



## Introduction to TigerSHARC® Multiprocessor Systems Using VisualDSP++™

Contributed by Maikel Kokaly-Bannourah

April 04, 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 assembly code written for the ADDS-TS101S EZ-Kit Lite, consisting of two TigerSHARC® Processors, using VisualDSP++™ 3.0.

TigerSHARC® Multiprocessor systems can be configured in different ways:

- Several processors sharing the external bus
- Link Port point-to-point communication

This note will discuss the implementation of an MP system with the processors sharing the external bus. For more details on other implementations please refer to the *TigerSHARC® Processor Hardware Reference*.

In addition to the assembly code explored throughout this note, an MP code example written in C is also available.

### 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 8 ADSP-TS101S's). Also note that

a combination of different DSPs with different architectures (i.e. ADSP-TS101S and ADSP-21160) in the same LDF is not supported by VisualDSP++.

```

MEMORY
{
  MOCode      { TYPE(RAM) START(0x00000000) END(0x0000FFFF) WIDTH(32) }
  M1Data      { TYPE(RAM) START(0x00080000) END(0x0008FFFF) WIDTH(32) }
  [...]
}
MPMEMORY
{
  ID0 { START(0x2000000) } // Declare Multiprocessor Memory Space (MMS)
  ID1 { START(0x2400000) } // offset values for processors ID0 and ID1
}
SHARED_MEMORY //Declare external shared memory
{
  OUTPUT(shared.sm)
  SECTIONS
  {
    ext_mem
    {
      INPUT_SECTIONS( data.doj(ext_data) )
    } >SDRAM
  }
}
PROCESSOR ID0 // Processor ID0 sections definition.
{
  LINK_AGAINST(shared.sm, ID1.dxe)
  OUTPUT(ID0.dxe)
  SECTIONS
  {
    code
    {
      FILL(0xb3c00000)
      INPUT_SECTION_ALIGN(4)
      INPUT_SECTIONS( ID0.doj(program))
    } >MOCode
    [...]
  }
}
PROCESSOR ID1 // Processor ID1 sections definition.
{
  LINK_AGAINST(shared.sm, ID0.dxe)
  OUTPUT(ID1.dxe)
  SECTIONS
  {
    code
    {
      FILL(0xb3c00000)
      INPUT_SECTION_ALIGN(4)
      INPUT_SECTIONS( ID1.doj(program))
    } >MOCode
    [...]
  }
}

```

Figure 1 Excerpt from and MP LDF example

An MP LDF example where all the above commands are used is shown in **Figure 1**. The remaining 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 TigerSHARC® DSPs*” or to “*EE-69 Understanding and Using Linker Description Files (LDFs)*” for a general description on LDF files).

In **Figure 1**, a 2 ADSP-TS101S and external shared memory system is defined

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 processor.

For MP system hardware configuration please refer to *Cluster Bus* chapter of the *TigerSHARC® Processor Hardware Reference*.

## 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 processors in an MP system. The ADSP-TS101S's multiprocessor memory space appears in **Figure 2**.

	0x04000000
Processor ID 7	0x03C00000
Processor ID 6	0x03800000
Processor ID 5	0x03400000
Processor ID 4	0x03000000
Processor ID 3	0x02C00000
Processor ID 2	0x02800000
Processor ID 1	0x02400000
Processor ID 0	0x02000000
Broadcast	0x01C00000
<b>RESERVED</b>	
	0x003FFFFFF
Internal Memory	0x00000000

Figure 2 ADSP-TS101S's Multiprocessor Memory Space.

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 0x3000000 – 0x33FFFFFF, is equivalent to accessing the internal memory of the DSP in the MP system with ID 4.

*Note: A TigerSHARC's® own Internal space can be accessed via the multiprocessing space for write transactions only. This, however, is performed through the external bus and should*

*not be used or other than in special cases where data must pass through the TigerSHARC® bus interface. Performing a self-multiprocessor read access will set the SELF MPROC READ bit in the SYSTAT register as an error indication for the illegal access.*

**Example 1** shows source code where the MMS is used to access a memory location of another DSP in the system. In this case DSP with ID0 accesses a memory location in ID1's internal memory.

