Multicore Programming Guide

Multicore Programming and Applications/DSP Systems

Abstract

As application complexity continues to grow, we have reached a limit on increasing performance by merely scaling clock speed. To meet the ever-increasing processing demand, modern System-On-Chip solutions contain multiple processing cores. The dilemma is how to map applications to multicore devices. In this paper, we present a programming methodology for converting applications to run on multicore devices. We also describe the features of Texas Instruments DSPs that enable efficient implementation, execution, synchronization, and analysis of multicore applications.

Contents

1 Introduction

For the past 50 years, Moore's law accurately predicted that the number of transistors on an integrated circuit would double every two years. To translate these transistors into equivalent levels of system performance, chip designers increased clock frequencies (requiring deeper instruction pipelines), increased instruction level parallelism (requiring concurrent threads and branch prediction), increased memory performance (requiring larger caches), and increased power consumption (requiring active power management).

Each of these four areas is hitting a wall that impedes further growth:

- Increased processing frequency is slowing due to diminishing improvements in clock rates and poor wire scaling as semiconductor devices shrink.
- Instruction-level parallelism is limited by the inherent lack of parallelism in the applications.
- Memory performance is limited by the increasing gap between processor and memory speeds.
- Power consumption scales with clock frequency; so, at some point, extraordinary means are needed to cool the device.

Using multiple processor cores on a single chip allows designers to meet performance goals without using the maximum operating frequency. They can select a frequency in the sweet spot of a process technology that results in lower power consumption. Overall performance is achieved with cores having simplified pipeline architectures relative to an equivalent single core solution. Multiple instances of the core in the device result in dramatic increases in the MIPS-per-watt performance.

2 Mapping an Application to a Multicore Processor

Until recently, advances in computing hardware provided significant increases in the execution speed of software with little effort from software developers. The introduction of multicore processors provides a new challenge for software developers, who must now master the programming techniques necessary to fully exploit multicore processing potential.

Task parallelism is the concurrent execution of independent tasks in software. On a single-core processor, separate tasks must share the same processor. On a multicore processor, tasks essentially run independently of one another, resulting in more efficient execution.

2.1 Parallel Processing Models

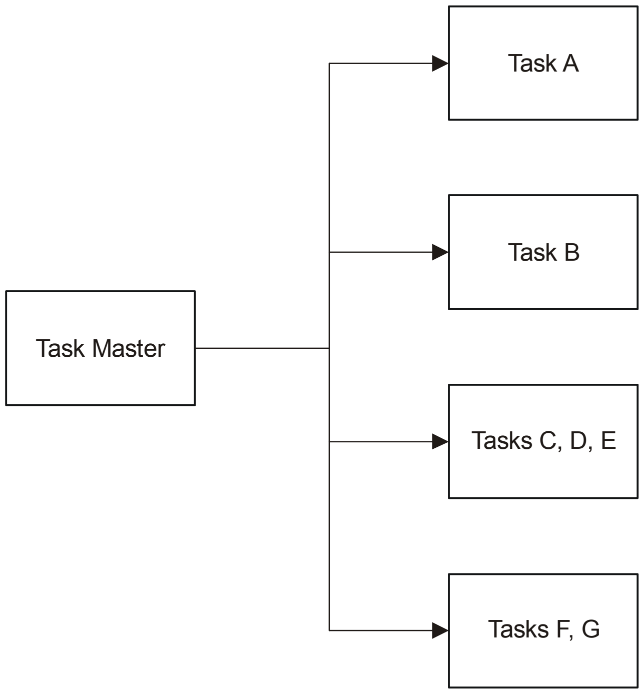
One of the first steps in mapping an application to a multicore processor is to identify the task parallelism and select a processing model that fits best. The two dominant models are a Master/Slave model in which one core controls the work assignments on all cores, and the Data Flow model in which work flows through processing stages as in a pipeline.

2.1.1 Master/Slave Model

The Master/Slave model represents centralized control with distributed execution. A master core is responsible for scheduling various threads of execution that can be allocated to any available core for processing. It also must deliver any data required by the thread to the slave core. Applications that fit this model inherently consist of many small independent threads that fit easily within the processing resources of a single core. This software often contains a significant amount of control code and often accesses memory in random order with multiple levels of indirection. There is relatively little computation per memory access and the code base is usually very large. Applications that fit the Master/Slave model often run on a high-level OS like Linux and potentially already have multiple threads of execution defined. In this scenario, the high-level OS is the master in charge of the scheduling.

The challenge for applications using this model is real-time load balancing because the thread activation can be random. Individual threads of execution can have very different throughput requirements. The master must maintain a list of cores with free resources and be able to optimize the balance of work across the cores so that optimal parallelism is achieved. An example of a Master/Slave task allocation model is shown in Figure 1.

Figure 1 Master / Slave Processing Model



One application that lends itself to the Master/Slave model is the multi-user data link layer of a communication protocol stack. It is responsible for media access control and logical link control of a physical layer including complex, dynamic scheduling and data movement through transport channels. The software often accesses multi-dimensional arrays resulting in very disjointed memory access.

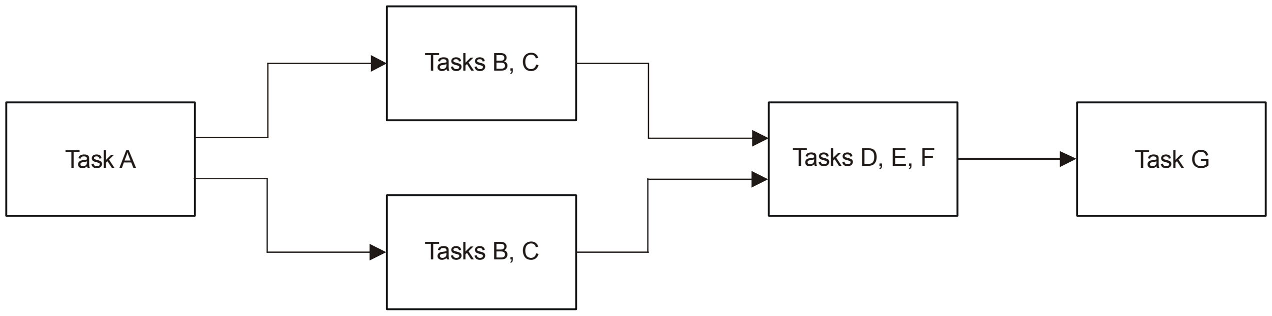
One or more execution threads are mapped to each core. Task assignment is achieved using message-passing between cores. The messages provide the control triggers to begin execution and pointers to the required data. Each core has at least one task whose job is to receive messages containing job assignments. The task is suspended until a message arrives triggering the thread of execution.

2.1.2 Data Flow Model

The Data Flow model represents distributed control and execution. Each core processes a block of data using various algorithms and then the data is passed to another core for further processing. The initial core is often connected to an input interface supplying the initial data for processing from either a sensor or FPGA. Scheduling is triggered upon data availability. Applications that fit the Data Flow model often contain large and computationally complex components that are dependent on each other and may not fit on a single core. They likely run on a realtime OS where minimizing latency is critical. Data access patterns are very regular because each element of the data arrays is processed uniformly.

The challenge for applications using this model is partitioning the complex components across cores and the high data flow rate through the system. Components often need to be split and mapped to multiple cores to keep the processing pipeline flowing regularly. The high data rate requires good memory bandwidth between cores. The data movement between cores is regular and low latency hand-offs are critical. An example of Data Flow processing is shown in Figure 2.

Figure 2 Data Flow Processing Model



One application that lends itself to the Data Flow model is the physical layer of a communication protocol stack. It translates communications requests from the data link layer into hardware-specific operations to affect transmission or reception of electronic signals. The software implements complex signal processing using intrinsic instructions that take advantage of the instruction-level parallelism in the hardware.

The processing chain requires one or more tasks to be mapped to each core. Synchronization of execution is achieved using message passing between cores. Data is passed between cores using shared memory or DMA transfers.

2.1.3 OpenMP Model

OpenMP is an Application Programming Interface (API) for developing multi-threaded applications in C/C++ or Fortran for shared-memory parallel (SMP) architectures.

OpenMP standardizes the last 20 years of SMP practice and is a programmer-friendly approach with many advantages. The API is easy to use and quick to implement; once the programmer identifies parallel regions and inserts the relevant OpenMP constructs, the compiler and runtime system figures out the rest of the details. The API makes it easy to scale across cores and allows moving from an 'm' core implementation to an 'n' core implementation with minimal modifications to source code. OpenMP is sequential-coder friendly; that is, when a programmer has a sequential piece of code and would like to parallelize it, it is not necessary to create a totally separate multicore version of the program. Instead of this all-or-nothing approach, OpenMP encourages an incremental approach to parallelization, where programmers can focus on parallelizing small blocks of code at a time. The API also allows users to maintain a single unified code base for both sequential and parallel versions of code.

2.1.3.1 Features

The OpenMP API consists primarily of compiler directives, library routines, and environment variables that can be leveraged to parallelize a program.

Compiler directives allow programmers to specify which instructions they want to execute in parallel and how they would like the work distributed across cores. OpenMP directives typically have the syntax "#pragma omp construct [clause [clause]…]." For example, "#pragma omp section nowait" where *section* is the construct and *nowait* is a clause. The next section shows example implementations that contain directives.

Library routines or runtime library calls allow programmers to perform a host of different functions. There are execution environment routines that can configure and monitor threads, processors, and other aspects of the parallel environment.

There are lock routines that provide function calls for synchronization. There are timing routines that provide a portable wall clock timer. For example, the library routine "omp_set_num_threads (int numthreads)" tells the compiler how many threads need to be created for an upcoming parallel region.

