



Introduction to SHARC® Multiprocessor Systems Using VisualDSP++™

Contributed by Maikel Kokaly-Bannourah

April 01, 2003

Introduction

The following Engineer-to-Engineer note is intended to give an introduction to Multiprocessor (MP) systems using VisualDSP++™. The explanation will be based on code example written for an MP system, which consists of two ADSP-21160s (2 x ADSP-21160 EZ-Kit Boards) using VisualDSP++™ 3.0.

SHARC® DSP Multiprocessor systems can be configured in different ways:

- Several DSPs sharing the external bus
- Link Port point-to-point communication
- Use of the DSP's serial ports in multi-channel mode.

This note will discuss the implementation of an MP system with the DSPs sharing the external bus. For more details on other implementations please refer to the ADSP-21160 SHARC® DSP Hardware Reference Manual.

Linker Description File (LDF) for MP Systems

The very first step in setting up an MP system is to create a multiprocessor project using the multiprocessing capabilities of the linker, and an LDF file to describe the system.

The LDF describes the multiprocessor memory offsets, shared memory, and each processor's memory. The following LDF commands must be considered when writing an MP LDF:

- **MPMEMORY{}**, it defines each processor's offset within multi-processor memory space (MMS). The linker uses the offsets during multiprocessor linking.
- **MEMORY{}**, it defines memory for all processors present in the system.
- **PROCESSOR{}** and **SECTIONS{}** commands define each processor and place program sections for each processor's output file, using the memory definitions.
- **SHARED MEMORY{}**, it is needed when external shared memory is used in the system. This command identifies the output for the shared memory items and generates Shared Memory executable files (.SM) that reside in the shared memory of the MP system.

The .SM file is generated from a source code file (.ASM, .C or .CPP), which must be included with the project files. This file contains the variable definitions for the data that will be placed in the external shared memory.

- **LINK_AGAINST()**, it resolves symbols within multiprocessor memory and directs the linker to check specified executables (.DXEs and .SMs) to resolve variables and labels that have not been resolved locally. Whenever expressions or variables are defined in the MMS (i.e. internal memory of another processor in the system) the LINK_AGAINST() command must be used in the LDF.

Note: if .SM files and DXE files are included in the command line, the .SM file must be placed

first, followed by all other DXE's, for the linker to be able to resolve the variables correctly.

The maximum number of processors that can be declared in one LDF is architecture-specific (i.e. maximum of 6 ADSP-21160's or 2 ADSP-21065L's). Also note that a combination of different DSPs with different architectures (i.e. ADSP-21062 and ADSP-21160) in the same LDF is not supported by VisualDSP++™. However, a combination of DSPs from the same architecture family (i.e. ADSP-2106x members such as ADSP-21060, ADSP-21061 and ADSP-21062) is supported although some memory

segments definitions considerations must be made.

An MP LDF example where all the above commands are used is shown in Figure 1. The remainder of the LDF file is basically the same as the default one provided with the tools (please refer to *the Linker and Utilities Manual for ADSP-21xxx Family of DSPs* or to *EE-69 "Understanding and Using Linker Description Files (LDFs)"* for a general description on LDF files). In the following example, a 2 ADSP-21160 and external shared memory system is defined.

```
MEMORY
{
  mem_rth { TYPE(PM RAM) START(0x00040000) END(0x000400ff) WIDTH(48) }
  mem_pmco { TYPE(PM RAM) START(0x00040100) END(0x000491ff) WIDTH(48) }
  [...]
}
MPMEMORY
{
  ID1 { START(0x00100000) } // Declare Multiprocessor Memory Space (MMS)
  ID2 { START(0x00200000) } // offset values for processors ID1 and ID2
}
SHARED_MEMORY //Declare external shared memory
{
  OUTPUT(shared.sm)
  SECTIONS
  {
    dxm_dmex
    { INPUT_SECTIONS( shared.doj(sram_da) )
    } >mem_dmex
  }
}
PROCESSOR ID1 // Processor ID1 sections definition
{
  LINK_AGAINST(shared.sm,ID2.dxe)
  OUTPUT(ID1.dxe)
  SECTIONS
  {
    dxm_rth
    { INPUT_SECTIONS( ID1.doj(seg_rth) $LIBRARIES(seg_rth))
    } >mem_rth

    [...]
  }
}
PROCESSOR ID2 // Processor ID2 sections definition
{
  LINK_AGAINST(shared.sm,ID1.dxe)
  OUTPUT(ID2.dxe)
  SECTIONS
  {
    dxm_rth
    { INPUT_SECTIONS( ID2.doj(seg_rth) $LIBRARIES(seg_rth))
    } >mem_rth

    [...]
  }
}
```

Figure 1 Excerpt from an MP LDF example

Now that the different sections of the LDF have been discussed, we can examine the example code that explores some of the MP capabilities of the DSP.

