

## Inter-processor Communication

### 1 Introduction

In many system on chip (SoC) environments, there are often many processing tasks that are more suited to a Digital Signal Processor (DSP), such as modem processing, voice processing, audio processing, video processing, and many tasks that are more suited to a Micro-Controller ( $\mu$ C), such as protocol stacks, man-machine interface. This will normally lead to an SoC architecture with one or more DSP cores and one or more  $\mu$ C cores on the same die. Obviously, these processors need to have some mechanism to facilitate communications between the different tasks. This application note outlines some of the common methods of achieving this inter-processor communication, with particular reference to the interface between the  $\mu$ C and the SP-x series of DSP cores using the DSP-Shuttle™ bus controller.

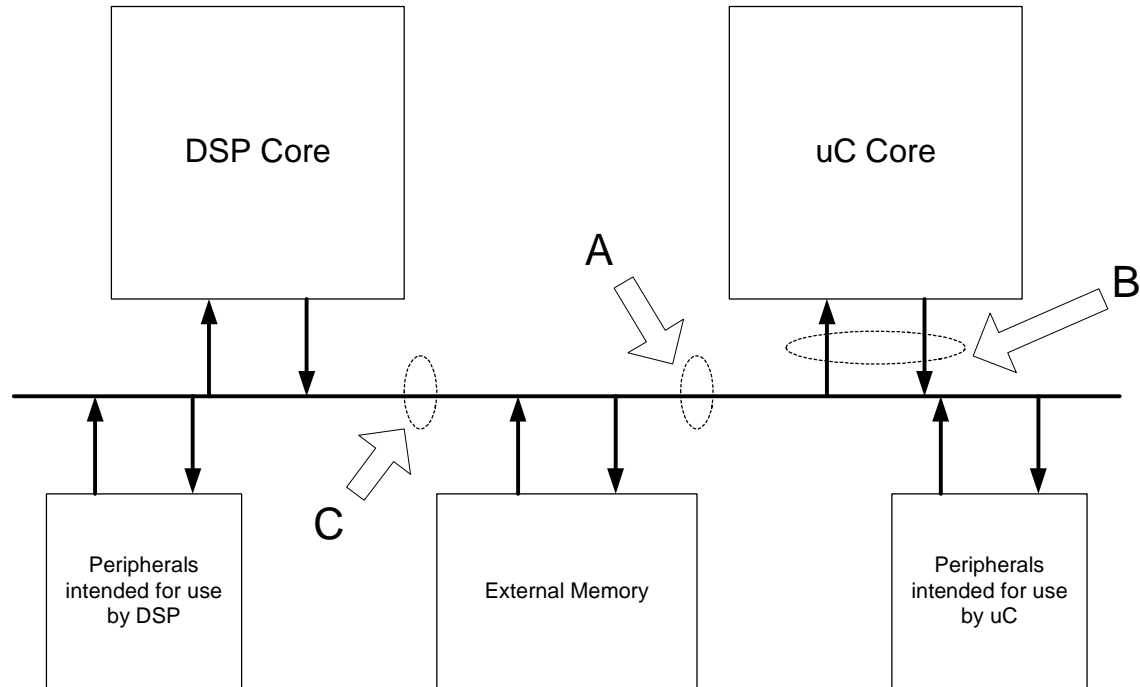
### 2 Basic SoC Architecture

For discussion purposes, we will use an example SoC architecture of one  $\mu$ C core and one DSP core. Each processor has its own local memory, which that processor can access very efficiently. There is also a larger, external memory system off chip, some peripheral devices that the  $\mu$ C needs to talk to, such as keyboard, screen etc., and some peripherals that the DSP needs to talk to, such as speaker, microphone, radio interface, modem line interface, etc. See Figure 1 below.

The DSP-Shuttle has been optimized specifically for the high throughput, data-centric data flow that is characteristic of a typical DSP system. 3DSP also has several bridges available to other processors such as the AMBA™ bus, which is standard in many SoC environments using the ARM® micro-controller. The bridge could be placed at either point A, point B or point C in Figure 1, depending on the particular data-flow that is present in the system designed. This allows both processors to access all peripherals, but keeps the DSP's peripherals on the DSP-Shuttle to allow its superior data-flow properties to solve the high throughput and real time constraints that are normally imposed on those peripheral communications.

The bridge is transparent as far as the software running on the DSP or the  $\mu$ C is concerned. Its function is to interface between the different signal sets and signaling protocols that are present on the two busses.