Finally, environment variables enable programmers to query the state or alter the execution features of an application like the default number of threads, loop iteration count, etc. For example, "OMP_NUM_THREADS" is the environment variable that holds the total number of OpenMP threads.

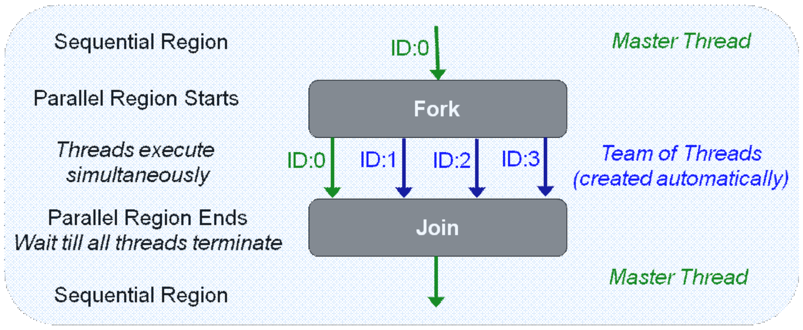
2.1.3.2 Implementation

This section contains four typical implementation scenarios and shows how OpenMP allows programmers to handle each of them. The following examples introduce some important OpenMP compiler directives that are applicable to these implementation scenarios. For a complete list of directives, see the OpenMP specification available on the official OpenMP website at http://www.openmp.org.

Create Teams of Threads

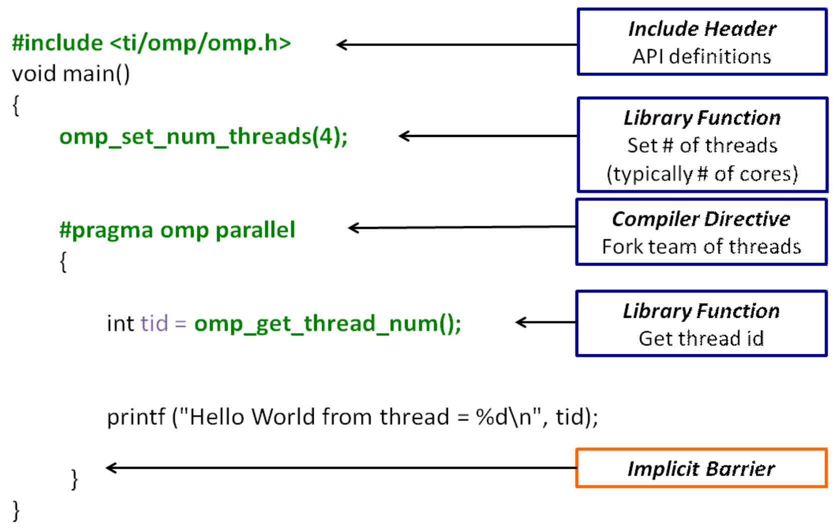
Figure 3 shows how OpenMP implementations are based on a fork-join model. An OpenMP program begins with an initial thread (known as a master thread) in a sequential region. When a parallel region is encountered—indicated by the compiler directive "#pragma omp parallel"—extra threads called worker threads are automatically created by the scheduler. This team of threads executes simultaneously to work on the block of parallel code. When the parallel region ends, the program waits for all threads to terminate, then resumes its single-threaded execution for the next sequential region.

Figure 3 OpenMP Fork-Join Model



To illustrate this point further, it is useful to look at an implementation example. Figure 4 on page 8 shows a sample OpenMP Hello World program. The first line in the code includes the omp.h header file that includes the OpenMP API definitions. Next, the call to the library routine sets the number of threads for the OpenMP parallel region to follow. When the parallel compiler directive is encountered, the scheduler spawns three additional threads. Each of the threads runs the code within the parallel region and prints the Hello World line with its unique thread id. The implicit barrier at the end of the region ensures that all threads terminate before the program continues.

Figure 4 Hello World Example Using OpenMP Parallel Compiler Directive



```
#include <ti/omp/omp.h>
void main()
{
    omp_set_num_threads(4);

    #pragma omp parallel
    {

        int tid = omp_get_thread_num();

        printf ("Hello World from thread = %d\n", tid);

    }
}
```

Share Work Among Threads

After the programmer has identified which blocks of code in the region are to be run by multiple threads, the next step is to express how the work in the parallel region will be shared among the threads. The OpenMP work-sharing constructs are designed to do exactly this. There are a variety of work-sharing constructs available; the following two examples focus on two commonly-used constructs.

The “#pragma omp for” work-sharing construct enables programmers to distribute a *for* loop among multiple threads. This construct applies to *for* loops where subsequent iterations are independent of each other; that is, changing the order in which iterations are called does not change the result.

To appreciate the power of the *for* work-sharing construct, look at the following three situations of implementation: sequential; only with the parallel construct; and both the parallel and work-sharing constructs. Assume a *for* loop with *N* iterations, that does a basic array computation.

Sequential Code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```

Only with Parallel Construct

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i]; }
}
```

Parallel and Work-sharing Constructs

```
#pragma omp parallel
#pragma omp for
    for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```

The second work-sharing construct example is "#pragma omp sections" which allows the programmer to distribute multiple tasks across cores, where each core runs a unique piece of code. The following code snippet illustrates the use of this work-sharing construct.

```
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
        x_calculation();

    #pragma omp section
        y_calculation();

    #pragma omp section
        z_calculation();
}
```

Note that by default a barrier is implicit at the end of the block of code. However, OpenMP makes the *nowait* clause available to turn off the barrier. This would be implemented as "#pragma omp sections nowait".

2.2 Identifying a Parallel Task Implementation

Identifying the task parallelism in an application is a challenge that, for now, must be tackled manually. TI is developing code generation tools that will allow users to instrument their source code to identify opportunities for automating the mapping of tasks to individual cores. Even after identifying parallel tasks, mapping and scheduling the tasks across a multicore system requires careful planning.

A four-step process, derived from *Software Decomposition for Multicore Architectures* [1], is proposed to guide the design of the application:

1. **Partitioning** — Partitioning of a design is intended to expose opportunities for parallel execution. The focus is on defining a large number of small tasks in order to yield a fine-grained decomposition of a problem.
2. **Communication** — The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks to allow computation to proceed. This information flow is specified in the communication phase of a design.
3. **Combining** — Decisions made in the partitioning and communication phases are reviewed to identify a grouping that will execute efficiently on the multicore architecture.
4. **Mapping** — This stage consists of determining where each task is to execute.

2.2.1 Partitioning

Partitioning an application into base components requires a complexity analysis of the computation (Reads, Writes, Executes, Multiplies) in each software component and an analysis of the coupling and cohesion of each component.

For an existing application, the easiest way to measure the computational requirements is to instrument the software to collect timestamps at the entry and exit of each module of interest. Using the execution schedule, it is then possible to calculate the throughput rate requirements in MIPS. Measurements should be collected with both cold and warm caches to understand the overhead of instruction and data cache misses.

Estimating the coupling of a component characterizes its interdependence with other subsystems. An analysis of the number of functions or global data outside the subsystem that depend on entities within the subsystem can pinpoint too many responsibilities to other systems. An analysis of the number of functions inside the subsystem that depend on functions or global data outside the subsystem identifies the level of dependency on other systems.

A subsystem's cohesion characterizes its internal interdependencies and the degree to which the various responsibilities of the module are focused. It expresses how well all the internal functions of the subsystem work together. If a single algorithm must use every function in a subsystem, then there is high cohesion. If several algorithms each use only a few functions in a subsystem, then there is low cohesion. Subsystems with high cohesion tend to be very modular, supporting partitioning more easily.

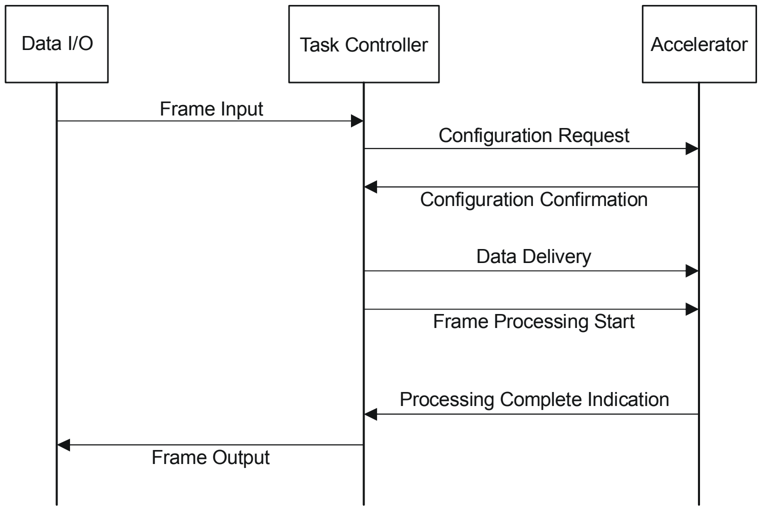
Partitioning the application into modules or subsystems is a matter of finding the breakpoints where coupling is low and cohesion is high. If a module has too many external dependencies, it should be grouped with another module that together would reduce coupling and increase cohesion. It is also necessary to take into account the overall throughput requirements of the module to ensure it fits within a single core.

2.2.2 Communication

After the software modules are identified in the partitioning stage it is necessary to measure the control and data communication requirements between them. Control flow diagrams can identify independent control paths that help determine concurrent tasks in the system. Data flow diagrams help determine object and data synchronization needs.

