

Using an ADSP-2181 DSP's IO Space to IDMA boot another ADSP-2181.

*Contributed by J. O.
Last Modified 10/28/97*

This application note describes an example hardware and software interface between the Internal DAM (IDMA) port of the ADSP-2181. Additionally, this application note shows how a host DSP can boot a target DSP using the IDMA port. As each specific system design has it's own requirements and challenges, this application note does not presume to provide the only possible solution. Rather, it is meant to provide the system designer a flexible framework of ideas that can be tailored to meet individual system requirement .

IDMA Operation

The host DSP can gain access of the target ADSP-2181's internal memory through the target's IDMA port. Any host processors accessing the target ADSP-2181 through IDMA can treat the target DSP as a memory mapped slave peripheral, and can have access to all of the target's internal Data Memory (DM) and Program memory (PM), excluding control registers. The adsp-2181's IDMA port consists of a 16 bit multiplexed address/data bus (IAD15:0), a select line (/IS), IDMA address latch (IAL), read (/IRD), write (/IWR), and acknowledge (/IACK) signals. The host DSP is responsible for initiating all data transfers. The target DSP memory address is loaded into the IDMA Address register (IDMAA). This register contains the 14 bit internal memory address, along with a bit to specify the type of transfer: 24 bit Program Memory opcodes, or 16 bit Data memory data. The IDMAA register can be initialized by either the target DSP or by the host DSP. The host can initialized this register by performing an address latch cycle. An address latch cycle is defined by the host asserting the IAL signal, and then transferring a 15 bit (14 address bits plus 1 destination memory type bit) value on the IAD pins. To streamline the transfer of large segments of opcodes or data, an Address Latch Cycle does not need to be performed for each IDMA access. Instead, once latched, the address

is automatically incremented after every IDMA word transfer. As the IDMA port has a 16 bit bus, 24 bit transfers require two host accesses. The first access transfers the most significant 16 bits, the second access transfers the least significant 8 bits, right justified, with a zero filled upper byte. IDMA address increments occur after the entire 24 bit word has been transferred for a PM transfer, or after the entire 16 bit data word has been transferred for a DM transfer

Booting the DSP

The IDMA Port on the DSP can be used to boot load the DSP on power up. This eliminates the need for a separate EPROM for the DSP. On the ADSP-2181, booting is controlled through the use of the MMAP and BMODE pins. Booting through the IDMA port is enabled by holding the MMAP pin low, and the BMODE pin high. With this signal combination, on RESET, the DSP does not activate its external address bus to access an EPROM. Instead, the DSP expects a host to begin IDMA transfers to fill its internal Data and Program memories. This process consists of the host performing standard IDMA instruction and data transfers. Booting is terminated when the DSP restart vector at DSP Address PM(0x0000) is written. An efficient boot loading sequence would consist of the host filling the DSP's internal Program Memory starting at location PM(0x0001), and using the automatic address increment feature on the IDMA port to speed the transfer of code block in ascending address order. The host can then initialize data memory. When all initialization is complete, the host should then initialize the DSP's restart vector and DSP program execution will commence.

Interface Hardware Design

The IDMA port of the target DSP is mapped into two locations in the host ADSP-2181's IO space. One location is used by the host to latch the address that an IDMA write would target. The other location is used when transferring data an instruction information. The target DSP has been configured for IDMA boot, the MMAP=0 and the BMODE=1.

Minimal logic is required to connect the external bus of the host ADSP-2181 to the target's IDMA port. The only external logic that is required is a singular or gate, which is used to ensure that IAL and /IWR are never asserted at the same time. This is achieved by using the A) and /WR lines from the host as inputs to an OR gate, whose output is then connected to /IWR on the target DSP. A0, from the host, is connected to the address latch signal, IAL, on the target. This connection allows the host to commence an address latch sequence by writing to an address that contains bit 0 in the host memory. The write strobe, /IOMS, is connected to /IS, so that any write to the IO space of the host will trigger the write strobe of the target DSP's IDMA port. The 16 data lines form the host's IO space (D8-D23) are directly connected to the IAD data bus on the target ADSP-2181. The host will use this bus to transmit the DSP memory addresses, as well as transfer data to the DSP.

Generating "Boot" Code

The ADSP-21xx family operates on 24-bit instruction opcodes. The IDMA port can only accept 16-bit values. To transfer instruction opcodes through the IDMA port, the most significant 16 bits transferred first, followed by the least significant 8 bits, right justified.

DSP Executable files are produced by the ADSP-21xx Family Linker. The Linker takes object files generated by the Assembler and C Compiler, places them within the memory architecture defined by the system architecture file, and generates a DSP executable (.EXE). A detailed description is available in Appendix B

of the ADSP-2100 Family Assembler Tools & Simulator Manual.

Since the executable file contains 24 bit DSP opcodes, the file needs to be re-formatted to adhere to the IDMA port's 16 bit data requirement. First, it is necessary to remove the place holders and formatting headers from the opcodes from the .exe file.

The ADSP-2100-Family .EXE format separates Program Memory and Data Memory segments on a module-by-module basis. The executable format has the following elements:

```
@PA          <---- Start of PM RAM
Module
0000         <---- Starting address
123456       <---- First Opcode
789abc       <---- Second Opcode
def012       <---- Third Opcode
:           :
#12345678    <---- End-of-module specifier
@DA          <---- Start of DM RAM
segment
0000         <---- Starting address
1234         <---- First data word
5679         <---- Second data word
:           :
:           :
#12345678    <---- End-of-module specifier
```

After the place holders have been removed, the 24 bit opcodes need to be reformatted into 2 16 bit words. The first the 16 MSBs and the second with the 8 LSBs with leading zeroes. Note, the usage of the PX register is greatly helpful for implementing this procedure, as seen in the attached code.