For MP system hardware configuration please refer to chapter 7 of ADSP-21160 SHARC® DSP Hardware Reference. Also for information on how to configure a cluster system using two ADSP-21160 evaluation boards refer to the ADSP-21160 EZ-KIT Lite User’s Guide.

Multiprocessor Memory Space (MMS)

The multiprocessor memory space is divided into a number of address regions (this number is processor specific) that correspond to the internal memory of the DSPs in an MP system. The ADSP-21160’s multiprocessor memory space appears in Figure 2.

Note: programs may only use Normal word addressing in multiprocessor memory space. Other addressing schemes may corrupt valid data.

Depending on the address range used, the internal memory of a particular DSP in the multiprocessor system will be accessed as a source or destination. Writes to the Broadcast region access the memory of all DSPs in the multiprocessing system.

For instance, accessing a memory location within the address range 0x300000 – 0x3FFFFFF, is equivalent to accessing the internal memory of the DSP in the MP system with ID 3.

A DSP can also use the MMS to access its own internal memory by accessing the corresponding memory region. Note that in this case, the DSP reads/writes from/to its own internal memory and does not make an access on the external system bus.

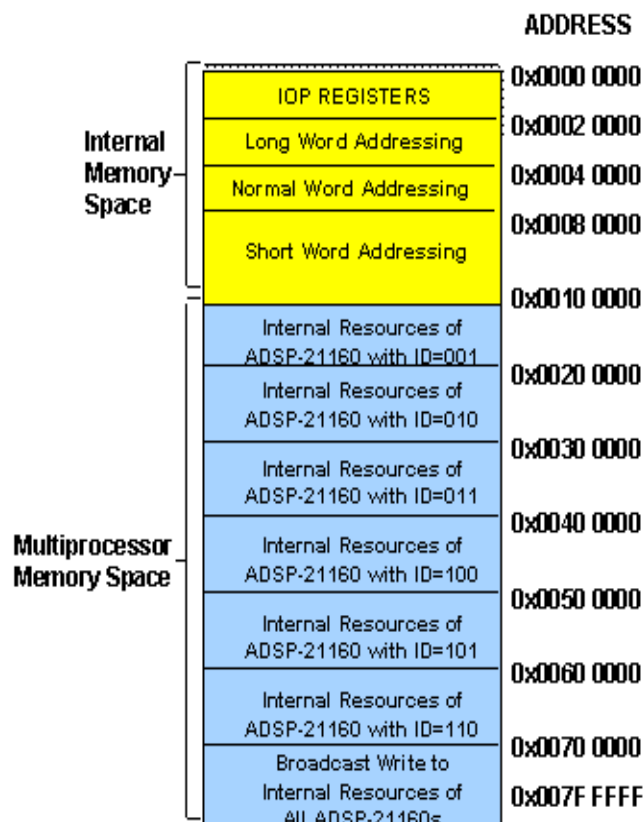


Figure 2 ADSP-21160 Multiprocessor Memory Space

The following is an example from the code where the MMS is used to access a memory location of another DSP in the system. In this case DSP with ID1 accesses the external port buffer 1 (EPB1) of DSP with ID2:

Example 1:

```
r0=0x200006;          dm(EI11)=r0;
```

In example 1, the MMS address for ID2 is 0x200000, which is then added to the address corresponding to the EPB1 (0x6). Therefore, this will result in a write access to ID2’s EPB1 by an external device (DSP or host).

Note: In DSP multiprocessor systems a DSP with ID=001 must be present, since this is the DSP responsible for driving the external bus control lines stable during reset.

External Memory

External memory is widely used in MP systems. An important point to keep in mind is that all DSPs in the system must initialize their own control registers before trying to access the external memory (i.e. WAIT register in case of SBSRAM).

The ADSP-21160 can be gluelessly interfaced to synchronous and asynchronous SRAM devices, however the use of DRAM requires an external controller.

It is very important to set up the proper access mode for the type of memory used in the hardware system. The access mode is programmed via the WAIT register. Default power up/reset settings for the System Control (SYSCON) and WAIT registers are detailed in the ADSP-21160 Hardware Reference Manual. User defined settings must support the external memory address ranges that the user intends to use in their code and hardware systems as well as the access mode appropriate to the memory device(s) in use (i.e. synchronous or asynchronous accesses). The MSIZE setting must also not exceed the size of the actual physical memory connected in the user's system.

Note that SDRAM is gluelessly supported by certain devices like the ADSP-21065L and the ADSP-21161. For these cases, specific registers must be initialized prior to accessing the external memory. The SDRDIV and IOCTL, for the ADSP-21065L, and the SDRDIV and SDCTL, for the ADSP-21161, registers of all processors in the system must be initialized to the same value. Once the DSP's internal memory controller has been configured, the external memory can be accessed by the DSP via the external bus.