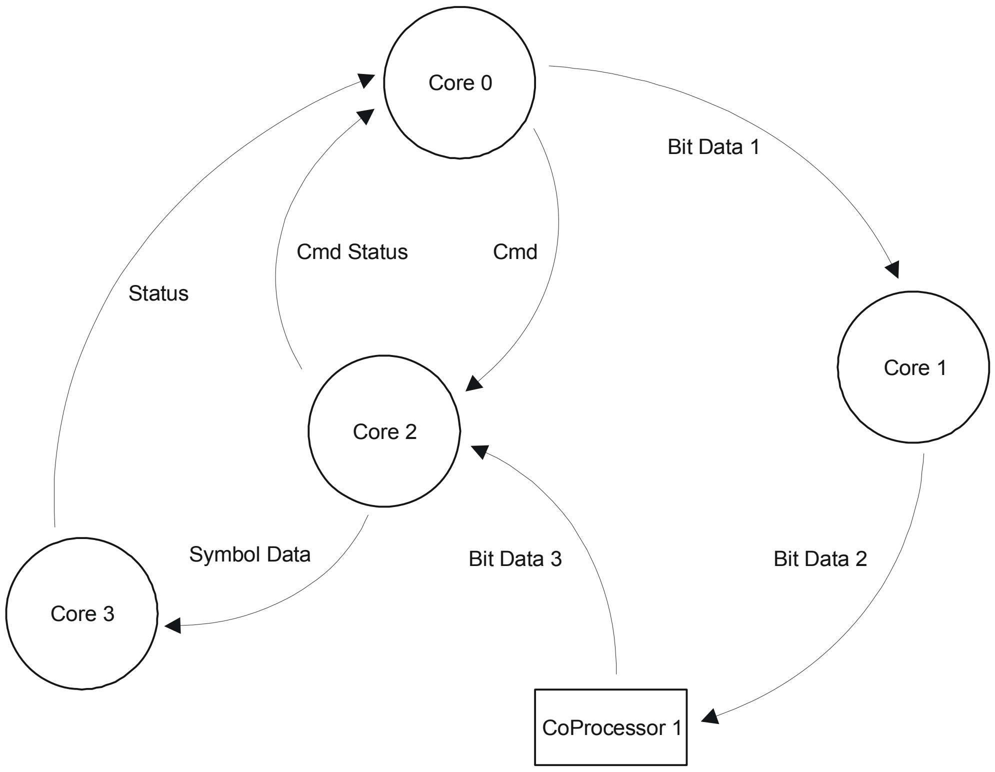
Control flow diagrams represent the execution paths between modules. Modules in a processing sequence that are not on the same core must rely on message passing to synchronize their execution and possibly require data transfers. Both of these actions can introduce latency. The control flow diagrams should be used to create metrics that assist the module grouping decision to maximize overall throughput. Figure 5 shows an example of a control flow diagram.

Figure 5 Example Control Flow Diagram



Data flow diagrams identify the data that must pass between modules and this can be used to create a measure of the amount and rate of data passed. A data flow diagram also shows the level of interaction between a module and outside entities. Metrics should be created to assist the grouping of modules to minimize the number and amount of data communicated between cores. Figure 6 shows an example diagram.

Figure 6 Example Data Flow Diagram



2.2.3 Combining

The combining phase determines whether it is useful to combine tasks identified by the partitioning phase, so as to provide a smaller number of tasks, each of greater size. Combining also includes determining whether it is worthwhile to replicate data or computation. Related modules with low computational requirements and high coupling are grouped together. Modules with high computation and high communication costs are decomposed into smaller modules with lower individual costs.

2.2.4 Mapping

Mapping is the process of assigning modules, tasks, or subsystems to individual cores. Using the results from Partitioning, Communication, and Combining, a plan is made identifying concurrency issues and module coupling. This is also the time to consider available hardware accelerators and any dependencies this would place on software modules.

Subsystems are allocated onto different cores based on the selected programming model: Master/Slave or Data Flow. To allow for inter-processor communication latency and parametric scaling, it is important to reserve some of the available MIPS, L2 memory, and communication bandwidth on the first iteration of mapping. After all the modules are mapped, the overall loading of each core can be evaluated to indicate areas for additional refactoring to balance the processing load across cores.

In addition to the throughput requirements of each module, message passing latency and processing synchronization must be factored into the overall timeline. Critical latency issues can be addressed by adjusting the module factoring to reduce the overall number of communication steps. When multiple cores need to share a resource like a DMA engine or critical memory section, a hardware semaphore is used to ensure mutual exclusion as described in Section 5.3. Blocking time for a resource must be factored into the overall processing efficiency equation.

Embedded processors typically have a memory hierarchy with multiple levels of cache and off-chip memory. It is preferred to operate on data in cache to minimize the performance hit on the external memory interface. The processing partition selected may require additional memory buffers or data duplication to compensate for inter-processor-communication latency. Refactoring the software modules to optimize the cache performance is an important consideration.

When a particular algorithm or critical processing loop requires more throughput than available on a single core, consider the data parallelism as a potential way to split the processing requirements. A brute force division of the data by the available number of cores is not always the best split due to data locality and organization, and required signal processing. Carefully evaluate the amount of data that must be shared between cores to determine the best split and any need to duplicate some portion of the data.

The use of hardware accelerators like FFT or Viterbi coprocessors is common in embedded processing. Sharing the accelerator across multiple cores would require mutual exclusion via a lock to ensure correct behavior. Partitioning all functionality requiring the use of the coprocessor to a single core eliminates the need for a hardware semaphore and the associated latency. Developers should study the efficiency of blocking multicore access to the accelerator versus non-blocking single core access with potentially additional data transfer costs to get the data to the single core.

Consideration must be given to scalability as part of the partitioning process. Critical system parameters are identified and their likely instantiations and combinations mapped to important use cases. The mapping of tasks to cores would ideally remain fixed as the application scales for the various use cases.

The mapping process requires multiple cycles of task allocation and parallel efficiency measurement to find an optimal solution. There is no heuristic that is optimal for all applications.

2.2.5 Identifying and Modifying the Code for OpenMP-based Parallelization

OpenMP provides some very useful APIs for parallelization, but it is the programmer's responsibility to identify a parallelization strategy, then leverage relevant OpenMP APIs. Deciding what code snippets to parallelize depends on the application code and the use-case. The 'omp parallel' construct, introduced earlier in this section, can essentially be used to parallelize any redundant function across cores. If the sequential code contains 'for' loops with a large number of iterations, the programmer can leverage the 'omp for' OpenMP construct that splits the 'for' loop iterations across cores.

Another question the programmer should consider here is whether the application lends itself to data-based or task-based partitioning. For example, splitting an image into 8 slices, where each core receives one input slice and runs the same set of algorithms on the slice, is an example of data-based partitioning, which could lend itself to the 'omp parallel' and 'omp for' constructs. In contrast, if each core is running a different algorithm, the programmer could leverage the 'omp sections' construct to split unique tasks across cores.

3 Inter-Processor Communication

The Texas Instruments KeyStone family of devices TCI66xx and C66xx, as well as the older TCI64xx and C64xx multicore devices, offer several architectural mechanisms to support inter-processor communication. All cores have full access to the device memory map; this means that any core can read from and write to any memory. In addition, there is support for direct event signaling between cores for notification as well as DMA event control for third-party notification. The signaling available is flexible to allow the solution to be tailored to the communication desired. Last, there are hardware elements to allow for atomic access arbitration, which can be used to communicate ownership of a shared resource. The Multicore Navigator module, available in most KeyStone devices, provides an efficient way to synchronize cores, communicate and transfer data between cores, and easy access to some of the high-bit-rate peripherals and coprocessors with minimal involvement of the cores.

Inter-core communication consists of two primary actions: data movement and notification (including synchronization).

3.1 Data Movement

The physical movement of data can be accomplished by several different techniques:

- **Use of a shared message buffer** — The sender and receiver have access to the same physical memory.
- **Use of dedicated memories** — There is a transfer between dedicated send and receive buffers.
- **Transitioned memory buffer** — The *ownership* of a memory buffer is given from sender to receiver, but the contents do not transfer.

For each technique, there are two means to read and write the memory contents: CPU load/store and DMA transfer. Each transfer can be configured to use a different method.

3.1.1 Shared Memory

Using a shared memory buffer does not necessarily mean that an equally-shared memory is used, though this would be typical. Rather, it means that a message buffer is set up in a memory that is accessible by both sender and receiver, with each responsible for its portion of the transaction. The sender sends the message to the shared buffer and notifies the receiver. The receiver retrieves the message by copying the contents from a source buffer to a destination buffer and notifies the sender that the buffer is free. It is important to maintain coherency when multiple cores access data from shared memory.

The SYS/BIOS message queue transport, developed for TCI64x and C64xx multicore devices to send messages between cores, as well as IPC software layer developed for the KeyStone family to send messages and synchronize between cores may use this scheme.

3.1.2 Dedicated Memories

It is also possible to manage the transport between the sending and receiving memories. This is typically used when the cores are using dedicated area of the shared memory for each core or local memory for their data where overhead is reduced by keeping the data local. The data movement can be done by direct communication between cores or with the Multicore Navigator specifically within the KeyStone family of devices. First we describe direct communication between cores. As with the shared memory, there are notification and transfer stages, and this can be accomplished through a push or pull mechanism, depending on the use case.

In a push model, the sender is responsible to fill the receive buffer; in a pull model, the receiver is responsible to retrieve the data from the send buffer (Table 1). There are advantages and disadvantages to both. Primarily, it affects the point of synchronization.

Table 1 Dedicated Memory Models

Push Model	Pull Model
Sender prepares send buffer	Sender prepares send buffer
Sender transfers to receive buffer	Receiver is notified of data ready
Receiver is notified of data ready	Receiver transfers to receive buffer
Receiver consumes data	Receiver frees memory
Receiver frees memory	Receiver consumes data

The differences are only in the notifications. Typically the push model is used due to the overhead of a remote read request used in the pull model. However, if resources are tight on the receiver, it may be advantageous for the receiver to control the data transfer to allow tighter management of its memory.