TOPICS FOR FURTHER DISCUSSION

Hardware Signaling

In many instances, it may be desirable for the host and DSP processors to have additional avenues of communication. The host can use one of its programmable flags as an output attached to a hardware interrupt on the DSP. With this method, the host can alert the DSP prior to a transfer, or inform the DSP that a transfer has been completed. This can be especially useful because there is no interrupt associated with IDMA operation on the ADSP-2181. The DSP can likewise use a programmable flag as an output to signal the host if there is new data for the host to use, or if new code is required for download.

Multiple DSP Processors

In this application note, we focused on connecting a single ADSP-2181 to a microprocessor. This scheme can be expanded to multiple DSP processors without too much trouble. In a multiple DSP system, all IDMA lines except IS and IACK can be bussed together. Multiple IS lines are needed to select each individual DSP processor, and multiple IACK lines are needed to monitor the activity on each individual DSP processor. Each DSP processor needs two micro controller memory space addresses assigned to it, and that address assignment used to assert the appropriate IS signal. Each DSP processor can then be accessed individually.

References

- ADSP-2100 Family User's Manual, Third Edition (9/95), Analog Devices, Inc.
- ADSP-2100 Family Assembler Tools & Simulator Manual, Analog Devices, Inc.
- ADSP-2100 Family EZ-Kit Lite Reference Manual, Analog Devices, Inc.

```

/*****
/*This program transfer data from a master ADSP-2181 to a slave ADSP-2181
*/
/*through the IDMA port
*/
/*****/
.MODULE/abs=0x0000          idma_transfer;
.CONST                    length =68;          {decimal length of vector}
.VAR/PM/RAM              x_input[length];
.INIT                    x_input: <xin.dat>;

                        JUMP start; NOP; NOP; NOP;          {Interupt vector table}
                        RTI; NOP; NOP; NOP;                {No interrupts used}
                        RTI; NOP; NOP; NOP;
                        RTI; NOP; NOP; NOP;
                        RTI; NOP; NOP; NOP;
                        RTI; NOP; NOP; NOP;
```

```
RTI; NOP; NOP; NOP;
RTI; NOP; NOP; NOP;
RTI; NOP; NOP; NOP;
RTI; NOP; NOP; NOP;
RTI; NOP; NOP; NOP;
```

start:

```
AX0=0x0000;
DM(0x3fff)=AX0;           {Sys Control reg, PM wait states off}
DM(0x3fe6)=AX0;          {PF control reg, all PF inputs}
DM(0x3FFE)=AX0;          {# of wait states for address 0-1ff}
L6=0;                     {length of the data buffer}
I6=^x_input;             {starting point of data to be transferred}
M6=1;                     {increment amount}
MODIFY(i6,m6);           {pointer init to 2nd value of vector}
```

/*****address latch to 0x0001 *****/

```
CALL check_ack;          {check to see if target is ready to receive}
AX0=0x0001;              {starting address to write at in the slave}
IO(0x0001)=AX0;          {writing to this address will toggle both the}
                          {/IS and IAL lines on the slave, additionally,}
                          {this will send the address to be latched}
```

/*****write all data, starting at address 0x0001*****/

```
CNTR=length-1;          {counter set to #words to be transmitted}
DO loop1 UNTIL CE;
CALL check_ack;          {check to see if target is ready to receive}
CALL format;             {format data for transfer}
IO(0x0100)=AY0;          {/TWR and IS asserted, the first 16 msb of}
                          {data are put onto IAD 0-15}
CALL check_ack;          {check to see if target is ready to receive}
IO(0x0100)=AY1;          {/TWR and IS asserted, 8 lsb of data, right}
                          {justified with leading zeros are put}
                          {onto IAD 0-15}
```

loop1: NOP;

/*****latch to address 0x0000*****/

```
CALL check_ack;          {check to see if target is ready to receive}
AX0=0x0000;              {starting address to write at in the slave}
IO(0x0001)=AX0;          {writing to this address will toggle both the}
                          {/IS and IAL lines on the slave, additionally,}
                          {this will send the address to be latched}
```

/*****write to address 0x0000*****/

```
I6=^x_input;            {point to beginning of data buffer}
CALL check_ack;          {check to see if target is ready to receive}
```

```
CALL format;                                {format data for transfer}
IO(0x0100)=AY0;                              {/IWR and IS asserted, the first 16 msb of
                                             {data are put onto IAD 0-15}

CALL check_ack;                             {check to see if target is ready to receive}
IO(0x0100)=AY1;                              {/IWR and IS asserted, 8 lsb of data, right}
                                             {justified with leading zeros are put}
                                             {onto IAD 0-15}

idle;

/*****
/*The "format" subroutine pre-formats the 24 bit wide program memory data into 16 bit wide data. The
*/ /*16 most significant bits are stored in AY0 and the 8 least significant bits are stored in AY1, right
*/
/*justified with leading zeroes.
*/
/*****/
format:      AY0=PM(I6,M6);                    {the 16 MSBs are stored in data register AY0}
             AY1=PX;                          {the 8 LSBs are stored in data register AY1}
             RTS;

/*****
/*The "check_ack" subroutine checks to see if the /IACK line is low
*/
/*****/
check_ack:
             AX0=dm(0x3fe5);                   {Poll PF1, which is connected to /IACK}
             AR=TSTBIT 1 of AX0;              {corresponds to PF1}
             IF NE JUMP check_ack;            {if not ready, check /IACK again}
             RTS;

.endmod;
```

SYSTEM BLOCK DIAGRAM

