Engineer To Engineer Note EE-39

Notes on using Analog Devices' DSP, audio, & video components from the Computer Products Division
Phone: (800) ANALOG-D or (781) 461-3881, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com

Interfacing Byte Programmed Flash Memories to the ADSP-218x

Last Modified: 5/22/98

Introduction:

Until recently, the most flexible external boot memory was an EPROM. However, if you needed to erase or update the data on an EPROM, you had to remove it from the system, expose it to ultraviolet light, place it in an EPROM burner, and then insert it back into the design. A better solution is to update code/data while the non-volatile memory is still in the system. This is particularly true in modem designs, where communications protocols are continuously updated, and in systems such as a digital answering machine, where you must save compressed voice/data in a simple and efficient fashion. With the advent of FLASH memories, it is possible to save data permanently and update it when necessary without removing the IC. FLASH memories are also an asset in systems that need to save data during a power outage or brownout. The DSP can store its code/data contents from volatile internal memory to an external, non-volatile memory, and on revival of the system, rewrite the old information back to the DSP.

This Engineering Note demonstrates how to interface a 5v FLASH memory to an ADSP-218x DSP. The supplied code shows you how to program, erase, and retrieve device ID information from several AMD FLASH devices.

BDMA Wiring:

The BDMA port allows for "glueless" connections between the DSP and a FLASH. Table 1 shows the wiring between the DSP and a FLASH.

DSP	FLASH
A0-A13	A0-A13
D16-D23	A14-A21
D8-D15	D0-D7
BMS	CE
WR	WE
RD	OE

Figure 1. DSP -> FLASH Wiring

During read operations, the DSP sees the FLASH as an EPROM. The only additional signal necessary for FLASH systems is the WR signal that is used when changing the contents of the FLASH..

Basic AMD FLASH Operation:

The AMD FLASH memory space is parsed into a series of sectors. You can set each sector so it is readable, but not erasable. These boundaries are invisible; it is possible to read/write seamlessly across sector boundaries. The sector protection feature is particularly helpful in systems where the DSP is booting from and writing to a FLASH. It can protect boot sectors and leave the remaining portions of memory unprotected.

The AMD FLASH uses a byte-by-byte programming protocol. A series of command words are written to the FLASH (using BDMA writes in this example), essentially "unlocking" the device so the proper information can be written or obtained. The following operations, called Embedded Programming Algorithms, can be performed on an AMD FLASH:

Autoselect:
This operation identifies the manufacturer, model number, and the protection status of any sector.

Byte Write:
Use this operation programs a single byte of data into an unprotected FLASH sector.

Sector Erase:

This operation erases a sector of unprotected memory.

Chip Erase:
This operation erases all unprotected sectors on a FLASH.

The particular FLASH determines the exact series of signals that must be sent from the DSP to perform each embedded programming algorithm. Please refer to the appropriate AMD Data Sheet for more memory programming information.

Flash Programming via the BDMA Port:

The ADSP-218x has an external memory port (BDMA port) that you can use for off-chip reads and writes of a byte-wide device. The data transfers are initiated by setting four system control registers: the BDMA Internal Address Register (**BDMA_BIAD**), the BDMA External Address Register (**BDMA_BEAD**), the BDMA Control Register (**BDMA_BDMA_CTRL**) and the BDMA Word Count Register (**BDMA_BWCOUNT**). These registers are explained below (for more information, please refer to Chapter 11 in the *ADSP-2100 Family User's Manual*):

BDMA_BIAD:

On data reads, this register contains the address where data is saved in internal memory. On data writes, it contains the address of the first byte that is transferred into the FLASH.

BDMA_BEAD:

On data reads, this register contains the lower fourteen bits of the address where data is read from and transferred into internal memory. On data writes, this register contains the lower fourteen bits of the external memories' starting transfer address.