In the example project, the shared.asm file contains the variable definitions for the data that will be placed in the external shared memory.

Note: the DSP with the lowest ID number (and therefore highest external bus arbitration

priority in the system) is responsible for initializing the external data defined in the .ASM shared memory file during the booting-up sequence.

Inter Processor Messages and Vector Interrupts

The Message Passing registers (MSGRx) are general-purpose memory mapped registers that can be used for message passing between the host and a DSP or between two DSPs. Similarly, Vector Interrupts are used for inter-processor communication between the host and a DSP or between DSPs.

The MSGRx and VIRPT registers can be used for message passing in the following ways:

- **Message Passing.** The host (or master DSP) can use any of the 8 message registers, MSGR0 through MSGR7, to communicate with the DSP.
- **Vector Interrupts.** The host (or master DSP) can issue a vector interrupt to the DSP by writing the address of an interrupt service routine to the VIRPT register. When serviced, this high priority interrupt causes the DSP to branch to the service routine at that address.

Example 2:

```
// Excerpt from ID2: VIRPT Generation
I0=0x100001; // VIRPT reg. address in ID1
R0=0x40080; // ISR at SFT0I address in ID1
will be executed

DM(I0,M0)=R0; // write to VIRPT reg.in ID1
[...]

// Excerpt from ID1: VIRPT Service Routine
// in SFT0I user software interrupt vector
// address

BIT SET IMASK VIRPTI; // VIRPT enabled [...]

// In vector interrupt table

R0=0x2f2f2f2f; // Value for msg. Passing RTI
(DB); // Serve VIRPTI generated by ID2
I0=MSGR0; // Load address of MSGR0 and write
value in ID2

DM(MMS_ID2,I0)=R0;

[...]
```

In example 2, ID2 triggers a vector interrupt in ID1 by writing the address of the service routine to be served in ID1 (0x40080 = SFTOI) to VIRPT (0x100001 = 0x1 VIRPT address + 0x100000 MMS ID1). Then, the service routine in ID1 writes a test value to the MSGR0 register of ID2, using a previously defined offset value (MMS_ID2 = 0x200000).

This is just an example of how inter-processor message passing and VIRPT interrupts can be used as flags or just to indicate program execution completion in MP systems.

Bus Lock and Semaphores

Semaphores are useful for synchronizing tasks performed in an MP system. A semaphore is a flag that can be accessed by any of the DSPs present in the system.

In critical tasks (i.e. should not be interrupted), when attempting a read-modify-write operation on a semaphore, the DSP must have bus mastership for the duration of the operation.

This can be achieved by using the DSP's bus lock feature, which retains mastership of the bus and prevents other processors from simultaneously accessing the semaphore.

A read-modify-write operation is accomplished with the following steps (Example 3):

1. Request bus lock by setting the BUSLK bit in MODE2.
2. Wait for bus mastership to be acquired.
3. Wait until Direct Write Pending (DWPD) is zero.
4. Read the semaphore, test it, and write to it.

The following is an Excerpt from ID1's code demonstrating the use of Bus Lock and Broadcast write:

Example 3:

```
// Excerpt ID1 code: BROADCAST write using
// Bus Lock
```

```
BIT SET MODE2 BUSLK;
if NOT BM jump(PC,0);
ustat1 = DM(SYSTAT);
BIT TST USTAT1 12;
if TF jump(PC,-2);
b1 = Broadcast_data;      l1 = N;  m1 = 1;
b8 = MMS_Broadcast;      l8 = N;  m8 = 1;
lcntr = 10;
do broadcast_transfer until lce;
broadcast_transfer: r2 = dm(i1,m1);
                    pm(i8,m8) = r2;

pm(i8,m8) = r2;
BIT CLR MODE2 BUSLK;
```

While the BUSLK bit is set, the DSP can determine if it has acquired bus mastership by executing a conditional instruction with the Not Bus Master (Not BM) condition code. If it has become the bus master, the DSP can proceed with the external read or write. If not, it can clear its BUSLK bit and try again later.

After bus mastership is acquired, the Direct Write Pending (DWPD) bit's status in SYSTAT must be checked to ensure that a semaphore write by another processor is not pending.

Bus lock can be used in combination with broadcast writes to implement reflective semaphores in a multiprocessing system. The reflective semaphore (i.e. located in the DSP's internal memory or an I/O processor register) must be located at the same address of each DSP. Once the DSP has become the bus master, it performs a broadcast write to the specified address on every DSP, including itself.

Lastly, the BUSLK bit must be cleared to free the bus after the broadcast transfer has finished.