Using the Multicore Navigator reduces the work that the cores have to do during realtime processing. The Multicore Navigator model for transporting data between dedicated memories is as follows:

1. Sender uses a predefined structure called a descriptor to either pass data directly or point to a data buffer to send. This is determined by the descriptor structure type.
2. The Sender pushes the descriptor to a hardware queue associated with the receiver.
3. The data is available to the receiver.

To notify the receiver that the data is available, the Multicore Navigator provides multiple methods of notification. These methods are described in the notification section of this document.

3.1.3 Transitioned Memory

It is also possible for the sender and receiver to use the same physical memory, but unlike the shared memory transfer described above, common memory is not temporary. Rather, the buffer ownership is transferred, but the data does not move through a message path. The sender passes a pointer to the receiver and the receiver uses the contents from the original memory buffer.

Message sequence:

1. Sender generates data into memory.
2. Sender notifies receiver of data ready/ownership given.
3. Receiver consumes memory directly.
4. Receiver notifies sender of data ready/ownership given.

If applicable for symmetric flow of data, the receiver may switch to the sender role prior to returning ownership and use the same buffer for its message.

3.1.4 Data Movement in OpenMP

Programmers can manage data scoping by using clauses such as *private*, *shared*, and *default* in their OpenMP compiler directives. As mentioned previously, OpenMP compiler directives take the form “#pragma omp construct [clause [clause]…].” The data scoping clauses are followed by a list of variables in curved brackets. For example, “#pragma omp parallel private(i,j).”

When variables are qualified by a *private* clause, each thread has a private copy of the variable and a unique value of the variable throughout the parallel construct. These variables are stored in the thread's stack, the default size of which is set by the compiler, but can be overridden.

When variables are qualified by a *shared* clause, the same copy of the variable is seen by all threads. These are typically stored in shared memory like DDR or MSMC.

By default, OpenMP manages data scoping of some variables. Variables that are declared outside a parallel region are automatically scoped as *shared*. Variables that are declared inside a parallel region are automatically scoped as *private*. Other default scenarios also exist; for example, iteration counts are automatically enforced by the compiler as *private* variables.

The *default* clause enables programmers to override the default scope assigned to any variable. For example, *default none* can be used to state that no variables declared inside or outside the parallel region are implied to be *private* or *shared*, respectively, and it is the programmer's task to explicitly specify the scope of all variables inside the parallel region.

The example block of code below shows these data scoping clauses:

```
#pragma omp parallel for default (none) private( i, j, sum )
shared (A, B, C)
{
    for (i = 0, i < 10; i++) {
        sum = 0;
        for ( j = 0; j < 20; j++ )
                sum += B[ i ][ j ] * C [ j ];
        A[ i ] = sum;
    }
}
```

3.2 Multicore Navigator Data Movement

The Multicore Navigator encapsulates messages—including messages that contain data in containers called descriptors—and moves them between hardware queues. The Queue Manager Subsystem (QMSS) is the central part of the Multicore Navigator that controls the behavior of the hardware queues and enables routing of descriptors. Multiple instances of logic-based DMA called PKTDMA moves descriptors between queues and to and from peripherals as will be discussed later. A special instance of PKTDMA called Infrastructure PKTDMA resides inside the QMSS and facilitates moving data between threads that belong to different cores. When a core wants to move data to another core, it puts the data in a buffer that is associated with a descriptor and pushes the descriptor to a queue. All the routing and monitoring is done inside the QMSS. The descriptor is pushed into a queue that belongs to the receive core. Different methods of notifying the receive core that a descriptor with data is available to it are described in the Notification chapter.

Moving data between cores using the Multicore Navigator queues enables the sending core to "fire and forget" the data movement and offloads the cores from copying the data. It enables a loose link between cores so that the send core is not blocked by the receive core.

3.3 Notification and Synchronization

The multicore model requires the ability to synchronize cores and to send notifications between cores. A typical synchronization use case is when a single core does all the system initialization and all other cores must wait until initialization is complete before continuing execution. Fork and joint points in parallel processing require synchronization between (conceivably a subset of) the cores. Synchronization and notification can be implemented using the Multicore Navigator or by CPU execution. Transport data from one core to another requires notifications. As previously mentioned, the Multicore Navigator offers a variety of methods to notify the receive core that data is available. The notification methods are described in "Multicore Navigator Notification Methods" on page 22.

For non-navigator data transport, after communication message data is prepared by the sender for delivery to the receiver using shared, dedicated, or transitional memory, it is necessary to notify the receiver of the message availability. This can be accomplished by direct or indirect signaling, or by atomic arbitration.

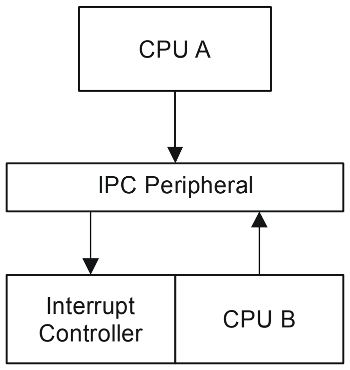
3.3.1 Direct Signaling

The devices support a simple peripheral that allows a core to generate a physical event to another core. This event is routed through the core's local interrupt controller along with all other system events. The programmer can select whether this event will generate a CPU interrupt or if the CPU will poll its status. The peripheral includes a flag register to indicate the originator of the event so that the notified CPU can take the appropriate action (including clearing the flag) as shown in Figure 7.

The processing steps are:

1. CPU A writes to CPU B's inter-processor communication (IPC) control register
2. IPC event generated to interrupt controller
3. Interrupt controller notifies CPU B (or polls)
4. CPU B queries IPC
5. CPU B clears IPC flag(s)
6. CPU B performs appropriate action

Figure 7 Direct IPC Signaling



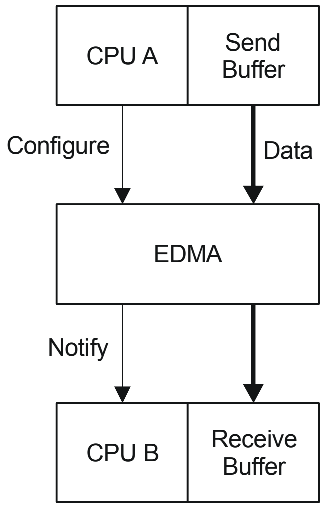
3.3.2 Indirect Signaling

If a third-party transport, such as the EDMA controller, is used to move data, the signaling between cores can also be performed through this transport. In other words, the notification follows the data movement in hardware, rather than through software control, as shown in Figure 8.

The processing steps are:

1. CPU A configures and triggers transfer using EDMA
2. EDMA completion event generated to interrupt controller
3. Interrupt controller notifies CPU B (or polls)

Figure 8 Indirect Signaling



3.3.3 Atomic Arbitration

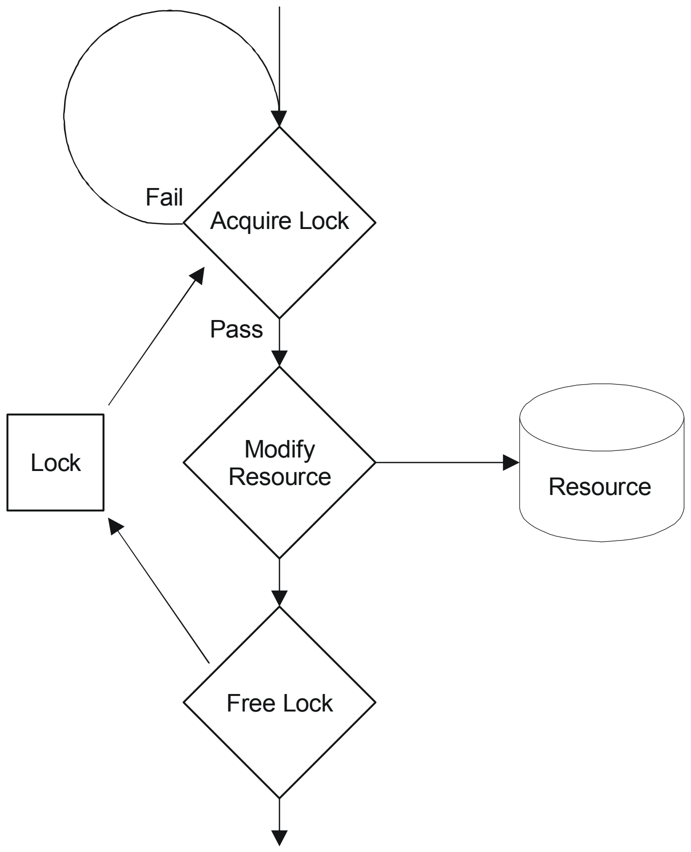
Each device includes hardware support for atomic arbitration. The supporting architecture varies on different devices, but the same underlying function can be achieved easily. Atomic arbitration instructions are supported with hardware monitors in the Shared L2 controller on the TCI6486 and C6472 devices, while a semaphore peripheral is on the TCI6487/88 and C6474 devices because they do not have a shared L2 memory. The KeyStone family of devices has both atomic arbitration instructions and a semaphore peripheral. On all devices, a CPU can atomically acquire a lock, modify any shared resource, and release the lock back to the system.

The hardware guarantees that the acquisition of the lock itself is atomic, meaning only a single core can own it at any time. There is no hardware guarantee that the shared resource(s) associated with the lock are protected. Rather, the lock is a hardware tool that allows software to guarantee atomicity through a well-defined (and simple) protocol outlined in Table 2 and shown in Figure 9.

Table 2 Atomic Arbitration Protocol

CPU A	CPU B
1: Acquire lock	**1:** Acquire lock
→ Pass (lock available)	→ Fail (Fails because lock is unavailable)
2: Modify resource	**2:** Repeat step 1 until Pass
3: Release lock	→ Pass (lock available)
	3: Modify resource
	4: Release lock

Figure 9 Atomic Arbitration



3.3.4 Synchronization in OpenMP

With OpenMP, synchronizations are either implicit or can be explicitly defined using compiler directives.