**Example 1:**

```
// Code in ID0
r0=0x02480000; //M0 Data in int mem of ID1
```

In **example 1**, the MMS address for ID1 is 0x2400000, which is then added to the address corresponding to ID1's internal memory (0x80000). Therefore, this will result in a read access of ID0 from ID1's internal memory.

*Note: In DSP multiprocessor systems including SDRAM a DSP with ID=000 must be present, since this DSP performs the initialization (MRS) of the SDRAM. Also, there are issues related to open drain pull-ups only enabled on the DSP with ID=000. After reset, ID0 becomes the bus master and priority rotates in a round robin fashion, going up from the present master.*

## External Memory

The ADSP-TS101S has 6Mbits of on-chip SRAM memory that can store both program and data. However, some applications also require the use of external memory devices.

External memory is widely used in MP systems, and can be implemented as a shared resource for all DSPs in the system, or dedicated to a particular processor.

It is very important to keep in mind that all DSPs in the system must set up the proper access mode for the type of memory used in the hardware system. The access mode is

programmed via the System Control (SYSCON) register. Default power up/reset settings for SYSCON are detailed in the *TigerSHARC® Processor Hardware Reference*. User defined settings must support the access mode appropriate to the memory device(s) that the user intends to use in their hardware systems. SYSCON settings must be the same across all devices sharing the cluster bus connection.

Note that SDRAM is gluelessly supported by the ADSP-TS101S. As with the SYSCON register, the SDRAM Configuration (SDRCON) register 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 this project, SDRAM was used as an external shared resource for the two DSPs in the system. The code shown in **example 2** corresponds to the SDRCON register initialization, which, as previously mentioned, must be done by all DSPs sharing the same external memory (in this case, ID0 and ID1).

#### Example 2:

```
// Excerpt from ID0 & ID1: SDRAM Init.

j11 = j31 + 0x00005913;;
SDRCON = j11;;

// Enable SDRAM: ENA=1,CAS=2CL, pipedepth=0,
page=512w,
ref.rate=1200, trp=2, tras=5, init=1
```

*Shared.asm* contains the variable definitions for the data that will be placed in external memory.

*Note: the DSP with the lowest ID number (and therefore highest priority in the system) is responsible for initializing the external data defined in the .ASM shared memory file during the booting-up sequence.*

## Vector Interrupt (VIRPT)

Vector Interrupts are used for inter-processor communication between a host and a DSP or between DSPs.

This interrupt is a general-purpose interrupt for another master's use. The host (or master DSP) can issue a vector interrupt to the slave 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 3:

```
// Extract from ID1: VIRPT Generation

j1 = MMS_ID0 + VIRPT_REG;;
xr0 = VIRPT_ISR_ID0 - MMS_ID0;;
[j1 += j31] = xr0;;
```

In **example 3**, ID1 triggers a vector interrupt in ID0 by writing the address of the service routine to be served in ID0 (labeled VIRPT\_ISR\_ID0) to the VIRPT register in ID0 (0x2180730 = 0x180730 VIRPT address + 0x2000000 MMS ID0).

Note: In case an external defined label is used for the ISR address (like in this example, i.e. VIRPT\_ISR\_ID0), the MMS offset value of the DSP serving the interrupt (MMS\_ID0) must be subtracted for the DSP to vector to the address of the ISR correctly.

This is just an example of how inter-processor 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, set of data or memory location 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 4**):

1. Request bus lock by setting the BUSLK bit in the BUSLK register.
2. Wait for bus mastership to be acquired (condition codes BM and NBM can be used).
3. Read the semaphore, and write to it.

**Example 4** is an excerpt from ID0's code demonstrating the use of Bus Lock in combination with a Broadcast write:

**Example 4:**

```
// Excerpt ID0 code: BROADCAST write using
// Bus Lock

j11 = j31 + 1;;
BUSLK = j11;;          // Lock the bus

BUS_MASTER: if NBM, jump BUS_MASTER;;
// Check for bus mastership

DCS0 = xr3:0;;
yr0 = MMS_Broadcast;;
DCD0 = yr3:0;;
// Perform Broadcast data transfer

j11 = j31 + 0;;
BUSLK = j11;; // Relinquish the bus
```

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 (NBM) condition code. If it has become the bus master, the DSP can proceed with the external read or write. If not, it can either clear its BUSLK bit and try again later, or simply wait until the bus is acquired.