Multiprocessor Data Transfers

Throughout the code, several types of data transfers have been implemented:

1. Master and Slave Direct Memory Access (DMA) between ID1 and ID2,

2. Master DMA from external memory,
3. Core transfer,
4. Broadcast Write to all DSPs in the system.

Let's now examine the different types of data transfers performed. Note that the Broadcast Write has already been discussed in the previous sections.

7.1. Master and Slave Direct Memory Access (DMA) between ID1 and ID2

Setup Master ID1

A channel in this mode can independently initiate internal or external memory transfers. Master mode applies to all external port DMA channels: 10, 11, 12, and 13.

In example 4, DMA channel 11 was used to perform a Master DMA transfer from ID1 to ID2 as follows:

Example 4:

```
// Excerpt from ID1:Master DMA,DMA channel
// 11, transfer from ID1 to ID2
r0=0; dm(DMAC11)=r0;// clear DMA Contr Reg
r0=DMA_source_ID1;dm(II11)=r0;// source ID1
r0=1; dm(IM11)=r0; // modifier = 1
r0=10; dm(C11)=r0; // counter = 10
r0=0x200006;
dm(EI11)=r0; // write to slave EPB1
r0=0; dm(EM11)=r0; // ext. modifier = 0
r0=10; dm(EC11)=r0; // ext. counter = 10
ustat1=0x0404;
dm(DMAC11)=ustat1; // transmit data
ustat1=dm(DMAC11);
bit set ustat1 0x1; // enable DMA channel
dm(DMAC11)=ustat1;
```

ID1 sets up the channel's parameter registers. The I/O processor uses the EI, EM, and EC registers to access MMS of ID2 in master mode DMA. ID1 will write to the FIFO (EPB1) in ID2. Note that the external modifier is set to zero.

To initiate a master mode DMA transfer ID1 sets the channel's DMA enable (DEN) bit. The DSP will then start transferring data to the EPB1 buffer FIFO, where the slave DSP, in this case ID2, can access it. ID2 needs to set up the Slave DMA before the data can be transferred to its own internal memory.

Setup Slave ID2

In a slave mode DMA channel, when the data transfer direction is external to internal, a slave mode DMA channel does not initiate any DMA transfers until the external device (in this case ID1) writes data to the channel's EPB1 buffer FIFO.

The Slave DMA transfer example looks as follows:

Example 5:

```
// Excerpt from ID2:Slave DMA, DMA channel
// 11, receive data transmitted by ID1
r0=0;
dm(DMAC11)=r0;// clear DMA Control Reg.
r0=DMA_dest_ID2;
dm(II11)=r0; // destination ID2
r0=1; dm(IM11)=r0; // modifier = 1
r0=10; dm(C11)=r0; // counter = 10
r0=0x01;
dm(DMAC11)=r0; // enable to receive data
```

Note that the I/O processor does not use the EI, EM, and EC registers in slave mode DMA.

From the previous Master DMA transfer, ID1 transmits data to EPB1 buffer FIFO. ID2 detects that the data is present and performs the DMA transfer to internal memory, emptying the EPB1 buffer FIFO.

Master DMA from External SBSRAM

Note that in the previous DMA transfer, where a FIFO (EPB1) is used, master and slave DMA need to be configured.

A type of Master DMA where no slave needs to be set up is when the master writes/reads directly (without use of the EPBx slave FIFOs) from/to

external memory. The advantage of using this type of transfer is that only one DSP needs to be configured. Same setup can be used when reading/writing from/to internal memory of the MMS.

Note: direct read/write from/to internal memory of the MMS it's not supported by certain devices (ADSP-21065L and ADSP-21161). These devices access internal memory indirectly with the use of DMA.

In example 6, DMA channel 10 was used to perform the Master DMA transfer from external SBSRAM to ID1's internal memory as follows:

Example 6:

```
// Excerpt from ID1: Master DMA, DMA channel
10, transfer from SBSRAM to ID1
r0=0;
dm(DMAC10)=r0;// clear DMA Control Reg.
r0=ext_mem_data;
dm(EI10)=r0; // source SBSRAM
r0=1; dm(EM10)=r0; // modifier = 1
r0=10; dm(EC10)=r0; // counter = 10
r0=DMA_dest_ID1;
dm(II10)=r0; // dest. in ID1's int. mem.
r0=1; dm(IM10)=r0; // modifier = 1
r0=10; dm(C10)=r0; // counter = 10
ustatl=0x0400;
dm(DMAC10)=ustatl; // receive data
ustatl=dm(DMAC10);
bit set ustat1 0x1; // enable DMA channel
dm(DMAC10)=ustatl;
```

Like before, the DSP sets up the channel's parameter registers. The only difference with respect to the previous example is that there is no need to set up a slave DMA. The SBSRAM master mode DMA transfer to the internal memory will initiate once the channel's DMA enable (DEN) bit is set.