Thread synchronizations are implicit at the end of parallel or work-sharing constructs. This means that no thread can progress until all other threads in the team have reached the end of the block of code.

Synchronization directives can also be defined explicitly. For example, the *critical* construct ensures that only one thread can enter the block of code at a time. It is important to include a unique region name as "#pragma omp critical <region name>." If critical sections are unnamed, threads will not enter any of the critical regions. Another example of an explicit synchronization directive is the *atomic* directive. There are some key differences between the *atomic* and *critical* directives: *atomic* applies only to a line of code, which is translated into a hardware-based atomic operation, and is therefore more hardware-dependent and less-portable than the *critical* construct, which applies to a block of code.

3.4 Multicore Navigator Notification Methods

The Multicore Navigator encapsulates messages, including messages that contain data in containers called descriptors, and moves them between hardware queues. Each destination has one or more dedicated receive queues. The Multicore Navigator enables the following methods for the receiver to access the descriptors in a receive queue.

3.4.1 Non-blocking Polling

In this method, the receiver checks to see if there is a descriptor waiting for it in the receive queue. If there is no descriptor, the receiver continues its execution.

3.4.2 Blocking Polling

In this method, the receiver blocks its execution until there is a descriptor in the receive queue, then it continues to process the descriptor.

3.4.3 Interrupt-based Notification

In this method, the receiver gets an interrupt whenever a new descriptor is put into its receive queue. This method guarantees fast response to incoming descriptors. When a new descriptor arrives, the receiver performs context switching and starts processing the new descriptor.

3.4.4 Delayed (Staggered) Interrupt Notification

When the frequency of incoming descriptors is high, the navigator can configure the interrupt to be sent only when the number of new descriptors in the queue reaches a programmable watermark, or after a certain time from the arrival of the first descriptor in the queue. This method reduces the context switching load of the receiver.

3.4.5 QoS-based Notification

A quality-of-service mechanism is supported by the Multicore Navigator to prioritize the data stream traffic of the peripheral modules; this mechanism evaluates each data stream with a view to delaying or expediting the data stream according to predefined quality-of-service parameters. The same mechanism can be used to transfer messages of different importance between cores.

The quality-of-service (QoS) firmware has the job of policing all packet flows in the system and verifying that neither the peripherals nor the host CPU are overwhelmed with packets. To support QoS, a special processor called the QoS PDSP monitors and moves descriptors between queues.

The key to the functionality of the QoS system is the arrangement of packet queues. There are two sets of packet queues: the QoS ingress queues and the final destination queues. The final destination queues are further divided into host queues and peripheral egress queues. Host queues are those that terminate on the host device and are actually received by the host. Egress queues are those that terminate at a physical egress peripheral device. When shaping traffic, only the QoS PDSP writes to either the host queues or the egress queues. Unshaped traffic is written only to QoS ingress queues.

It is the job of the QoS PDSP to move packets from the QoS ingress queues to their final destination queues while performing the proper traffic shaping in the process. There is a designated set of queues in the system that feed into the QoS PDSP. These are called QoS queues. The QoS queues are simply queues that are controlled by the firmware running on the PDSP. There are no inherent properties of the queues that fix them to a specific purpose.

4 Data Transfer Engines

Within the device, the primary data transfer engines on current Texas Instruments KeyStone TCI66xx and C66xx devices are the EDMA (enhanced DMA) modules and the PKTDMA (Packet DMA, part of the Multicore Navigator) modules. For high-bit communication between devices, there are several transfer engines, depending on the physical interface selected for communication. Some of the transfer engines have an instance of PKTDMA to move data in and out from the peripheral engine. High bit-rate peripherals include:

- **Antenna Interface (Wireless devices):** Multiple PKTDMA instances are used in conjunction with multiple AIF instances to transport data.
- **Serial RapidIO:** There are two modes available — DirectIO and Messaging. Depending on the mode, the PKTDMA or built-in direct DMA control are available.
- **Ethernet:** There is a PKTDMA instance for handling all of the data movement.
- **PCI express**: There is build-in DMA control that moves data in and out of the PCI express into dedicated memory
- **HyperLink**: The KeyStone family has a proprietary point to point fast bus that enables direct linking of two devices together. There is a build-in DMA control that moves data in and out of the HyperLink module into dedicated memory.

In addition, PKTDMA is used to move data between the cores and high-bit-rate coprocessors such as the FFT engines on the wireless devices of the family.

4.1 Packet DMA

Packet DMA (PKTDMA) instances are part of the Multicore Navigator. Each PKTDMA instance has a separate hardware path for receive and transmit data with multiple DMA channels in each direction. For transmit data, PKTDMA converts data encapsulated in descriptors into a bit stream. Receive bit-stream data is encapsulated into descriptors and is routed to a predefined destination.

The other part of the Multicore Navigator is the Queue Manager Subsystem (QMSS). Currently, the Multicore Navigator has 8192 hardware queues and can support up to 512K descriptors. It includes a queue manager, multiple processors (called PDSP), and an interrupt manager unit. The queue manager controls the queues while the PKTDMA moves descriptors between queues. The notification methods that are described above are controlled by the queue manager special PDSP processors. The queue manager is responsible for routing descriptors to the correct destination.

Some peripherals and coprocessors that may require routing of data to different cores or different destinations have an instance of PKTDMA as part of the peripheral or the coprocessor. In addition, a special PKTDMA instance called an infrastructure PKTDMA resides in the QMSS to support communications between cores.

4.2 EDMA

Channels and parameter RAM can be separated by software into regions with each region assigned to a core. The event-to-channel routing and EDMA interrupts are fully programmable, allowing flexibility as to ownership. All event, interrupt, and channel parameter control is designed to be controlled independently, meaning that once allocated to a core, that core does not need to arbitrate before accessing the resource. In addition, a sophisticated mechanism ensures that EMDA transfer initiated by a certain core will keep the same memory access attributes of the originated core in terms of address translation and privileges. For more information, see "Shared Resource Management" on page 26.

4.3 Ethernet

The Network Coprocessor (NetCP) peripheral supports Ethernet communication. It has two SGMII ports (10/100/1000) and one internal port. A special packet accelerator module supports routing based on L2 address values (up to 64 different addresses), L3 address values (up to 64 different addresses), L4 address values (up to 8192 addresses) or any combination of L2, L3, and L4 addressing. In addition, the packet accelerator can calculate CRC and other error detection values to help incoming and outgoing packets. A special security engine can do decryption and encryption of packets to support VPN or other applications that require security.

An instance of PKTDMA is part of the NetCP and it manages all traffic into, out of, and inside the NetCP and enables routing of packets to a predefined destination.

4.4 RapidIO

Both DirectIO and messaging protocols allow for orthogonal control by each of the cores. For DSP-initiated DirectIO transfers, the load-store units (LSUs) are used. There are multiple LSUs (depending on the device), each independent from the others, and each can submit transactions to any physical link. The LSUs may be allocated to individual cores, after which the cores need not arbitrate for access. Alternatively, the LSUs can be allocated as needed to any core, in which case there would need to be a temporary ownership assigned that may be done using a semaphore resource. Similar to the Ethernet peripheral, messaging allows for individual control of multiple transfer channels. When using messaging protocols, a special instance of PKTDMA is responsible for routing incoming packets to a destination core based on destination ID, mail-box and letter values, and to route outbound messages from cores to the external world. After each core configures the Multicore Navigator parameters for its own messaging traffic, the data movement is done by the Multicore Navigator and is transparent to the user.

4.5 Antenna Interface

The AIF2 antenna interface supports many wireless standards such as WCDMA, LTE, WiMAX, TD-SCDMA, and GSM/EDGE. AIF2 can be accessed in direct mode using its own DMA module, or packet-based access using PKTDMA instance that is part of each AIF2 instance.

When direct IO is used, it is the responsibility of the cores to manage the ingress and egress traffic explicitly. In many cases, egress antenna data comes from the FFT engine (FFTC) and ingress antenna data goes to the FFTC. Using the PKTDMA and the Multicore Navigator system can facilitate the data movement between the AIF and FFTC without the involvement of any core.

Each of the FFTC engines has its own PKTDMA instance. The Multicore Navigator can be configured to send incoming antenna data directly into the correct FFTC engine for processing; from there, the data will be routed to continue processing.

128 queues of the queue manager subsystem are dedicated to the AIF2. When a descriptor enters into one of these queues, a pending signal is sent to the appropriate PKTDMA of the AIF instance that is associated with the queue, and the data is read and sent out via the AIF2 interface. Similarly, data arriving at the AIF is encapsulated by the PKTDMA into descriptors and, based on pre-configuration, the descriptor is routed to the destination, usually, an FFTC instance for FFT processing.

4.6 PCI Express

The PCI express engine in the KeyStone TCI66XX and C66XX devices supports three modes of operation, Root complex, endpoint, and legacy endpoint. The PCI express peripheral uses a built-in DMA control to move data to and from the external world directly into internal or external memory locations.

4.7 HyperLink

The HyperLink peripheral in the KeyStone TCI66XX and C66XX devices enables one device to read and write to and from the other device memory via the HyperLink. In addition, the Hyperlink enables sending events and interrupts to the other side of the HyperLink connection. The HyperLink peripheral uses a built-in DMA control to read and write data to and from the memory to the interface.

5 Shared Resource Management

When sharing resources on the device, it is critical that there is a uniform protocol followed by all the cores in the system. The protocol may depend on the set of resources being shared, but all cores must follow the same rules.

Section 3.3 describes signaling in the context of message passing. The same signalling mechanisms can also be used for general resource management. Direct signaling or atomic arbitration can be used between cores. Within a core, a global flag or an OS semaphore can be used. It is not recommended to use a simple global flag for intercore arbitration because there is significant overhead to ensure updates are atomic.