Normally the  $\mu$ C is in charge of high-level scheduling of all tasks in the system. This would require the  $\mu$ C to tell the DSP to initiate a task; when the task has completed then the DSP would report the results to the  $\mu$ C.



**Figure 1 Typical SoC Architecture**

### 3 How to Communicate

There are several ways that communication can be handled between the  $\mu\text{C}$  core and the DSP core. These depend a little on how many possible classes of interactions there could be, the real-time requirements on those interactions, and the amount of information that needs to be passed in each interaction.

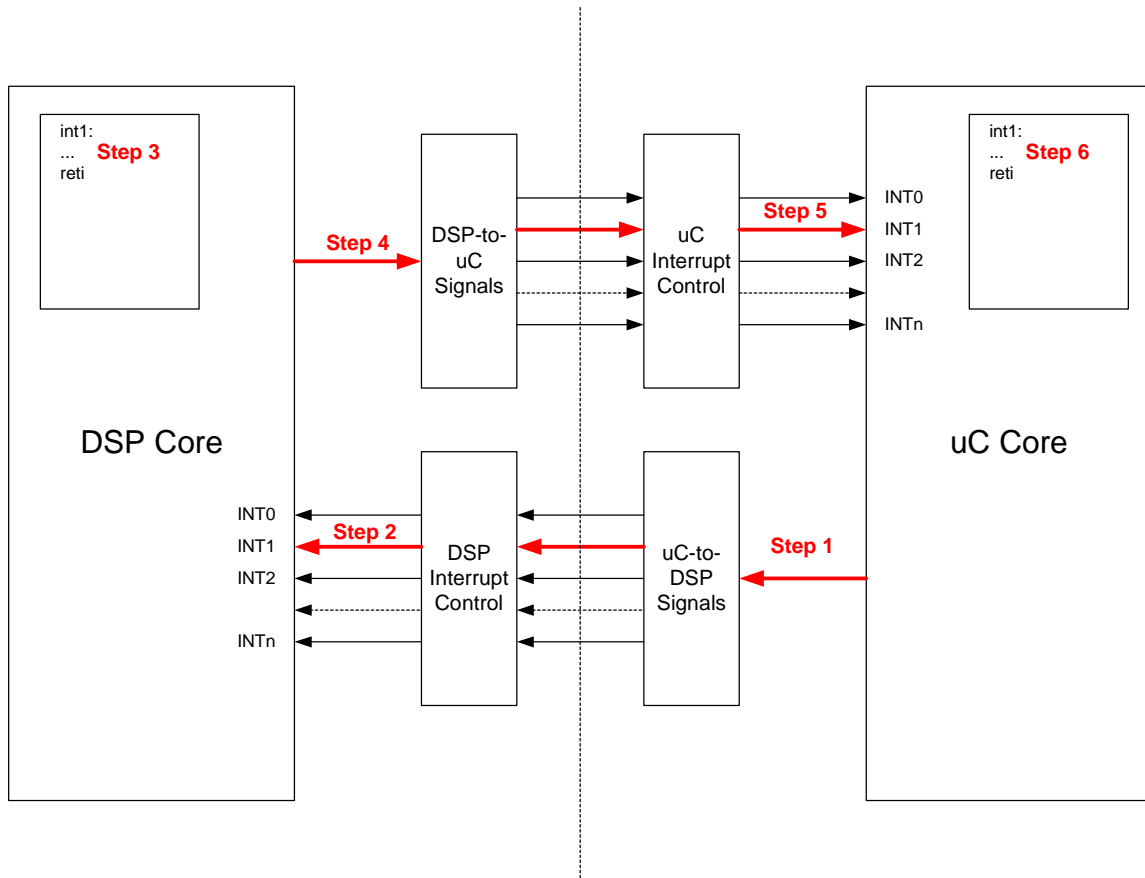
A few examples are given here to show how some different scenarios could be handled.

#### 3.1 Fast and Simple

If the interactions are very simple but need to be completed in a very time-sensitive manner then interrupts should be used. In this approach there would be a control register mapped into the  $\mu\text{C}$ 's local memory space, which could be written by the  $\mu\text{C}$ . Each bit in this control register would be connected to an interrupt signal on the DSP. Similarly, there would be a control register mapped into the DSP's local memory space, which could be written by the DSP. Each bit in this control register would be connected to an interrupt signal on the  $\mu\text{C}$ .