Core transfer

Core transfer is a different way of handling data where no DMA is used. In this case, the Data

Address Generators (DAGs) are used to directly transfer data from internal memory of ID2 to internal memory of ID1.

An example of this is shown below:

Example 7:

```
// Excerpt from ID2: Core transfer, ID2 to
ID1.
b1=DAG_source_ID2; // Source in ID2
l1=0; m1 = 1;
b8=DAG_dest_ID1; // Dest. in ID1
l8=0; m8 = 1;
r2=dm(i1,m1);
lcntr = N-1, do DAG_transfer until lce;
DAG_transfer: //Dual access in 1 cycle
r2 = dm(i1,m1), pm(i8,m8) = r2;
pm(i8,m8) = r2;
```

Two data arrays are declared, one in each DSP's internal memory. In example 7, ID2 accesses the array stored in ID1 through the MMS space.

The DAG registers are used to access the two data buffers to perform a direct data transfer.

Note that values are fetched from both program and data memory, resulting in a dual memory access and executing in just one cycle.

Some Performance Considerations

Core data transfers are a nice and fast way of transferring words of data since the code can be optimized to transfer a word of data per cycle. However, DMA is a better choice when large amounts of data need to be transferred since the core can be utilized for computational processing. *Remember that DMA transfers operate in the background freeing up the core.*

Also, a master DMA transfer can be configured by the slave DSP (or host), increasing performance in applications where the master DSP might be overloaded with processing activity. For more details on DMAs and data transfer, please refer to *EE-84 "SHARC® DMA Modes of Operation"* and the *I/O Processor*

chapter in the *ADSP-21160 SHARC® DSP Hardware Reference Manual*.

ID Checking

This routine can be used to check whether the executable file generated gets loaded into the correct DSP in the system. This code ensures that no ID mismatch occurs.

Example 8:

```
// Excerpt from ID2: ID Checking
R0=DM(SYSTAT); // get SYSTAT value
R1=FEXT R0 BY 8:3; // get the IDC value
R2=0x2; // ID=2
R1=R1-R2; // is this DSP ID2?
IF NE JUMP incorrect_ID;
// if incorrect jump to endless loop
```

Basically, it reads the DSP ID value from the SYSTAT register and it compares it with the theoretical value of the DSP ID. In this case, the code has been written for ID2, so it makes sure it has been loaded into the correct target, which is DSP 2. If false, it will enter an endless loop indicating that an error has occurred.

Note: loading executable file into wrong DSP will cause the program not to work properly since all MMS offset values will not correspond to the correct ones and therefore, inter-processor accesses will fail.

Multiprocessor Debugger Support

VisualDSP++™ Multiprocessor Debugger provides the user with full system evaluation using the Emulator. The Emulator allows code testing and evaluation on the hardware platform. I/O inter-processor communications as well as MMS data transfers are supported.

MP debugger operations like MP load, run or reset provide the user with the capability of testing the system with full synchronization of all DSPs.

Note: The VisualDSP++™ Simulator allows to fully test the algorithms and core code for each DSP in the system independently.

Some of the MP debugger features are:

- Multiprocessor debug commands allow the user to download, reset, restart, run and step through the code just like with single-processor commands, except that they work synchronously on all active DSPs in the selected MP group.
- The Debugger provides a Multiprocessor *Status* window. This window displays the current status of each DSP in the system: Running, Halted, or Unknown.
- The contents of each debugger window within an MP emulation debugger session reflects the selected DSP, i.e. the window in *Focus*.
- By default, the contents of each window will change depending on which DSP is in focus. The debugger supports *Pinning* windows (Memory, Registers, etc.) dedicating them to a specific DSP in the MP system. This will allow the user to dedicate a particular debugger window to only display information from one particular DSP in the system, as opposed to having the contents of the window change whenever a new processor is selected via the MP Status window.
- The debugger provides a Multiprocessor *Group* window from which the processors can be grouped into multiple, logical units upon which all MP commands are applied. This window is particularly useful when many processors are present in a system and the user wishes to control/debug subsets of these processors together.