5.1 Global Flags

Global flags are useful within a single core using a single-threaded model. If there is a resource that depends on an action being completed (typically a hardware event), a global flag may be set and cleared for simple control. While global flags that are based on software structure can be used in multicore environment, it is not recommended. The overhead needed to ensure proper operation across multiple cores (preventing race conditions, ensuring that all cores see a global flag, managing state change over multiple cores) is too high and other methods such as using the IPC registers or semaphores are more efficient.

5.2 OS Semaphores

All multitask operating systems include semaphore support for arbitration of shared resources and for task synchronization. On a single core, this is essentially a global flag controlled by the OS that keeps track of when a resource is owned by a task or when a thread should block or proceed with execution based on signals the semaphore has received.

5.3 Hardware Semaphores

Hardware semaphores are needed only when arbitrating between cores. There is no advantage to using them for single-core arbitration; the OS can use its own mechanism with much less overhead. When arbitrating between cores, hardware support is essential to ensure updates are atomic. There are software algorithms that can be used along with shared memory, but these consume CPU cycles unnecessarily.

5.4 Direct Signaling

As with message passing, direct signaling can be used for simple arbitration. If there is only a small set of resources being shared between cores, the IPC signaling described in Section 3.3.1 can be used. A protocol can be followed to allow a notify-and-acknowledge handshake to pass ownership of a resource. The KeyStone TCI66XX and C66XX devices have a set of hardware registers that can be used to facilitate efficiently core-to-core interrupts, event/signaling and host-to-core interrupts, and events generation and acknowledgements.

6 Memory Management

In programming a multicore device, it is important to consider the processing model. On the Texas Instruments TCI66xx and C6xx devices, each core has local L1/L2 memory and equal access to any shared internal and external memory. It is typically expected that each core will execute some or the entire code image from shared memory, with data being the predominant use of the local memories. This is not a restriction on the user and is described later in this section.

In the case of each core having its own code and data space, the aliased local L1/L2 addresses should not be used. Only the global addresses should be used, which gives a common view to the entire system of each memory location. This also means that for software development, each core would have its own project, built in isolation from the others. Shared regions would be commonly defined in each core's map and accessed directly by any master using the same address.

In the case of there being a shared code section, there may be a desire to use aliased addresses for data structures or scratch memory used in the common function(s). This would allow the same address to be used by any of the cores without regard for checking which core it is. The data structure/scratch buffer would need to have a run address defined using the aliased address region so that when accessed by the function it is core-agnostic. The load address would need to be the global address for the same offset. The runtime, aliased address is usable for direct CPU load/store and internal DMA (IDMA) paging, though not EDMA, PKTDMA, or other master transactions. These transactions must use the global address.

It is always possible for the software to verify on which core it is running, so the aliased addresses are not required to be used in common code. There is a CPU register (DNUM) that holds the DSP core number and can be read during runtime to conditionally execute code and update pointers.

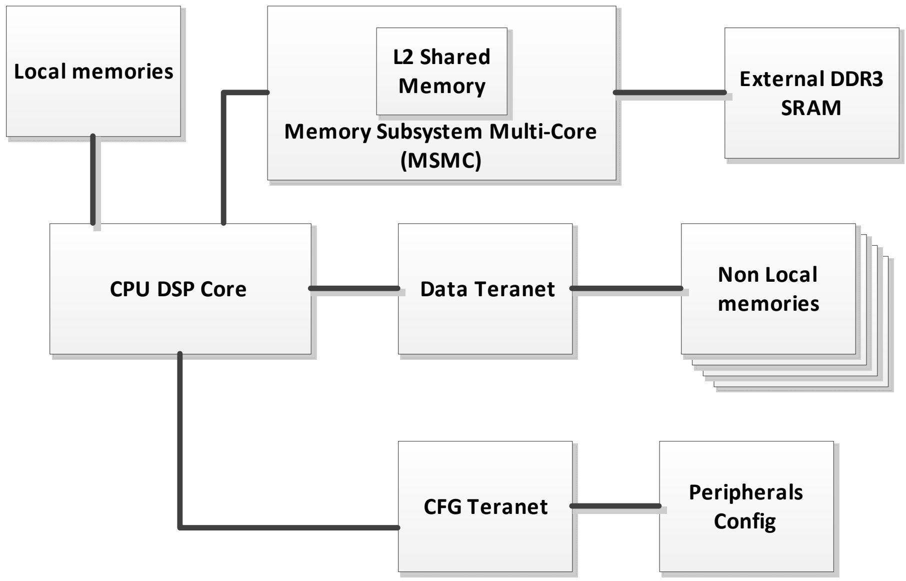
Any shared data resource should be arbitrated so that there are no conflicts of ownership. There is an on-chip semaphore peripheral that allows threads executing on different CPUs to arbitrate for ownership of a shared resource. This ensures that a read-modify-write update to a shared resource can be made atomically.

To speed up reading program and data from external DDR3 memory and from the shared L2 memory, each core has a set of dedicated prefetch registers. These prefetch registers are used to pre-load consecutive memory from the external memory (or the shared L2 memory) before it is needed by the core. The prefetch mechanism assesses the direction from which data and program are read from external memory, and pre-load data and program that may be read in the future, resulting in higher bandwidth if the pre-load data is needed, or with un-needed reading from external memory if the read data is not read later. Each core can control the prefetch as well as the cache for each memory segment (16MB) separately.

6.1 CPU View of the Device

Each of the CPUs has an identical view of the device. As shown in Figure 10, beyond each core's L2 memory there is a switched central resource (SCR) that interconnects the cores, external memory interface, and on-chip peripherals through a switch fabric.

Figure 10 CPUs' Device View



Each of the cores is a master to both the configuration (access to peripheral control registers) and DMA (internal and external data memories) switch fabrics. In addition, each core has a slave interface to the DMA switch fabric allowing access to its L1 and L2 SRAM. All cores have equal access to all slave endpoints with priority assigned per master by user software for arbitration between all accesses at each endpoint.

Each slave in the system (e.g. Timer control, DDR3 SDRAM, each core's L1/L2 SRAM) has a unique address in the device's memory map that is used by any of the masters to access it. Restrictions to the chip-level routing is beyond the scope of the document, but for the most part, each core has access to all control registers and all RAM locations in the memory map. For details of restrictions to chip-level routing, see TI reference guide SPRUGW0, *TMS320C66x DSP CorePac User Guide*[3].

Within each core there are Level 1 program and data memories directly connected to the CPU, and a Level 2 unified memory. Details for the cache and SRAM control (see [3]) are beyond the scope of this document, but each memory is user-configurable to have some portion be memory-mapped SRAM.

As described previously, the local core's L1/L2 memories have two entries in the memory map. All memory local to the processors has global addresses that are accessible to all masters in the device. In addition, local memory can be accessed directly by the associated processor through aliased addresses, where the eight most significant bits are masked to zero. The aliasing is handled within the core and allows for common code to be run unmodified on multiple cores. For example, address location 0x10800000 is the global base address for core 0's L2 memory. Core 0 can access this location by using either 0x10800000 or 0x00800000. Any other master on the device must use 0x10800000 only. Conversely, 0x00800000 can be used by any of the cores as their own L2 base addresses. For Core 0, as mentioned, this is equivalent to 0x10800000, for Core 1 this is equivalent to 0x11800000, and for Core 2 this is equivalent to 0x12800000 and so on for all cores in the device. Local addresses should be used only for shared code or data, allowing a single image to be included in memory. Any code/data targeted to a specific core, or a memory region allocated during runtime by a particular core, should always use the global address only.

Each core accesses any of the shared memories either L2 shared memory (MSM - multicore shared memory) or the external memory DDR3 via the memory subsystem multicore (MSMC) module. Each core has a direct master port into the MSMC. The MSMC arbitrate and optimize access to shared memory from all masters, including each core, EDMA access or other masters and it performs error detection and correction. The XMC (external memory controller) registers and the EMC (enhanced memory controller) registers manage the MSMC interface individually for each core and provide memory protection and address translation from 32 bits to 36 bits to enable various address manipulations such as accessing up to 8 GB of external memory.

6.2 Cache and Prefetch Considerations

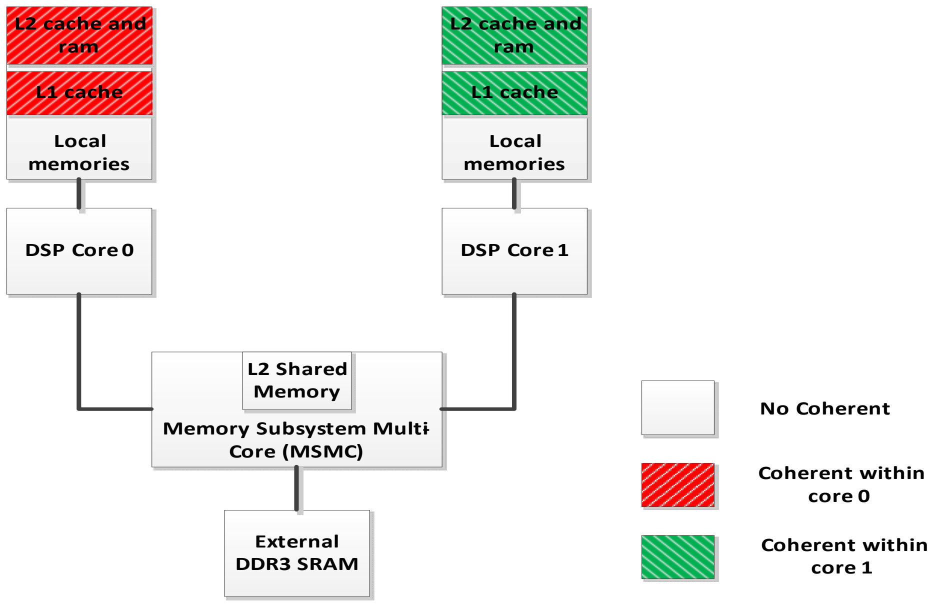
It is important to point out that the only coherency guaranteed by hardware with no software management is L1D cache coherency with L2 SRAM within the same core. The hardware will guarantee that any updates to L2 will be reflected in L1D cache, and vice versa. There is no guaranteed coherency between L1P cache and L2 within the same core, there is no coherency between L1/L2 on one core and L1/L2 on another core, and there is no coherency between any L1/L2 on the chip and shared L2 memory and external memory.