BDMA_BDMA_CTRL:

This register contains four pieces of information. It contains the upper eight bits of the external address where data is stored to or read from. This register also contains a bit that determines if the DSP automatically reboots once a BDMA transfer is complete. There is a two bit data field that determines the configuration of byte memory storage (8 LSB, 8 MSB, 16 (LSB then MSB) or 24 bit packets). Lastly, there is a one bit switch that determines if the BDMA transfer is an external read or write.

BDMA_BWCOUNT:

This register sets the number of BDMA transfers that are completed. This number refers to the number of BDMA words that are sent, not the number of bytes that are transferred from the external memory (for example, to transfer 16 data memory words to external memory, set **BDMA_BWCOUNT** to 16, not 32). The BDMA transfers are initiated immediately after this register is set and a BDMA interrupt is generated once **BDMA_BWCOUNT** is equal to zero. It is also possible to poll the **BDMA_BWCOUNT** register to see if the BDMA transfer was completed (this is useful if the processor cannot service a BDMA at that time).

BDMA Wait State Generation:

You set the number of wait states for BDMA transfers in the memory-mapped control register **PF_CSC** (DM(0x3FE6)), at bit locations 12, 13 and 14. When the DSP is rebooted, the number of wait states is automatically set to seven. When setting the wait state register, you must consider the rise/fall times for every I/O line. Generosity in calculating wait states will help guarantee that your system works under greater bus loads.

Flash Server Software:

Attached to this paper are a number of software modules used for accessing and controlling the AMD29LV010, AMD29LV020, AMD20LV040, and AMD29F040 FLASH memories. This software offers five *.ENTRY* points in the server code that are function names for FLASH operations. These functions are listed and described below:

bdma_setup:
This function sets the number of wait states and sets the appropriate interrupts.

prog_byte:
This function unlocks and writes the value of **d_byte** into the FLASH.

read_byte:
This function reads back the appropriate byte from the FLASH.

sect_erase:
This function erases one sector of the FLASH.

auto_inc:

This function increments the external address to the FLASH.

More information on each of these functions, including the parameters that are passed into each subroutine is included in the comments of the supplied code.

To access the FLASH server code, it must first be assembled and linked into your existing software. In order to make the server code "visible" to pre-existing code in other modules, the subroutines that will be used must be declared as external functions. After the module *flash_srv* (listing 1), a small example program is given (listings 2-4), showing how to implement the FLASH programming software. For more information on linking and calling functions located in separate modules, please refer to the *ADSP-2100 Family Assembler Tools and Simulator Manual*.

FLASH System Caveats:

The BDMA Write Anomaly:

On some revisions of the ADSP-218x, there is anomalous behavior on the data lines during BDMA writes. This exception is characterized as a two volt negative glitch (width 3ns) on each of the data lines. This glitch appears at the beginning of each wait state and is independent of the input voltage. Therefore, the outputs are briefly driven to 3v on a 5v DSP and 1.3v on a 3.3v DSP.

You can counter the anomaly on a 5v DSP with a simple RC network (see Figure 1) attached to each of the data pins and to ground (R=120Ω, C=47pF) that filters out the unwanted high frequency information while maintaining acceptable edge characteristics. On some systems, the RC circuit is not necessary to ensure that the transfers are completed properly (the bus capacitance is enough to hold the data lines high). For a complete list of the revisions of the DSP with this exception, please see the appropriate *Anomalies and Workarounds* document, available from Analog Devices.

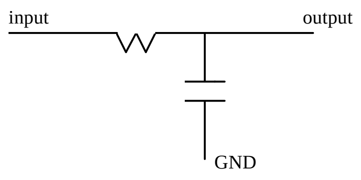

Figure 1. RC circuit configuration

Byte Programming:

You cannot correctly program a byte of data unless the contents of that register first equal 0xFF. Essentially, a byte program only converts a logic '1' to a logic '0'. A sector or chip erase operation is the only operation that converts a logic '0' to a logic '1' .