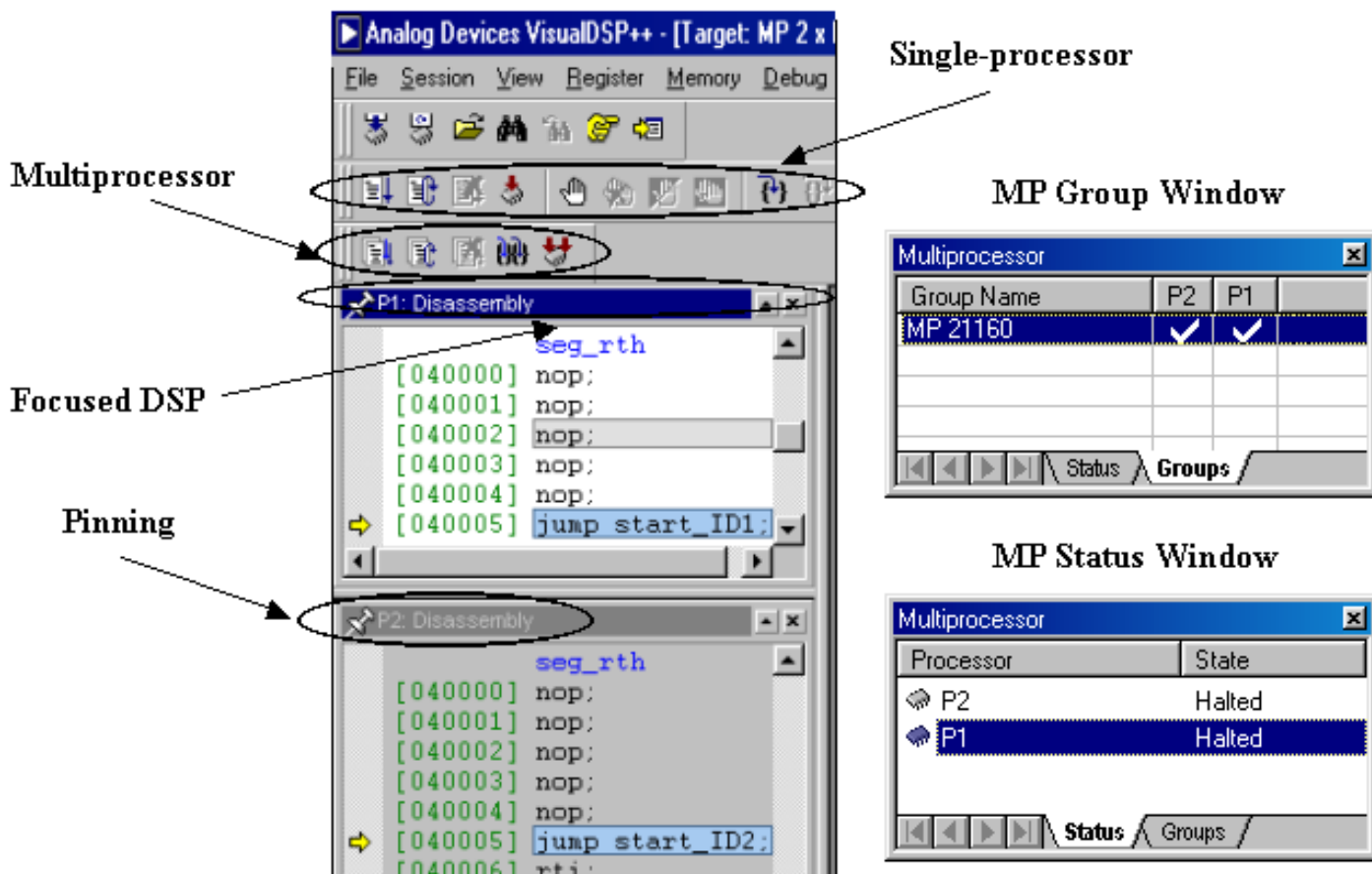


Figure 3 Multiprocessor Debugger Support

Use pinning, and the processor status items in the Multiprocessor window, in conjunction with single-processor debug commands to debug individual processors in an MP session.

VisualDSP ICE Configurator

The Debugger allows the use of emulator targets. The DSP In Circuit Emulator (ICE) is a development tool for debugging programs running in real time on DSP target system hardware. The emulator reads executable files and loads them into the DSP.

The ICE provides a controlled environment for observing, debugging, and testing activities in a target system by connecting directly to the target processor through its JTAG interface.

For the MP system emulation, the Summit-ICE Universal Emulator system was used. As a first step, the MP platform must be configured using the Visual DSP ICE Configurator. The Configurator is used to describe the user's hardware platform to the JTAG emulator. Once a platform has been described, an emulator target session can be based upon it. The following steps should be followed when configuring the MP platform:

1. Open the VisualDSP ICE Configurator.
2. Create a new platform.
3. Specify the name, number and type of devices to be included as part of the platform.

These steps are illustrated in Figure 4.

Please be aware of the *Initial Reset on Startup* option, which appears in the *Device Properties* window shown in Figure 4. Enabling this option will perform a complete reset on the selected device every time the emulator session is initiated. *In systems where some settings may need to be preserved (i.e. WAIT register) this option should be cleared.*

Note: there is also a similar option in the debugger itself, reset before loading executable, which performs a complete reset of all devices in the system upon downloading code to the DSPs. This option can be found under Settings/Target Options/.