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 external port (EP) data transfers have been implemented:

1. Direct Memory Access (DMA) between ID0 and ID1,
2. DMA from ID0 and ID1 to external memory (SDRAM),
3. Core transfer from ID1 to ID0,
4. Broadcast Write to all DSPs in the system.

The TigerSHARC® includes 14 DMA channels, four of which are dedicated to external memory devices: channels 0, 1, 2, and 3.

For details on DMAs and how the different data transfers are performed, please refer to “*EE-143 Understanding DMA on the ADSP-TS101S TigerSHARC®*” and the *DMA Controller* chapter of the *TigerSHARC® Processor Hardware Reference*.

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

### DMA transfer from ID0 to ID1

This example shows a DMA transfer from internal memory of ID0 to internal memory of ID1. In this case, DMA channel 0 is used to transmit the data stored in *tx\_ID0* to *rx\_ID1*.

For this kind of transmission, two transfer control blocks (TCBs), one for the source and another one for the destination, must be set up.



**Example 5** shows the loaded values into each TCB using DMA channel 0. Note that the value in register `xr2/yr2` is irrelevant due to the fact that 2-dimensional DMA is not selected for this particular example. As soon as both the source and destination TCBs are loaded with values the DMA transfer starts.

Once the DMA is completed an interrupt occurs and the `_dma_int` vector interrupt routine is then run.

#### Example 5:

```
// Excerpt from ID0: DMA0
// External Port transfer from ID0 to ID1
xr0 = tx_ID0;; // xr0=index= ID0
xr1 = 0x00100004;; // count=0x10,modify=4
xr3 = 0x47000000;; // int mem,prio=norm,
//2D=no,word=quad,int=yes,RQ=enbl,chain=no
DCS0 = xr3:0;; // Source

yr0 = rx_ID1;; // xr0=index= ID1
yr1 = 0x00100004;; // count=0x10,modify=4
yr3 = 0x87000000;; // ext mem,prio=norm,
//2D=no,word=quad,int=yes,RQ=enbl,chain=no
DCD0 = yr3:0;; // Destination
```

#### DMA transfer from ID0 to SDRAM

The transfer of the data from internal memory of ID0 to the SDRAM is executed with minor alterations to the previous example. The source TCB is loaded with the same contents as before (`xR3:0`) and the destination TCB is written with the values of registers `yr3:0` where `yr0` is altered from `rx_ID1` to `shared_data`.

In this case, DMA channel 1 is used for the data transfer by replacing `DCS0` and `DCD0` with `DCS1` and `DCD1` respectively.

#### Example 6:

```
// Excerpt from ID0: DMA0
// Ext Port transfer from ID0 to SDRAM
DCS1 = xr3:0;; // Same as before
yr0 = shared_data;; // xr0=index= SDRAM
DCD0 = yr3:0;; // Destination
```

Note: writing to an active TCB, i.e. back-to-back DMA using the same channel before the current transfer has completed, results in an illegal operation. An error indication will be flagged in the DMA status register (`DSTAT`).

#### DMA from ID1 to SDRAM

As in **example 6**, this is a DMA data transfer from internal memory (in this case from ID1 instead of ID0) to the SDRAM.

Again, the same concepts apply, where the source and destination TCBs are set up as shown in **example 7**.

#### Example 7:

```
// Excerpt from ID1: DMA0
// External Port transfer from ID1 to SDRAM

xr0 = tx_ID1;; // xr0=index= ID1
xr1 = 0x00100004;; // count=0x10,modify=4
xr3 = 0x47000000;; // int mem,prio=norm,
//2D=no,word=quad,int=yes,RQ=enbl,chain=no
DCS0 = xr3:0;; // Source

yr0 = shared_data+TAPS;; //xr0=index=SDRAM
yr1 = 0x00100004;; //count=0x10,modify=4
yr3 = 0x87000000;; //ext mem,prio=norm,
//2D=no,word=quad,int=yes,RQ=enbl,chain=no
DCD0 = yr3:0;; // Destination
```

The source TCB is loaded with the same contents as before (`xR3:0`) with the only variation that the index now points to internal memory of ID1, `tx_ID1`. The destination TCB is written with the values of registers `yr3:0` where `yr0` now points to `shared_data+TAPS`. `TAPS` is the offset value used to point to the second half of the buffer declared in SDRAM to prevent from overwriting the already transferred data by ID0. Once again, DMA channel 0 is used.