The TCI66xx and C66xx devices do not support automated cache coherency because of the power consumption involved and the latency overhead introduced. Realtime applications targeted for these devices require predictability and determinism, which comes from data coherency being coordinated at select times by the application software. As developers manage this coherency, they develop designs that run faster and at lower power because they control when and if local data must be replicated into different memories. Figure 11 describes the coherency and non-coherency of the cache.

As with L2 cache, prefetch coherency is not maintained across cores. It is the application responsibility to manage coherency, either by disable the prefetch for certain memory segment, or by invalidate the prefetch data if necessary.

TI provides a set of API functions to perform cache coherency and prefetch coherency operations including cache line invalidation, cache line writeback to stored memory, and a writeback-invalidation operation.

Figure 11 Cache Coherency Mapping



In addition, if any portion of the L1s is configured as memory-mapped SRAM, there is a small paging engine built into the core (IDMA) that can be used to transfer linear blocks of memory between L1 and L2 in the background of CPU operation. IDMA transfers have a user-programmable priority to arbitrate against other masters in the system. The IDMA may also be used to perform bulk peripheral configuration register access.

In programming a TCI66XX or C66XX device, it is important to consider the processing model. Figure 11 shows how each core has local L1/L2 memory and a direct connection to the MSMC (memory Subsystem multi Core) that provides access to the shared L2 memory and to the external DDR3 SDRAM (if present in the system).

6.3 Shared Code Program Memory Placement

When CPUs execute from a shared code image, it is important to take care to manage local data buffers. Memory used for stack or local data tables can use the aliased address and will therefore be identical for all cores. In addition, any L1D SRAM used for scratch data, with paging from L2 SRAM using the IDMA, can use the aliased address.

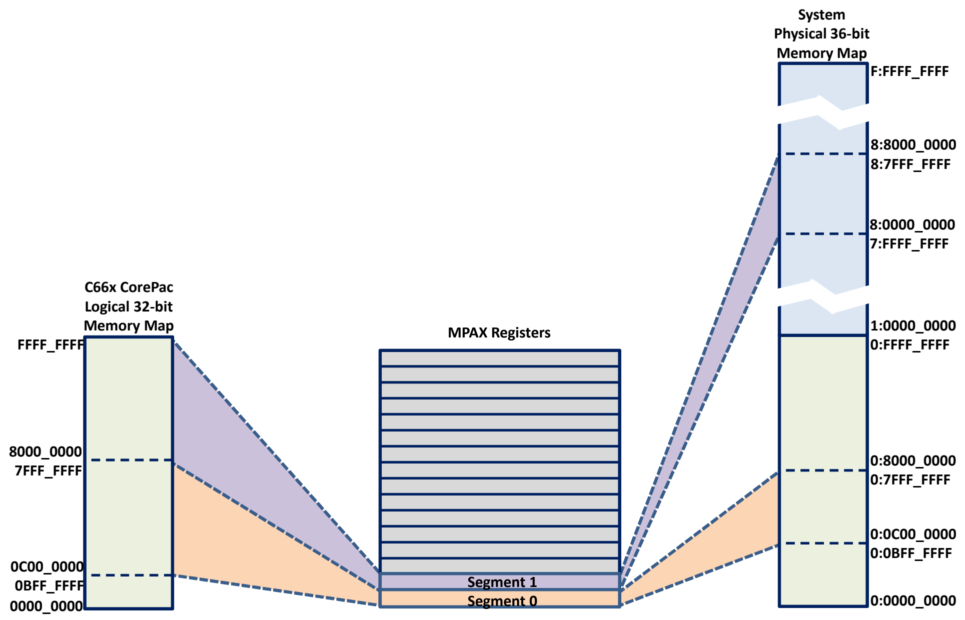
As mentioned previously, DMA masters must use the global address for any memory transaction. Therefore, when programming the DMA context in any peripheral, the code must insert the core number (DNUM) into the address.

To partition external memory sections between cores in the KeyStone family of devices, the application uses the MPAX module. Using MPAX, a KeyStone SoC with native addressing of 32-bits can address memory space of 64 Gbytes addressable with a 36-bit address. There are multiple MPAX units available in the KeyStone SoC which allows address translation for all masters of the SoC to shared memories like MSM SRAM and

DDR memory. The C66x CorePac uses its own MPAX modules to extend 32-bit addresses to 36-bit addresses before presenting them to the MSMC module. The MPAX module uses MPAXH and MPAXL registers to do address translation per master.

6.3.1 Using the Same Address for Different Code In Shared Memory

As mentioned previously for the Keystone family of devices, the XMC of each core has 16 MPAX registers that translate 32-bit logical addresses into 36-bit physical addresses. This feature enables the application to use the same logical memory address in all cores and to configure the MPAX registers of each core to point to a different physical address.



Detailed information about how to use the MPAX registers is given in Chapter 2 of the *KeyStone Architecture Multicore Shared Memory Controller (MSMC) User Guide* (SPRUGW7) [4].

6.3.2 Using a Different Address for the Same Code In Shared Memory

If the application uses a different address for each core, the per-core address must be determined at initialization time and stored in a pointer (or calculated each time it is used).

The programmer can use the formula:

<base address> + <per-core-area size> × DNUM

This can be done at boot time or during thread-creation time when pointers are calculated and stored in local L2. This allows the rest of the processing through this pointer to be core-independent, so the correct unique pointer is always retrieved from local L2 when it is needed.

Thus, the shared application can be created, using the local L2 memory, so each core can run the same application with little knowledge of the multicore system (such knowledge is only in initialization code). The actual components within the thread are not aware that they are running on a multicore system.

For the KeyStone family of devices, the MPAX module inside each CorePac can be configured to a different address for the same program code in the shared memory.

6.4 Peripheral Drivers

All device peripherals are shared and any core can access any of the peripherals at any time. Initialization should occur during the boot process, either directly by an external host, by parameter tables in an I^2C EEPROM, or by an initialization sequence within the application code itself (one core only). For all runtime control, it is up to the software to determine when a particular core is to initialize a peripheral.

Generally speaking, peripherals that read or write directly from a memory location use a generic DMA resource that is either built into the peripheral or provided by an EDMA controller or controllers (depending on the device). Peripherals that send or receive data based on a routing scheme use the Multicore Navigator and have an instance of PKTDMA.

Therefore, when a routing peripheral such as SRIO type 9 or type 11 or a NetCP Ethernet coprocessor is used, the executable must initialize the peripheral hardware, the PKTDMA that is associated with the peripheral, and the queues that are used by the peripheral and by the routing scheme.

Each routing peripheral has dedicated transmit queues that are hard-connected to the PKTDMA; when a descriptor is pushed into on of these TX queues, the PKTDMA sees a pending signal that prompts it to pop the descriptor, read the buffer that the descriptor is linked to if it is a host descriptor, convert the data to a bitstream, send the data, and recycle the descriptor back into a free descriptor queue. Note that all cores that send data to a peripheral use the same queues. Usually each TX queue is connected to a channel. For example, SRIO has 16 dedicated queues and 16 dedicated channels where each queue is hard-connected to a channel. If the peripheral sets priorities based on its channel number, pushing a descriptor to different queue results in a different priority for the transmit data. See the *KeyStone Architecture Multicore Navigator User Guide* (SPRUGR9)[2]for more information about channels priorities.

While the transmit queues for peripherals are fixed, receive queues can be chosen from a general purpose queue set or from a special queue set based on the notification methods used to notify a core that a descriptor is available for processing (as described in chapter 3.3). For the pulling method, any general purpose queue can be used. Special interrupt queues should be used for the fastest response. Accumulation queues are used to reduce context switching for delayed notification method.

The application must configure the routing mechanism. For example, for NetCP the user can route packets based on L2, L3, or L4 layers or any combination of the above. The application must configure the NetCP engine to route any package. To route a package to a specific core, the descriptor must be pushed into a queue associated with that core. The same is true for SRIO: the routine information, ID, mailbox and letters for type 11, stream ID for type 9 must be configured by the application.

Peripherals that use memory location directly (SRIO directIO, HyperLink, PCI express) have built-in DMA engines to move data to and from memory. When the data is in memory, the application is responsible to assign one or more cores to access the data.

For each of the DMA resources on the device—PKTDMA or built-in DMA engine—the software architecture determines whether all resources for a given peripheral will be controlled by a single core (master control) or if each core will control its own (peer control). With the TCI66XX or C66XX, as summarized above, all peripherals have multiple DMA channel context as part of the PKTDMA engine or the DMA built-in engine that allows for peer control without requiring arbitration. Each DMA context is autonomous and no considerations for atomic access need to be taken into account.

Because a subset of the cores can be reset during runtime, the application software must own re-initialization of the reset cores so that it avoids interrupting cores that are not being reset. This can be accomplished by having each core check the state of the peripheral it is configuring. If the peripheral is not powered up and enabled for transmit and receive, the core will perform the power up and global configuration. There is an inherent race condition in this method if two cores read the peripheral state when it is powered down and begin a power up sequence, but this can be managed by using the atomic monitors in the shared memory controller (SMC) or other synchronization methods (semaphores and others).