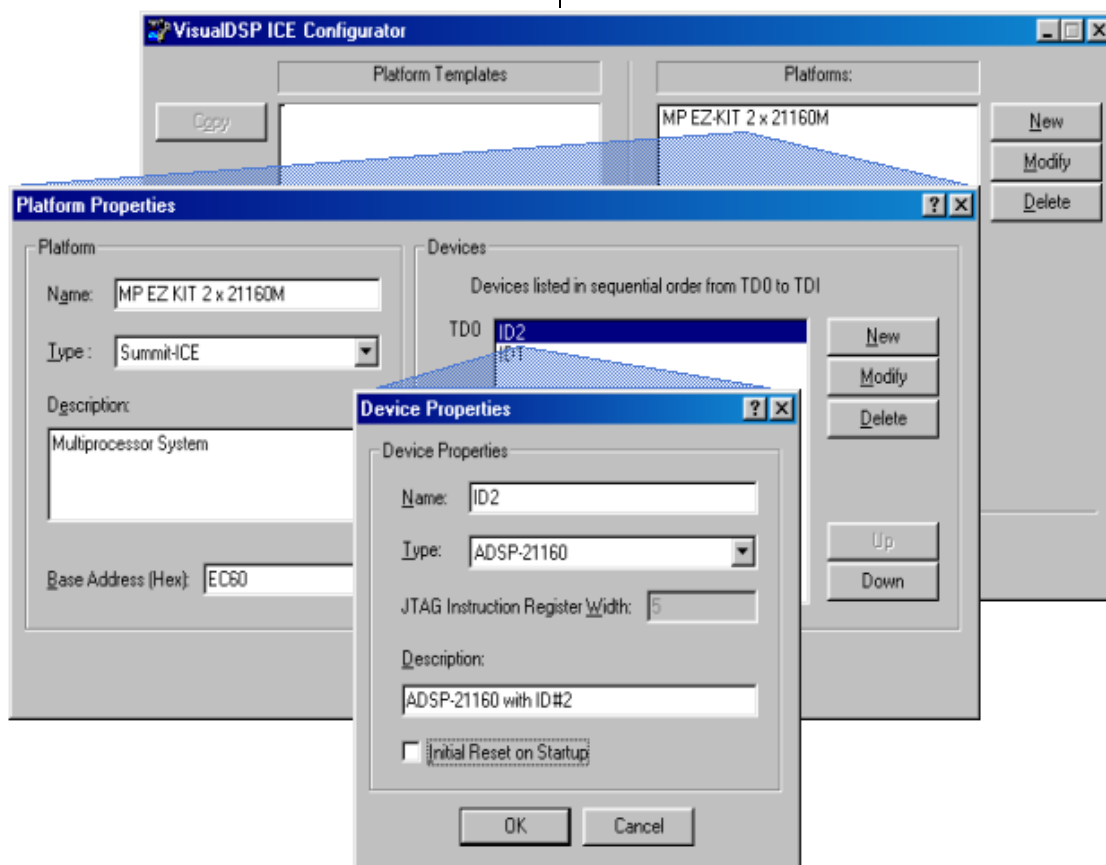


Figure 4 VisualDSP ICE Configurator

ICE Test Utility and JTAG Scan Test

Before getting into the actual system debugging, the ICE must be tested to make sure that has been properly configured. The ICE Test Utility (Figure 5) is used for this purpose. Open the utility, select the proper emulator I/O address, check the continuous scan box and start testing. The scan test will then be performed and the output window would look as follows after a successfully completed scan test:

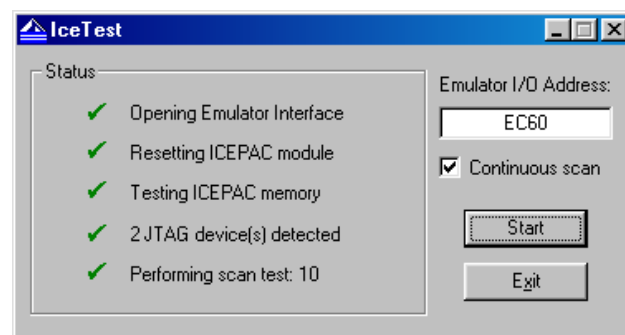


Figure 5 VisualDSP ICE Test Utility.

In case the test does not complete successfully, an error message will be displayed with a possible solution for the problem. Here is a description of some issues that should be kept in mind for the system design:

1. In a multiprocessor system it is imperative that the JTAG header is buffered. This will keep the signals clean and avoid noise problems that occur with longer signal traces (ultimately resulting in reliable emulator operation).

2. In one scan chain, it is not recommended to use more than eight physical devices (although, theoretically, the devices that can be supported in one JTAG scan chain by the software is about 50). The recommendation of not more than eight physical devices is mostly due to the transmission line effects that appear in long signal traces, and based on some field-collected empirical data.