Additional Information:

For more information about interfacing ADSP-218x DSPs to various FLASHes, please consult the following sources:

ADSP-2100 Family User's Manual
ADSP-2100 Family Assembler Tools and Simulator Manual
http://www.analog.com
http://www.amd.com

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

 ANALOG DEVICES
 EUROPEAN DSP APPLICATIONS

 versatile FLASH utility for embedded systems

 History:
   created     15-SEP-97 by hs
   last change 01-DEC-98 by hs

 *****************************************************************************
}
.module/ram flash_srv;

{ Follwing four constants are the requirements for the FLASH memory and are
  preset for AMD 8bit wide flash memories requiring either 16bit or 12bit
  address unlock sequences. Possible types would be i.e. AM29LV001, AM29LV002,
  AM29F004, AM29LV004.

  To adapt the given FLASH server to a different FLASH EPROM vendor, please
  check whether the programming sequence can be used and change the following
  four constants accordingly.

  The word ulock1_a decribes the 14bit address range for the BEAD register,
  while ulock1_c must be put into BDMAC, containing the upper 8 page bits
  shifted 8bits left and ORed with 0x7 command bits to perform the BDMA write.
  i.e.: the first bus cycle is address 0x5555 and 0xAA as data
}
.const ulock1_a=    0x1555;             { lower 14bit part of address }
.const ulock1_c=    0x0107;             { upper  8bit part of address }
.const ulock1_b=    0x00AA;             { byte used in unlock }

{ The word ulock2_a decribes the 14bit address range for the BEAD register,
  while ulock2_c must be put into BDMAC, containing the upper 8 page bits
  shifted 8bits left and ORed with 0x7 command bits to perform the BDMA write.
  i.e.: the second bus cycle is address 0x2AAA and 0x55 as data
}
.const ulock2_a=    0x2AAA;             { lower 14bit part of address }
.const ulock2_c=    0x0007;             { upper  8bit part of address }
.const ulock2_b=    0x0055;             { byte used in unlock }

{ ADSP-218x specific registers, please do not change
}
.const  BDMA_BIAD=              0x3fe1;
.const  BDMA_BEAD=              0x3fe2;
.const  BDMA_BDMA_Ctrl=         0x3fe3;
.const  BDMA_BWCOUNT=           0x3fe4;
.const  PFTYPE=                 0x3fe6;

.var/dm/ram  d_byte;                    { holds value read / written }
.global      d_byte;
.var/dm/ram  d_btmp;                    { temporaray value read / written }

.var/dm/ram  c_byte;                    { flash command }
.global      c_byte;

.var/dm/ram  addr_hi;                   { upper 6bit of 22bit address }
.global      addr_hi;
.var/dm/ram  addr_ht;                   { upper 8bit of BDMA address }
```

```
.var/dm/ram  addr_lo;                   { lower 16bit of 22bit address }
.global      addr_lo;
.var/dm/ram  addr_lt;                   { lower 14bit of BDMA address }

.entry       bdma_setup;                { sets proper wait states }
.entry       prog_byte;                 { unlocks FLASH and writes byte }
.entry       read_byte;                 { reads selected byte from FLASH }
.entry       sect_erase;                { erases sector of FLASH device }
.entry       mem_ident;                 { read FLASH vendor identity }
.entry       auto_inc;                  { for burst transfers increments
                                          address }

{******************************************************************************
 *
 *  Global setup for the BDMA port
 *
 *  REGISTER USAGE SUMMARY:
 *
 *  modify : PFTYPE, IMASK
 *  output : none
 *  destroy: ar, ax0, ay0
 *  calls  : none
 *
 ******************************************************************************}
bdma_setup:
      ax0 = dm(PFTYPE);
      ay0 = 0x7000;                     { 7 wait states for BDMA access}
      ar = ax0 or ay0;
      dm(PFTYPE) = ar;

      ar = imask;
      ar = setbit 3 of ar;              { IRQ driven writes and reads }
      imask = ar;
      ena ints;

      rts;

