literals into PM rather than the default, DM. This is not a requirement but may come in handy if memory space is tight. Again, when using the **-loader** option, DO NOT use the **-b#** option when compiling.

Example 4 Modify the *run_hdr* program to reside in boot page 0 (see Example 1). For module 1 (INTERNAL), compile with the **-b0** option. For module 2 (EXTERNAL), compile with both the **-b0** and **-crom** options. This will assign both modules to the same run-time context and places PM modules into ROM segments.

12.5.7 Linking Modules Generated By The C-Compiler

For Examples 1, 3, and 4, use the **-p** option when linking. This places the run-time libraries on the correct boot pages. For Example 2, link without this option.

12.5.8 Additional Suggestions

If you find that you have memory limitations, such as the stack having no room to grow (examine the .MAP file generated by the linker's **-x** option), use the **-s###** option when you link to define the minimum stack size (default = 1). This is important because the linker allocates internal DM memory first, starting with the stack, until internal DM is filled. If you suspect that the stack will grow past its default boundaries, such that it may start to overwrite DM variables, give yourself some room using the **-s###** option. The linker will reserve the specified space and then proceed to fill internal memory until space is exhausted. The linker will then start to fill external DM RAM if it exists. The C-compiler does not allow you to explicitly define variables in external memory. External memory is allocated only after internal memory is filled.

12.5.9 About The Example Programs

The disk provided with this book contains sample architecture, assembler, and C files for each example system. Batch files are also provided to create final bootable image files for each example system so that their operation can be examined in the simulator or emulator. Please examine these files and feel free to use them as a framework for your own systems.

Example 1	System file	**ex1.sys**
	Source modules	**ex1_1.dsp, ex1_2.dsp, ex1_3.dsp, ex1_4.dsp, ex1_5.dsp, ex1_6.dsp, ex1_7.dsp, ex1_8.dsp**
	Batch file	**makeex1.bat**
	C source modules	**ex1_1.c, ex1_2.c, ex1_3.c, ex1_4.c, ex1_5.c, ex1_6.c, ex1_7.c, ex1_8.c**
	C batch file	**makeex1c.bat**

Each source module for Example 1 resides on a separate boot page. Each module performs a dummy task and then boots in the next page. Note in the **.map** file (**ex1.map**) that the data memory variables declared in each module overlap. This occurs because each boot page is considered to be a separate run-time context by the processor system.

In the C modules, the construct

**(short *)Sys_Ctrl_reg = BOOTx*

is used to boot in page x.

Example 2	System file	**ex2.sys**
	Source module	**ex2_1.dsp**
	Batch file	**makeex2.bat**
	C source module	**ex2_1.c**
	C batch file	**makeex2c.bat**

The source module for Example 2 declares and initializes two buffers stored in RAM and then performs a dummy task. The PROM Splitter creates *two* boot pages (0 and 1). The **loader** code is placed on boot page 0 and the source code on boot page 1. In the simulator you will see the **loader** code booted in first. This code initializes the proper memory locations and then boots in the source module.

Example 3	System file	**ex3.sys**
	Source module	**ex3_1.dsp, ex3_2.dsp**
	Batch file	**makeex3.bat**
	C source module	**ex3_1.c, ex3_2.c**
	C batch file	**makeex3c.bat**

One source module in Example 3 is defined to reside in *internal* PM RAM, the other in *external* PM RAM. When viewing the C example, the code is loaded starting with internal memory, until it is filled, and then will begin placing code in external memory, if needed. It is not possible to explicitly place modules in internal or external memory in C. If this feature is desired, the C-compiler-generated assembly modules must be hand modified for explicit placement.

Example 4	System file	**ex4.sys**
	Source module	**ex4_1.dsp, ex4_2.dsp**
	Batch file	**makeex4.bat**
	C source module	**ex4_1.c, ex4_2.c**
	C batch file	**makeex4c.bat**

The program ROM module (**ex4_2.dsp**) is declared to reside in both program ROM *and* on the boot page. Note that in contrast to Example 1, there is no data memory overlap, since both modules are part of the same run-time context.

12.5.10 Appendix: Example System 1

ex1.sys

```
.SYSTEM  example1;
.ADSP2105;
.MMAP0;
.seg/rom/boot=0                 boot_page_0[1024];      {2105 1K boot size}
.seg/rom/boot=1                 boot_page_1[1024];      {2105 1K boot size}
.seg/rom/boot=2                 boot_page_2[1024];      {2105 1K boot size}
.seg/rom/boot=3                 boot_page_3[1024];      {2105 1K boot size}
.seg/rom/boot=4                 boot_page_4[1024];      {2105 1K boot size}
.seg/rom/boot=5                 boot_page_5[1024];      {2105 1K boot size}
.seg/rom/boot=6                 boot_page_6[1024];      {2105 1K boot size}
.seg/rom/boot=7                 boot_page_7[1024];      {2105 1K boot size}
.seg/PM/ram/abs=0/code/data     int_pm[1024];           {2105 1K int pm}
.seg/DM/ram/abs=h#3800/data     int_dm[512];            {2105 int dm}
.endsys;
```

ex1_1.dsp

```
.module/ram/boot=0        ex1_module_1;
#include <def2105.h>

{this module is loaded into boot memory page 0}

.var/dm/ram     var_mod1_1[100];      {module 1 dm variable 1}
.var/dm/ram     var_mod1_2[100];      {module 1 dm variable 2}

{code section of module 1: this code boots module 2}

ex1_pg0:                ax0=1;
                        ax0=2;
                        ax0=3;
                        ax0=4;
                        ax0=5;
                        ax0=6;
                        ax0=7;
                        ax0=8;
                        ax0=9;
                        ax0=10;
                        ax0=0x240;
                        dm(Sys_Crtl_Reg)=ax0;      {boot page 1}
```

(example system continues on next page)

12 Hardware Interfacing

makeex1.bat

```
bld21 ex1
asm21 -cp ex1_1
asm21 -cp ex1_2
asm21 -cp ex1_3
asm21 -cp ex1_4
asm21 -cp ex1_5
asm21 -cp ex1_6
asm21 -cp ex1_7
asm21 -cp ex1_8
ld21 ex1_1 ex1_2 ex1_3 ex1_4 ex1_5 ex1_6 ex1_7 ex1_8 -a ex1 -e ex1 -g -x
spl21 ex1 ex1 -bs 1024
```