3. The power-on sequence for the target and emulation system is as follows: Apply power to the emulator first, then to the target board. This ensures that the JTAG signals are in the correct state for the DSP to run free.

Please refer to *EE-Note 68 Analog Devices JTAG Emulation Technical Reference (2.5)* for a more detailed description on this topic.

MP System Emulation

Now that the MP project has been created and the emulator platform is ready for debugging, we can begin with the hardware emulation.

First of all, the DSP executable files (.DXE's) are downloaded to the corresponding DSPs. For MP emulation, a Load Multiprocessor

Confirmation window (Figure 6) appears. This window enables the user to select which .DXE file is loaded into which DSP.

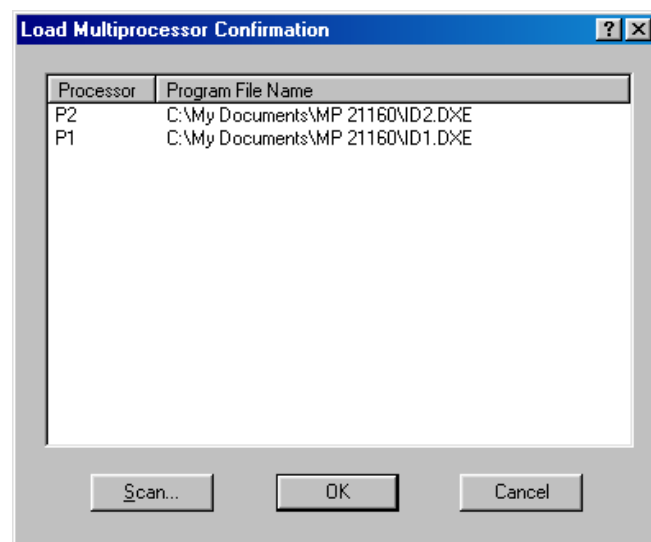


Figure 6 Load Multiprocessor Processor Window.

Once the code has been successfully loaded into each DSP, the system can be fully evaluated using the MP features previously described.

After running the code in both DSPs the user can view the contents in the data memory windows and should be able to verify that all data transfers between the two DSPs have completed successfully.

Figure 7 illustrates a classical example of some of the MP debugger windows that can be viewed when evaluating the system.

Running code in the DSP targets (synchronously in both DSPs or independently), setting up break points, viewing the memory contents, and system registers are just some of *VisualDSP++™* MP debugger capabilities.

The screenshot displays the VisualDSP++ Multiprocessor Session interface. The main window shows assembly code for two processors, ID1 and ID2. ID1's code includes initialization of broadcast data, DMA transfer, and interrupt generation. ID2's code handles DAG transfer and interrupt processing. The interface includes several data windows:

- ID1: Data(DM) Memory [Hexadecimal]:** Shows memory addresses and values for Broadcast_data, DMA_source_ID1, and DAG_source_ID1.
- ID2: Data(DM) Memory [Hexadecimal]:** Shows memory addresses and values for DMA_dest_ID2 and DAG_source_ID2.
- ID1: Interrupts Registers:** Displays registers like IRPTL, IMASK, and IMA.
- ID2: Interrupts Registers:** Displays registers like IRPTL, IMASK, and IMA.
- ID1: System Control Registers:** Displays registers like SYSCON, SYSTAT, and MSGRO.
- ID2: System Control Registers:** Displays registers like SYSCON, SYSTAT, and MSGRO.

The bottom of the window features an Output Window with console messages and a Processor window showing the state of ID2 and ID1 processors.

Figure 7 VisualDSP++™ Multiprocessor Session

References

ADSP-21160 SHARC® DSP Hardware Reference, Analog Devices Inc.

ADSP-21160 EZ-KIT Lite User's Guide, Analog Devices Inc.

VisualDSP++™ Linker & Utilities Manual for ADSP-21xxx Family of DSPs DSPs, Analog Devices Inc.

VisualDSP++™ Users Guide for ADSP21xxx Family DSPs, DSPs, Analog Devices Inc.

VisualDSP++™ Emulation Tools Installation Guide for Windows 95/98/NT/2000, Analog Devices Inc.

Analog Devices JTAG Emulation Technical Reference 2.5 (EE-68), Analog Devices Inc.

Understanding and Using Linker Description Files (LDFs) (EE-69), Analog Devices Inc.

SHARC® DMA Modes of Operation (EE-84), Analog Devices Inc.

Document History

	Description
March 31, 2003 by M.Kokaly-Bannourah	Updated document and code (<i>VisualDSP++™</i> 3.0 compatible)
October 11, 2001 by M.Kokaly-Bannourah	Initial Release