{******************************************************************************
 *
 *  Flash application server
 *
 *    byte program:
 *
 *     c_byte : 0xA0
 *     d_byte : value
 *     addr_lo: low 16bit
 *     addr_hi: high 6bit
 *
 *  REGISTER USAGE SUMMARY:
 *
 *  modify : addr_lt, addr_ht, d_btmp
 *  destroy: ar, ax0, ay0, af
 *  calls  : init_seq, cmd_write
 *
 ******************************************************************************}
prog_byte:
      call init_seq;                    { unlock sequence }
      call cmd_write;                   { command word written }

      ar = dm(d_byte);                  { fetch byte }
      dm(d_btmp) = ar;
      call calc_adr;                    { compute registers from address }
      call DQ7_poll;                    { check for internal completion }

      rts;

{******************************************************************************
 *
 *  Flash application server
 *
 *    byte readback:
 *
 *     c_byte : don't care
 *     d_byte : readback value
 *     addr_lo: low 16bit
 *     addr_hi: high 6bit
 *
 *  REGISTER USAGE SUMMARY:
 *
 *  modify : addr_lt, addr_ht, d_btmp, d_byte
 *  output : see above
 *  destroy: ar, i7
 *  calls  : none
 *
 ******************************************************************************}
read_byte:
      ar = ^d_byte;
      dm(BDMA_BIAD) = ar;
      i7 = dm(addr_lo);                 { set the lower 14 bit }
      dm(BDMA_BEAD) = i7;
      dm(addr_lt) = i7;                 { store low address for readback }
      sr1 = dm(addr_hi);
      sr = lshift sr1 by 10 (hi);
      ar = dm(addr_lo);
      sr = sr or lshift ar by 10 (lo);
      ay0 = 0xff00;
      ar = sr1 and ay0;                 { holds now the upper 8 page bits }
      ay0 = 0x3;
      ar = ar or ay0;
      dm(BDMA_BDMA_Ctrl) = ar;
      ar = 0x1;
      dm(BDMA_BWCOUNT) = ar;            { start BDMA sequence }

      idle;                             { wait for BDMA access completed }
      rts;

{******************************************************************************
 *
 *  Flash application server
 *
 *    sector erase:
 *
 *     c_byte : 0x80
 *     d_byte : don't care
 *     addr_lo: low 16bit
 *     addr_hi: high 6bit
 *
 *
 *  REGISTER USAGE SUMMARY:
 *
 *  modify : addr_lt, addr_ht, d_btmp
 *  destroy: ax0, ay0, ar, af, sr0, sr1
 *  calls  : none
 *
 ******************************************************************************}
```

```
sect_erase:
      call init_seq;                  { unlock sequence }
      call cmd_write;                 { write command }
      call init_seq;                  { unlock sequence }

      ar = 0x30;
      dm(d_btmp) = ar;                { sector erase command 0x30 }
      call calc_adr;

      call DQ7_poll;                  { check for internal completion }

      rts;


{******************************************************************************
 *
 *  Flash application server
 *
 *    read back FLASH identity:
 *
 *     c_byte : 0x90
 *     d_byte : readback value
 *     addr_lo: low 16bit
 *     addr_hi: high 6bit
 *
 *  REGISTER USAGE SUMMARY:
 *
 *  modify : addr_lt, addr_ht, d_btmp
 *  destroy: ar, ax0, ay0, af
 *  calls  : init_seq, cmd_write, read_byte
 *
 ******************************************************************************}
mem_ident:
      call init_seq;                  { unlock sequence }
      call cmd_write;                 { command word written }
      call read_byte;                 { get the information }

      ar = 0xf0;                      { reset FLASH }
      dm(c_byte) = ar;
      call cmd_write;

      rts;