In this scenario, each of the interrupts to the DSP could trigger a specific process, such as collecting a frame of data from the audio input, compressing it and putting the data into a predefined buffer in external memory. There may not need to be any other information that the DSP needs in order to complete this request. Therefore, a single interrupt signal is sufficient to initiate the DSP's task. Upon completion of that task, the DSP would cause an interrupt to be sent back to the  $\mu\text{C}$ . This would be interpreted as completion of the task and then the  $\mu\text{C}$  could continue with its own tasks.

The code structures on the DSP and on the  $\mu\text{C}$  are both very simple, but no information beyond Start and Stop is passed between the tasks.



**Figure 2 Connection of Interrupt Signals between Cores**

The sequence of operations depicted in Figure 2 is listed below:

- #1 uC writes to its local control register
- #2 DSP interrupt triggered
- #3 DSP ISR runs; this is the requested DSP Task
- #4 DSP writes to its local control register
- #5 uC interrupt triggered
- #6 uC ISR runs; this lets uC know the DSP Task is done

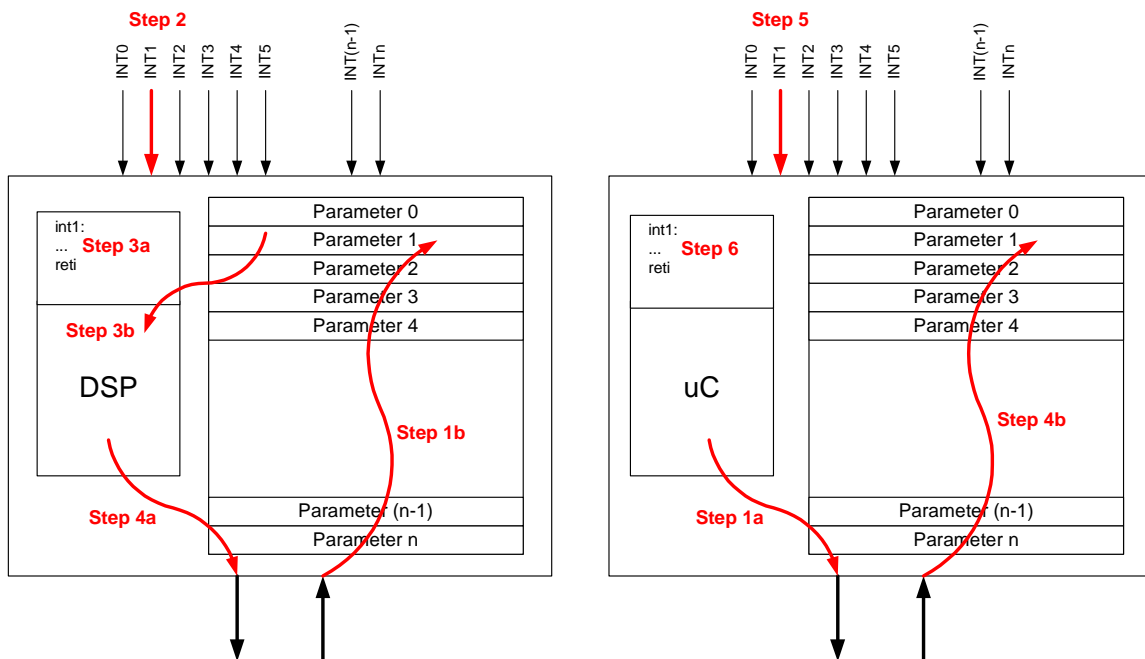
### 3.2 Fast and Simple with Some Parameters

Now let's examine the case where the interactions are still simple but at least one parameter needs to be passed to the DSP when the task is initiated, and some tokens must be passed back to the  $\mu\text{C}$  upon completion (e.g. to indicate if any error conditions occurred).

In this scenario, a simple interrupt is no longer sufficient. Some other data needs to be passed along with the interrupt. This can be achieved by defining a data structure in the DSP's local memory space; each element in the data structure is allocated for each interrupt service routine that can be triggered by the  $\mu\text{C}$ . When the  $\mu\text{C}$  wants to start a task on the DSP, it will first write some parameters into the data structure in the DSP's memory which is associated with this task. Once the data has been written, the  $\mu\text{C}$  forces an interrupt to the DSP. The first thing the DSP does upon entering the Interrupt Service Routine (ISR) is to read the data structure and use those parameters to decide what it should really be doing.