#### Core transfer from ID1 to ID0

Core transfer is a different way of handling data where no DMA is used. In this case, the Integer Arithmetic Logic Unit (IALU) is used to directly transfer data from internal memory of ID1 to internal memory of ID0.

An example of this is shown below:

#### Example 8:

```
// Excerpt from ID1: Core transfer
// from ID1 to ID0 using the IALU

jB0 = tx_ID1;; // Base address in ID1
j0 = jB0;; // Set index equals to base
jL0 = TAPS;; // Set buffer length
```

```

jB1 = rx_ID0;; // Base register in ID0
j1 = jB1;; // Set index equals to base
jL1 = TAPS;; // Set buffer length

j4 = 1;; // Set loop modifier
LC0 = TAPS;; // loop counter

write_ext:  xr0 = CB[j0 += j4];;
            // read data from tx_ID1
            CB[j1 += j4] = xr0;;
            // write data to rx_ID0

if NLCOE, jump write_ext;;
// keep looping until completion

```

Two data arrays are declared, one in each DSP's internal memory. ID1 writes to the array stored in ID0 through MMS (*tx\_ID1* to *rx\_ID0*). The IALU registers are used to access the two data buffers to perform the direct data transfer.

### Some Performance Considerations

Core data transfers are a nice and fast way of transferring words of data since there is no need to set up a transfer control block of any kind. 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.*