{******************************************************************************
 *
 *  Flash application server subroutine
 *  start FLASH embedded algorithms
 *
 ******************************************************************************}
init_seq:
      ar = ulock1_b;                  { start sequence, first command }
      dm(d_btmp) = ar;
      ar = ^d_btmp;
      dm(BDMA_BIAD) = ar;
      ar = ulock1_a;                  { places 1st unlock address }
      dm(BDMA_BEAD) = ar;
      ar = ulock1_c;                  { places 1st unlock command }
      dm(BDMA_BDMA_Ctrl) = ar;
      ar = 0x1;
      dm(BDMA_BWCOUNT) = ar;          { start BDMA sequence }

      idle;                           { wait for BDMA access completed }
      ar = ulock2_b;                  { second command }
      dm(d_btmp) = ar;
      ar = ^d_btmp;
      dm(BDMA_BIAD) = ar;
      ar = ulock2_a;                  { places 2nd unlock addr }
      dm(BDMA_BEAD) = ar;
      ar = ulock2_c;                  { places 2nd unlock cammand }
      dm(BDMA_BDMA_Ctrl) = ar;
      ar = 0x1;
      dm(BDMA_BWCOUNT) = ar;          { start BDMA sequence }

      idle;                           { wait for BDMA access completed }
      rts;

{******************************************************************************
 *
 *  Flash application server subroutine
 *  do the command word write
 *
 ******************************************************************************}
cmd_write:
      ar = dm(c_byte);                { third command }
      dm(d_btmp) = ar;                { now get the command byte }
      ar = ^d_btmp;
      dm(BDMA_BIAD) = ar;
      ar = ulock1_a;                  { places 1st unlock addr }
      dm(BDMA_BEAD) = ar;
      ar = ulock1_c;                  { places 1st unlock command }
      dm(BDMA_BDMA_Ctrl) = ar;
      ar = 0x1;
      dm(BDMA_BWCOUNT) = ar;          { start BDMA sequence }

      idle;                           { wait for BDMA access completed }
      rts;

{******************************************************************************
 *
 *  Flash application server subroutine split the 22bit address
 *  and write byte
 *
 ******************************************************************************}
calc_adr:
      ar = ^d_btmp;
      dm(BDMA_BIAD) = ar;
      i7 = dm(addr_lo);               { set the lower 14 bit }
      dm(BDMA_BEAD) = i7;
      dm(addr_lt) = i7;               { store low address for readback }
      sr1 = dm(addr_hi);
      sr = lshift sr1 by 10 (hi);
      ar = dm(addr_lo);
      sr = sr or lshift ar by 10 (lo);
      ay0 = 0xff00;
      ar = sr1 and ay0;               { holds now the upper 8 page bits }
      ay0 = 0x7;
      ar = ar or ay0;
      dm(BDMA_BDMA_Ctrl) = ar;
      ar = clrbit 2 of ar;
      dm(addr_ht) = ar;               { store high address & control for readback }
      ar = 0x1;
      dm(BDMA_BWCOUNT) = ar;          { start BDMA sequence }

      idle;                           { wait for BDMA access completed }
      rts;


{*******************************************************************************
 *
 *  Flash application server subroutine
 *  detect end of sequence
 *
 *******************************************************************************}
DQ7_poll:
      ar = ^d_btmp;                     { first readback }
      dm(BDMA_BIAD) = ar;
      ar = dm(addr_lt);
      dm(BDMA_BEAD) = ar;               { write again lower 14 bit }
      ar = dm(addr_ht);
      dm(BDMA_BDMA_Ctrl) = ar;          { write higher 8 bit & control }
      ar = 0x1;
      dm(BDMA_BWCOUNT) = ar;            { start BDMA sequence }