Similarly, the  $\mu\text{C}$  has a data structure in its local memory space, where each element also corresponds to its own ISRs. When the DSP has finished its assigned task, it would first write a word into the appropriate place in the  $\mu\text{C}$ 's data structure, and then send an interrupt to the  $\mu\text{C}$ . The  $\mu\text{C}$ 's ISR would read the value in the data structure and use this to assess the completion status of the task.

The code structures on the DSP and on the  $\mu\text{C}$  are still quite simple, but some small amount of information can be passed back and forth. Maintaining the data structures requires some more careful considerations in the code to ensure that there are no handshaking confusions. In general, this can be avoided if the  $\mu\text{C}$  makes sure it doesn't write to the DSP's parameter list between giving an interrupt (Start command) and receiving its own interrupt from the DSP (Stop command), and the DSP makes sure it doesn't write to the  $\mu\text{C}$ 's parameter list between giving an interrupt to the  $\mu\text{C}$  (Stop command) and receiving its own next interrupt (next Start command).



**Figure 3 Initiating DSP Tasks with Parameters**

The sequence of operations depicted in Figure 3 is listed below:

- #1a uC writes parameter out
- #1b parameter is written across bridge into DSP's memory
- #2 DSP interrupt triggered
- #3a DSP ISR runs; this is the requested DSP Task
- #3b DSP reads data from Parameter 1 and performs operation
- #4a DSP writes parameter out
- #4b parameter is written across bridge into uC's memory
- #5 uC interrupt triggered
- #6 uC ISR runs; this reads the Parameter 1

### 3.3 When Large Blocks of Data Need to be Communicated

Sometimes a large block of data may be identified by the  $\mu$ C as being ready for processing by the DSP. One method to proceed would be for the  $\mu$ C to move that large block of data into the DSP's local memory and then trigger an interrupt. However, it is often more convenient if the block of data itself is not moved, but instead a pointer to the block of data is passed to the DSP and the DSP then goes and reads in the large data block as it needs to process it. The pointer to the data structure can be passed using the mechanisms described in the previous section.