## ID Checking

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

### Example 9:

```

// Extract from ID1: ID Checking
xR0 = 0x3;; // FEXT operand
xR1 = SYSTAT;; // read SYSTAT
xr2 = 1;; // DSP = ID1
xR1 = FEXT R1 by R0;; // get ID value
xr1 = r2 - r1;; // is DSP ID1?
if NAEQ, jump incorrect_ID;;
// if false, stop and enter 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 ID1, so it makes sure it has been loaded into the correct target that is

DSP 1. If false, it will enter an endless loop indicating that an error has occurred.

## Multiprocessor Support

## Debugger

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. 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.

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.

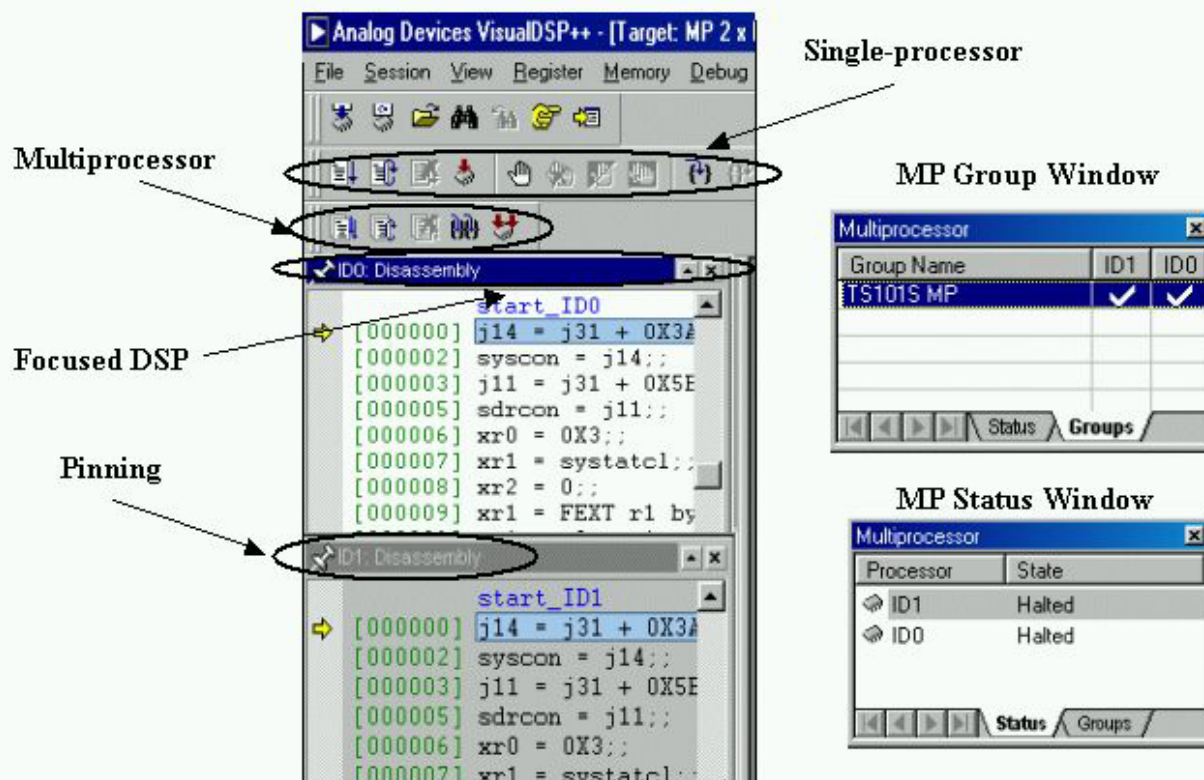


Figure 3 Multiprocessor Debugger Support

### 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 VisualDSP 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 Visual DSP 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. SDRCON 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/.*

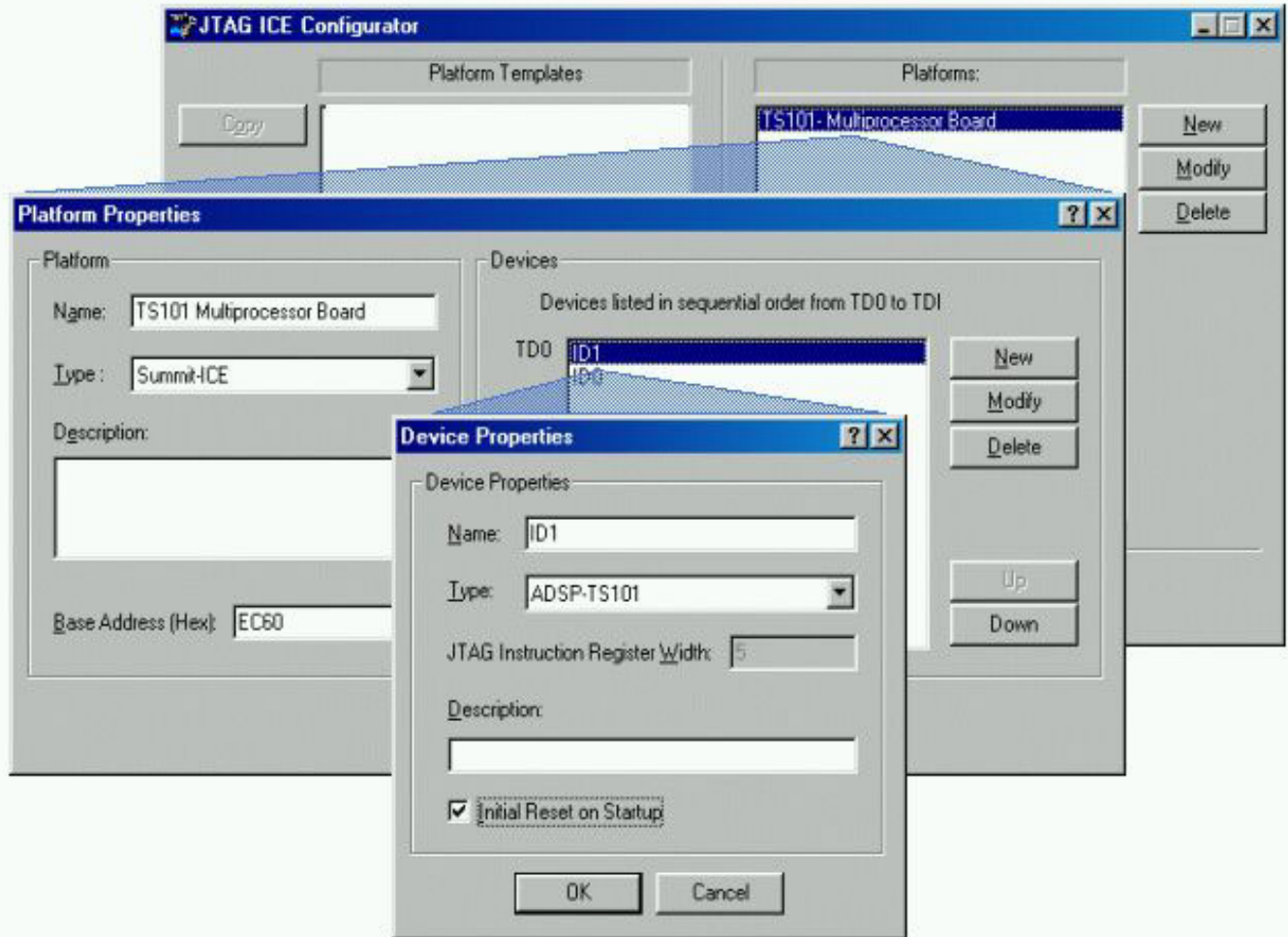