DQ_1: idle;                             { wait for BDMA access completed }

      ay0 = dm(d_btmp);                 { save first readback data }

      ar = ^d_btmp;                     { second readback }
      dm(BDMA_BIAD) = ar;
      ar = dm(addr_lt);
      dm(BDMA_BEAD) = ar;               { write again lower 14 bit }
      ar = dm(addr_ht);
      dm(BDMA_BDMA_Ctrl) = ar;          { write higher 8 bit & control }
      ar = 0x1;
      dm(BDMA_BWCOUNT) = ar;            { start BDMA sequence }

DQ_2: idle;                             { wait for BDMA access completed }

      ax0 = dm(d_btmp);                 { save second readback data }
      ar = ax0 - ay0;                   { two readbacks hold same value ? }
      if eq jump DQ7_exit;              { if so, operation successful }
      ar = tstbit 5 of ax0;             { timeout ? }
      if eq jump DQ7_poll;              { no, re-read status }

      ar = ^d_btmp;                     { first control readback }
      dm(BDMA_BIAD) = ar;
      ar = dm(addr_lt);
      dm(BDMA_BEAD) = ar;               { write again lower 14 bit }
      ar = dm(addr_ht);
      dm(BDMA_BDMA_Ctrl) = ar;          { write higher 8 bit & control }
      ar = 0x1;
      dm(BDMA_BWCOUNT) = ar;            { start BDMA sequence }

DQ_3: idle;                             { wait for BDMA access completed }

      ay0 = dm(d_btmp);                 { save first readback data }

      ar = ^d_btmp;                     { second control readback }
      dm(BDMA_BIAD) = ar;
      ar = dm(addr_lt);
      dm(BDMA_BEAD) = ar;               { write again lower 14 bit }
      ar = dm(addr_ht);
      dm(BDMA_BDMA_Ctrl) = ar;          { write higher 8 bit & control }
      ar = 0x1;
      dm(BDMA_BWCOUNT) = ar;            { start BDMA sequence }
```

```
DQ_4:  idle;                          { wait for BDMA access completed }

       ax0 = dm(d_btmp);              { save second readback data }
       ar = ax0 - ay0;                { two readbacks hold same value ? }
       if eq jump DQ7_exit;           { if so, operation successful }
       ar = tstbit 5 of ax0;          { timeout ? }
       if eq jump DQ7_poll;           { no, re-read status }

       ar = 0xf0;                     { reset FLASH }
       dm(c_byte) = ar;
       call cmd_write;
       jump fs_error;                 { flag error occurred }

DQ7_exit:
       rts;


{******************************************************************************
 *
 *  Flash application server subroutine
 *  error treatment
 *
 ******************************************************************************}
fs_error:
       ar = 0xff;
       dm(c_byte) = ar;
       rts;


{******************************************************************************
 *
 *  autoincrement external address;
 *
 *  REGISTER USAGE SUMMARY:
 *
 *  input  : none
 *  modify : addr_lo, addr_hi
 *  output : none
 *  destroy: ar
 *  calls  : none
 *
 ******************************************************************************}
auto_inc:
       ar = dm(addr_lo);              { increment lower address }
       ar = ar + 1;
       dm(addr_lo) = ar;
       if not AC rts;

       ar = dm(addr_hi);              { increment high address if carry set }
       ar = ar + 1;
       dm(addr_hi) = ar;
       rts;

.endmod;
```

listing 1. FLASH.DSP

```
{******************************************************************************
 ANALOG DEVICES
 EUROPEAN DSP APPLICATIONS
 example calls for the flash server
 ******************************************************************************
}
.module/ram/abs=0 INTVEC;
.include    <flash.h>;