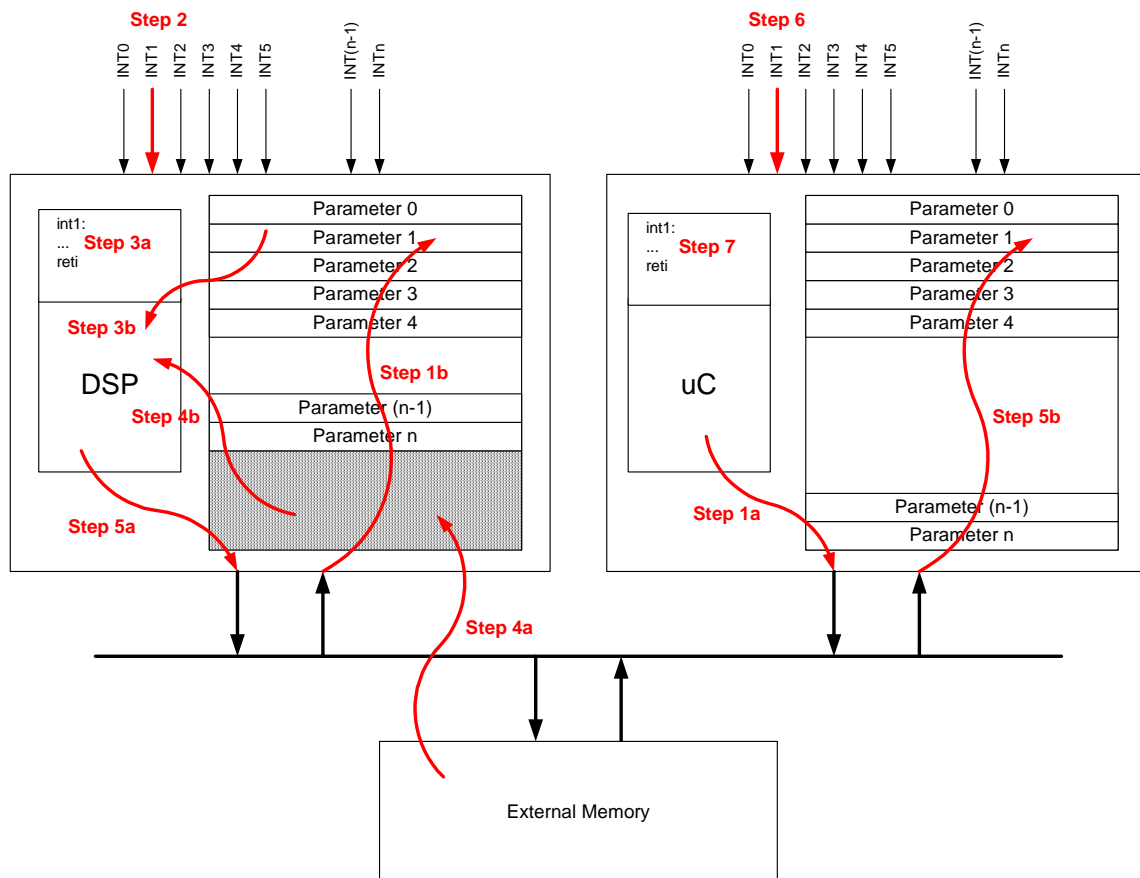


Figure 4 Passing a Pointer to a Large Data Block into the DSP Task

The sequence of operations depicted in Figure 4 is listed below:

```
#1a  uC writes parameter out
#1b  parameter is written across bridge into DSP's memory
#2   DSP interrupt triggered
#3a  DSP ISR runs; this is the requested DSP Task
#3b  DSP reads pointer from Parameter 1
#4a  DSP moves data from pointer address to local memory
#4b  DSP Task processes data block
#5a  DSP writes parameter out
#5b  parameter is written across bridge into uC's memory
#6   uC interrupt triggered
#7   uC ISR runs; this reads the Parameter 1
```

### 3.4 When Complex Task Control is Necessary, Using Polling

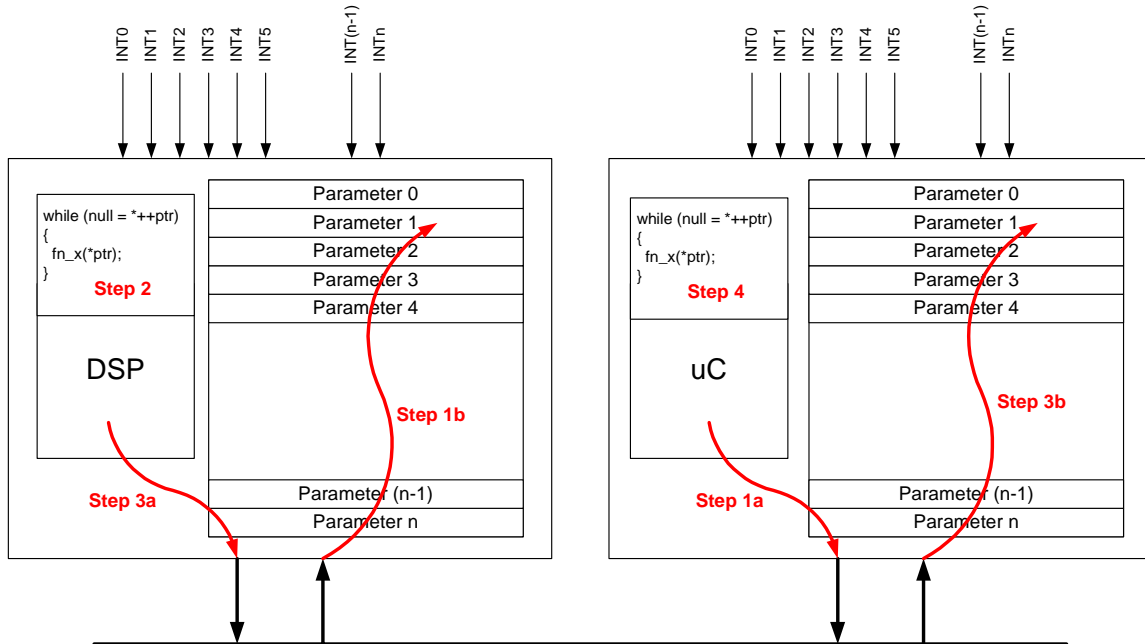
In some situations, there are many different tasks that could be running in many different modes on the DSP core. The scheduling of the individual tasks may not be absolutely time critical, but a large number of tasks need to be run within a given space of time. In this case, using interrupts from the  $\mu\text{C}$  to initiate task execution on the DSP can become inefficient. Instead, the DSP should run its own scheduler and receive messages signaling what tasks need to be run and in what modes. The scheduler on the DSP can then sift through the pending tasks and organize the processing in an efficient manner.