Figure 4 VisualDSP ICE Configurator

### 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:

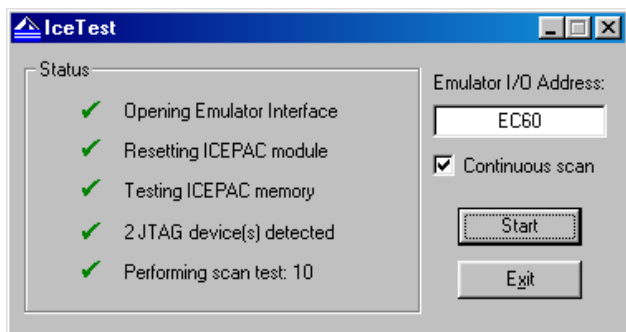


Figure 5 VisualDSP ICE Test Utility

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 recommended power-up procedure for the target and emulation system is as follows:

- Power up PC with POD connected to the PC but not the target.
- Power up the target with jumpers on /TRST and TCK of the JTAG header.
- Remove jumpers on /TRST and TCK from the JTAG header of target.
- Connect POD to JTAG header of target.
- Open VisualDSP++™ IDDE.

Similarly, the recommended power-down procedure is as follows:

- Close VisualDSP++™ IDDE.
- Disconnect POD from JTAG header of target.
- Power-down the target.
- Replace jumpers on /TRST and TCK of the JTAG header of target (for next power up).

Please refer to “EE- 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.

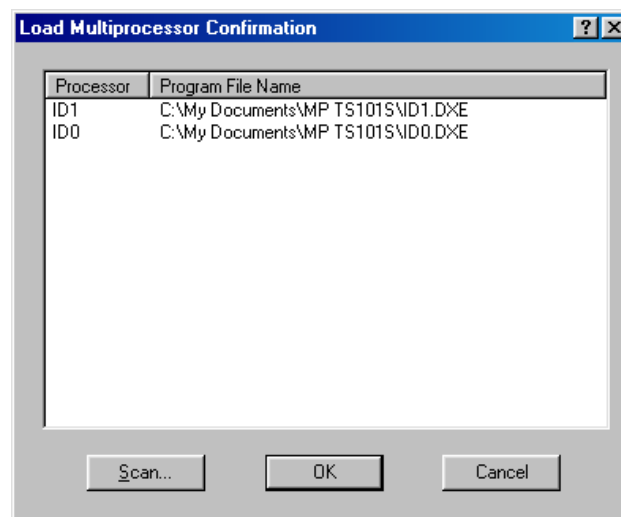


Figure 6 Load Multiprocessor Processor Window

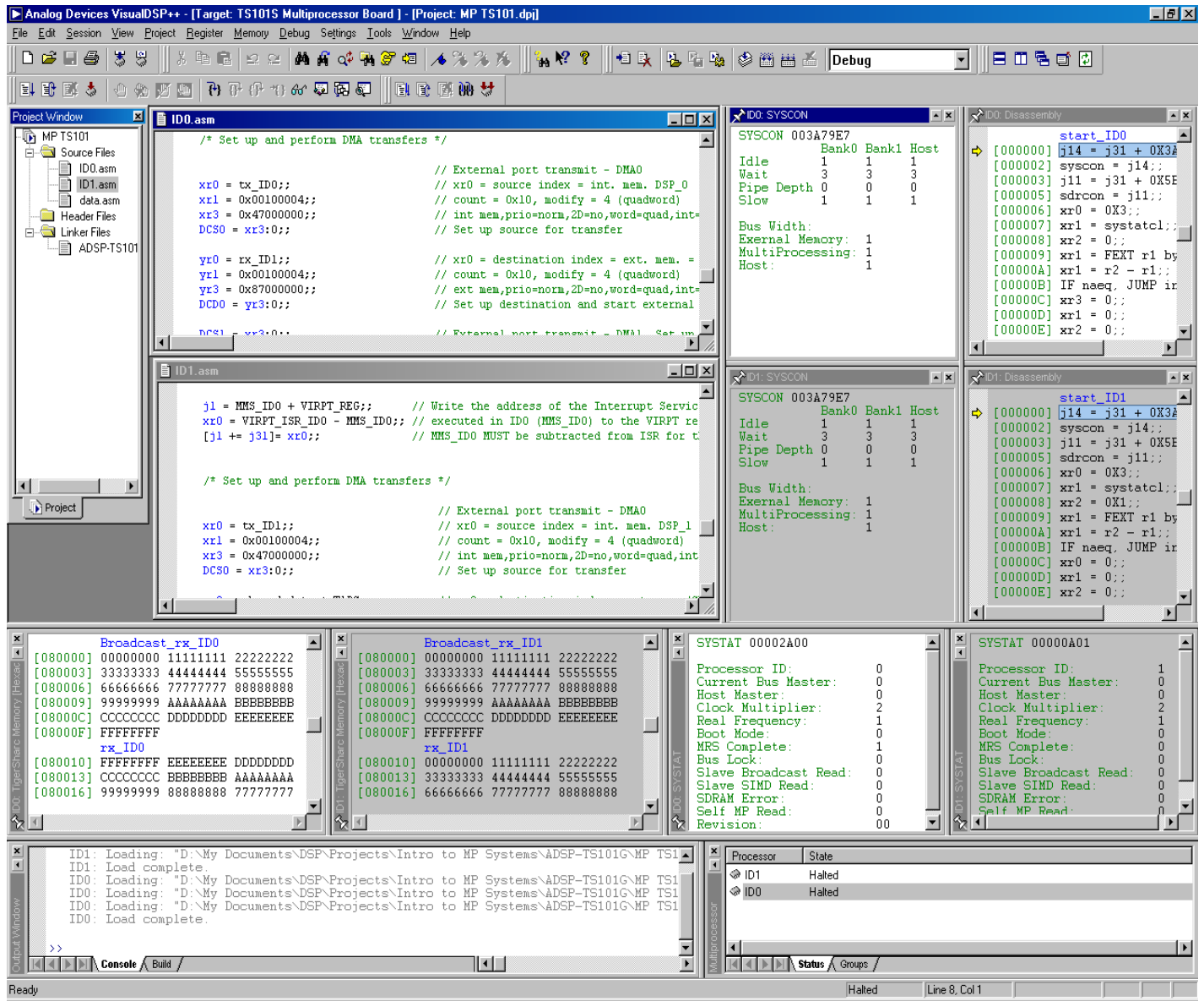
First of all, the DSP executable files (.DXE’s) are downloaded to the corresponding DSPs. For MP emulation, a Load Multiprocessor Confirmation window (Figure 6) appears. This window enables the user to select which .DXE file is loaded into which DSP.

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++™ Multiprocessor debugger capabilities.



**Figure 7** VisualDSP++™ Multiprocessor Session

## References

- [1] TigerSHARC® Processor Hardware Reference, First Edition, March 2003. Analog Devices Inc.
- [2] VisualDSP++ Linker & Utilities Manual for TigerSHARC® DSPs, Analog Devices Inc.
- [3] VisualDSP++ Emulation Tools Installation Guide for Windows 95/98/NT/2000, Analog Devices Inc.
- [4] Analog Devices JTAG Emulation Technical Reference (2.5) (EE-68), Analog Devices Inc.
- [5] Understanding DMA on the ADSP-TS101S TigerSHARC® (EE-143), Analog Devices Inc.

## Document History

Version	Description
April 04, 2003 by Maikel Kokaly-Bannourah	Updated trademark usage and upgraded code example according to VisualDSP++ release 3.0
June 26, 2002 by Maikel Kokaly-Bannourah	Initial Release