.var/dm/ram  count;        /* temporary value */

ISR_RESET:          jump start;  NOP; NOP; NOP;
ISR_IRQ2:           RTI;         NOP; NOP; NOP;
ISR_IRQ1:           RTI;         NOP; NOP; NOP;
ISR_IRQ0:           RTI;         NOP; NOP; NOP;
ISR_S0_TRANSMIT:    RTI;         NOP; NOP; NOP;
ISR_S0_RECEIVE:     RTI;         NOP; NOP; NOP;
ISR_IRQE:           RTI;         NOP; NOP; NOP;
ISR_BDMA:           RTI;         NOP; NOP; NOP;
ISR_S1_TRANSMIT:    RTI;         NOP; NOP; NOP;
ISR_S1_RECEIVE:     RTI;         NOP; NOP; NOP;
ISR_TIMER:          RTI;         NOP; NOP; NOP;
ISR_POWERDOWN:      RTI;         NOP; NOP; NOP;

start:
      call bdma_setup;    /* set up the waitstates.. */

      ar = 0x0;
      dm(addr_lo) = ar;   /* set low address */
      ar = 0x0;
      dm(addr_hi) = ar;   /* set high address */
      ar = 0x80;
      dm(c_byte) = ar;    /* set flash command to erase */
      call sect_erase;    /* call the API */

      nop;                /* for debug purposes */

      ar = 0x0;
      dm(count) = ar;     /* set count to zero */
      dm(d_byte) = ar;    /* start off with 0x00 */
      ar = 0x0;
      dm(addr_lo) = ar;   /* at low address 0x0 */
      ar = 0x0;
      dm(addr_hi) = ar;   /* and high address 0x0 */
      ar = 0xa0;
      dm(c_byte) = ar;    /* requiring 0xa0 as write command */

rest: call prog_byte;     /* call the API */
      call auto_inc;      /* go to next address */
      ar = dm(count);
      ar = ar + 1;        /* increase count */
      dm(count) = ar;
      dm(d_byte) = ar;    /* and write count out to flash */

      ay0 = 0x100;        /* perform this 256 times */
      ar = ar - ay0;
      if eq jump wait;

      jump rest;

wait: jump wait;          /* set breakpoint here */

      idle;

.endmod;
```

listing 2. MAIN.DSP

```
{***************************************************************************
 Definitions for ADSP-2186 Host Processor,                    Ver: 0.90.2b
      created     01.Apr.95 by Analog Devices
      last change 30.Oct.96 by Stefan Hacker
 ***************************************************************************
}
.const  IDMA=                  0x3fe0;
.const  BDMA_BIAD=             0x3fe1;
.const  BDMA_BEAD=             0x3fe2;
.const  BDMA_BDMA_Ctrl=        0x3fe3;
.const  BDMA_BWCOUNT=          0x3fe4;
.const  PFDATA=                0x3fe5;
.const  PFTYPE=                0x3fe6;

.const  SPORT1_Autobuf=        0x3fef;
.const  SPORT1_RFSDIV=         0x3ff0;
.const  SPORT1_SCLKDIV=        0x3ff1;
.const  SPORT1_Control_Reg=    0x3ff2;
.const  SPORT0_Autobuf=        0x3ff3;
.const  SPORT0_RFSDIV=         0x3ff4;
.const  SPORT0_SCLKDIV=        0x3ff5;
.const  SPORT0_Control_Reg=    0x3ff6;
.const  SPORT0_TX_Channels0=   0x3ff7;
.const  SPORT0_TX_Channels1=   0x3ff8;
.const  SPORT0_RX_Channels0=   0x3ff9;
.const  SPORT0_RX_Channels1=   0x3ffa;
.const  TSCALE=                0x3ffb;
.const  TCOUNT=                0x3ffc;
.const  TPERIOD=               0x3ffd;
.const  IO_Wait_Reg=           0x3ffe;
.const  System_Control_Reg=    0x3fff;

.external d_byte;
.external c_byte;
.external addr_hi;
.external addr_lo;
.external bdma_setup;
.external prog_byte;
.external sect_erase;
.external auto_inc;
```

listing 3. FLASH.H

```
{***************************************************************************
 batch file to create application code
**************************************************************************
}
asm21 flash -2181
asm21 main -2181
ld21 flash main -e prog_it -a 2186
```

listing 4. MAKE.BAT