One method to achieve this is to maintain a data structure in the DSP's local memory where each possible task on the DSP has a location in the data structure. A similar data structure will exist in the  $\mu\text{C}$ 's local memory. When a task is to be initiated with some parameters, the  $\mu\text{C}$  simply writes into the DSP's data structure. No interrupt is sent from the  $\mu\text{C}$ .

The DSP's scheduler will periodically poll the parameter list to see what is requested by the  $\mu\text{C}$ . It can then run tasks as it sees fit. When a task is completed, the DSP will clear the parameter list in its local memory that corresponds to that task and then writes into the  $\mu\text{C}$ 's data structure. Some time later, the  $\mu\text{C}$ 's scheduler will poll through its parameter list and see the message from the DSP. It will process the message and then clear its own data structure.

In this way, tasks are started on the DSP and their completion is signaled back to the  $\mu\text{C}$  but no interrupts are issued. This allows for more complex interactions and also allows the DSP to apply some more intelligence in the task scheduling at a granularity that can be much finer than if the  $\mu\text{C}$  initiates all tasks.

The DSP's scheduler may need to be re-modeled for each new application in order to optimize operation.



**Figure 5 Message Passing by Polling**

The sequence of operations depicted in Figure 5 is listed below:

- #1a uC writes parameter out
- #1b parameter is written across bridge into DSP's memory
- #2 DSP runs its polling loop to find non-null parameter
- #3a DSP writes parameter out
- #3b parameter is written across bridge into uC's memory
- #4 uC runs its polling loop to find non-null parameter

### 3.5 When Complex Task Control is Necessary, Using Message Passing

A more generic method to solve this same scheduling scenario is to allow the DSP's scheduler to be based on a simple Operating System (OS) environment. There are many OS packages available for DSPs; one that has been highly optimized for the SP-x series of processors is SPEEDi™. This is designed as a very small, fast, robust kernel that handles task scheduling, task prioritization, semaphore passing and message passing.

To achieve the control necessary in the previous example, SPEEDi can be used efficiently. At initialization, the necessary tasks are created on the DSP but are dormant. A small data structure is maintained in the DSP's local memory. When the  $\mu$ C wants to activate a task on the DSP, it writes a message into the DSP's data structure and then signals an interrupt. In this case, only one interrupt line to the DSP is necessary; when that ISR runs it starts to parse through the list of commands in the data structure.

Each of the commands in the DSP's data structure will be interpreted and turned into an activation message for a task or some other action. If a message is to be sent to a task on the DSP, then this message is sent to the OS, which immediately places the activated task



The sequence of operations depicted in Figure 6 is listed below:

- #1a uC writes parameter out
- #1b parameter is written across bridge into DSP's memory
- #2 DSP interrupt is triggered
- #3 ISR sends message to OS to activate required DSP Task
- #4 DSP OS reschedules tasks; highest priority active task starts to run
- #5 parameter is received by DSP Task
- #6a DSP writes parameter out when Task is finished
- #6b parameter is written across bridge into uC's memory
- #7 uC interrupt is triggered
- #8 ISR sends message to OS to indicate required DSP Task has completed

#### 4 Summary

In summary, the methods of communication between processors can be very different depending on the messages that need to be passed, on the latency that is needed on execution, and other system issues. Therefore, the preferred method can only be chosen after analyzing the overall system requirements.

Using a bridge between the DSP's system bus and the  $\mu$ C's system bus allows all peripherals and memories to be visible to both processors. An example of this bridge is the 3DSP AHB Bridge, which interfaces between the DSP-Shuttle bus and the AMBA bus. Various software methods can then be used to pass messages back and forth between the processors.

A simple, small, robust operating system kernel can help in scheduling tasks when it is necessary for the DSP to intelligently organize this. 3DSP supplies the SPEEDi OS, which provides all the standard task switching, prioritization, semaphore and messaging features that are necessary.

---

3DSP is a registered trademark of 3DSP Corporation. DSP-Shuttle, HiFi, iDMA, IP Gallery, IP Jukebox, Sketchpad, Software Studio, SP-3, SP-5, SP-5flex, SPEEDi, SuperSIMD, DSP Done Right!, DSP Solutions for the Connected World, and The DSP Solution Company are trademarks of 3DSP Corporation.

ARM is a registered trademark and AMBA is a trademark of ARM Limited.