A host control method allows deferring the decision on device initialization to a higher layer outside the DSP. When a core needs to access a peripheral, it is directed by this upper layer whether to perform a global or a local initialization.

6.5 Data Memory Placement and Access

Memory selection for data depends primarily on how the data is to be transmitted and received and the access pattern/timing of the data by the CPU(s). Ideally, all data is allocated to L2 SRAM. However, there is often a space limitation in the internal DSP memory that requires some code and data to reside off-chip in DDR3 SDRAM.

Typically, data for runtime critical functions are located within local L2 RAM for the core to which the data is assigned and non-time-critical data such as statistics are pushed to external memory and accessed through the cache. When runtime data must be placed off-chip, it is often preferred to move data using EDMA and ping-pong buffer structure between external memory and L2 SRAM rather than access through the cache. The trade-off is simply control overhead versus performance, though even if accessing the data through the cache, coherency must be maintained in software for any DMA of data to or from the external memory.

7 DSP Code and Data Images

To better support the configuration of multicore devices, it is important to understand how to define the software project(s) and OS partitioning. In this section, SYS/BIOS will be referenced, but comparable considerations would need to be observed for any OS.

SYS/BIOS provides configuration platforms for all Texas Instruments C64xx and C66xx devices. In the SYS/BIOS configuration for any of the multicore SoCs, there are separate memory sections for local L2 memory (LL2RAM) and shared L2 memory (SL2RAM). Depending on how much of the application is common across the cores, different configurations are necessary to minimize the footprint of the OS and application in the device memory.

7.1 Single Image

The single image application shares some code and data memory across all cores. This technique allows the exact same application to load and run on all cores. If running a completely shared application (when all cores execute the same program), only one project is required for the device, and likewise, only one SYS/BIOS configuration file is required. As mentioned previously, there are some considerations for the code and linker command file:

- The code must set up pointer tables for unique data sections that reside in shared L2 or DDR SDRAM.
- The code must add DNUM to any data buffer addresses when programming DMA channels.
- The linker command file should define the device memory map using aliased addresses only.

7.2 Multiple Images

In this scenario, each core runs a different and independent application. This requires that any code or data placed in a shared memory region (L2 or DDR) be allocated a unique address range to prevent other cores from accessing the same memory region.

For this application, the SYS/BIOS configuration file for each application adjusts the locations of the memory sections to ensure that overlapping memory ranges are not accessible by multiple cores.

Each core requires a dedicated project—or at least a dedicated linker command file—if the code is to be replicated. The linker output needs to map all sections to unique addresses, which can be done using global addressing for all sections. In this case, there is no aliasing required, and all addresses used by DMA are identical to those used by each CPU.

7.3 Multiple Images with Shared Code and Data

In this scenario, a common code image is shared by different applications running on different cores. Sharing common code among multiple applications reduces the overall memory requirement while still allowing for the different cores to run unique applications.

This requires a combination of the techniques used for a single image and multiple images, which can be accomplished through the use of partial linking.

The output generated from a partially-linked image can be linked again with additional modules or applications. Partial linking allows the programmer to partition large applications, link each part separately, then link all the parts together to create the final executable. The TI Code Generation tool's linker provides an option (–r) to create a partial image. The –r option allows the image to be linked again with the final application.

There are a few restrictions when using the –r linker option to create a partial image:

- Conditional linking is disabled. The memory requirement may increase.
- Trampolines are disabled. All code needs to be within a 21-bit boundary.
- .cinit and .pinit cannot be placed in the partial image.

The partial image must be located in shared memory so all the cores can access it, and it should contain all code (.bios, .text, and any custom code sections) except for .hwi_vec. It should also contain the constant data (.sysinit and .const) needed by the SYS/BIOS code in the same location. The image is placed in a fixed location, with which the final applications will link.

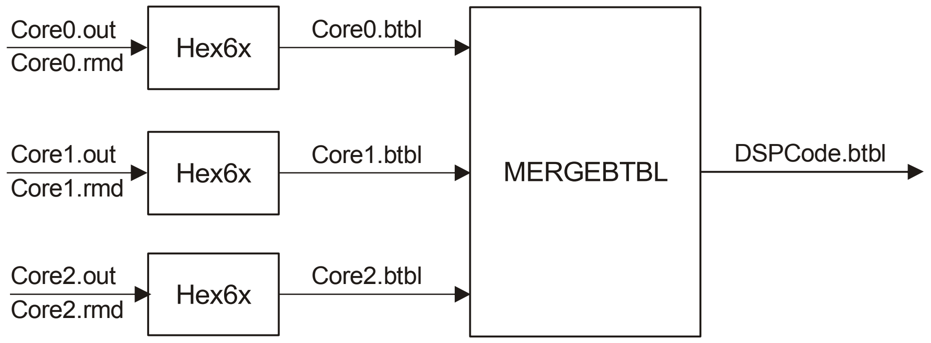
Because the SYS/BIOS code contains data references (.far and .bss sections), these sections need to be placed in the same memory location in non-shared memory by the different applications that will link with this partial image. ELF Format requires that the .neardata and .rodata sections be placed in the same section as .bss. For this to work correctly, each core must have a non-shared memory section at the same address location. For the C64xx and C66xx multicore devices, these sections must be placed in the local L2 of each core.

7.4 Device Boot

As discussed in Section 6, there may be one or more projects and resulting .out files used in software development for a single device depending on the mix of shared and unique sections. Regardless of the number of .out files created, a single boot table should be generated for the final image to be loaded in the end system.

TI has several utilities to help with the creation of the single boot table. Figure 12 shows an example of how these utilities can be used to build a single boot table from three separate executable files.

Figure 12 Boot Table Merge



Once a single boot table is created, it can be used to load the entire DSP image. As mentioned previously, there is a single global memory map, which allows for a straightforward boot loading process. All sections are loaded as defined by their global address.

The boot sequence is controlled by one core. After device reset, Core 0 is responsible for releasing all cores from reset after the boot image is loaded into the device. With a single boot table, Core 0 is able to load any memory on the device and the user does not need to take any special care for the multiple cores other than to ensure that code is loaded correctly in the memory map to all cores' start addresses (which is configurable).

Details about the bootloader are available in TI user guides SPRUEA7, *TMS320TCI648x DSP Bootloader* [5] and SPRUG24, *TMS320C6474 DSP Bootloader* [6], and SPRUGY5, *Bootloader for KeyStone Devices User's Guide* [7].

7.5 Multicore Application Deployment (MAD) Utilities

Tools for deploying applications on Multicore devices are supplied with the Multicore Software Development Kit (MCSDK) Version 2.x. See the MAD Utilities User's Guide for details about how to leverage these tools to deploy applications. The MAD Utilities are stored in the following folder:

<MCSDK_INSTALL_DIR>\mcsdk_2_xx_xx_xx\tools\boot_loader\mad-utils

7.5.1 The MAD Utilities

The MAD Utilities provide a set of tools for use at both build and run time for deploying an application.

- Build Time Utilities
  - **Static Linker** — For linking the applications and dependent dynamic shared objects (DSO)
  - **Prelink Tool** — For binding segments in an ELF file to virtual addresses
  - **MAP Tool** — Multicore Application Prelinker (MAP) tool to assign virtual address to segments for multicore applications

- Runtime Utilities
  - **Intermediate Bootloader** — provides the functionality of downloading the ROM file system image to the device's shared external memory (DDR)
  - **Mad Loader** — provides the functionality of starting an application on a given core

For additional information about the MAD Utilities, see the MAD Tools User's Guide.

7.5.2 Multicore Deployment Example

The Image Processing example supplied with the MCSDK utilizes the MAD tools for Multicore deployment. This example is supplied in the following folder:

<MCSDK_INSTALL_DIR>\mcsdk_2_xx_xx_xx\demos\image_processing

For additional information about the Image Processing Example, see the MCSDK Image Processing Demonstration Guide.

8 System Debug

The Texas Instruments C64xx and C66xx devices offer hardware support for visualization of the program and data flow through the device. Much of the hardware is built into the core, with system events used to extend the visibility through the rest of the chip. Events also serve as synchronization points between cores and the system, allowing for all activity to be “stitched together” in one timeline.

8.1 Debug and Tooling Categories

There are hardware and software tools available during runtime that can be used to debug a specific problem. Given that different problems can arise during different phases of system development, the debug and tooling resources available are described in several categories. The four scenarios are shown in Table 3.

Table 3 Debug and Tooling Categories

	Resident Configuration	**Debug Configuration**
Emulation Hardware	• Configured at start-up and always available for non-intrusive debug • Resources may be steered by application or external host, based on system events that are available within the application (e.g. no code modification required) • May be intrusive to the system software, depending on when configuration occurs (startup vs. runtime), but performance is not changed when leveraged for diagnostics	• Used as needed for system bring-up issues • Resources must be traded off to look at points of interest • May be intrusive to the system performance, depending on the resources used to investigate a problem • May require multiple runs of the software to collect all needed information
Software Instrumentation	• Code must be built with hooks to prevent the need to re-compile for diagnostic purposes • Hooks leveraged during runtime either by software (through host interaction) or through Code Composer Studio (CCS) commands • Host tools/processor can analyze data offline while system is running • May be intrusive to the software performance, but performance is not changed when leveraged for diagnostics as it is always present	• Code must be re-compiled to include additional diagnostic capability • Hooks enabled during compile-time and re-loaded onto target • Host tools/processor can analyze data offline while system is running • May be intrusive to the software performance and may modify system behavior slightly, depending on the resources used

While the characteristics described in Table 3 are not unique to multicore devices, having multiple cores, accelerators, and a large number of endpoints means that there is a lot of activity within the device. As such, it is important to use the emulation and instrumentation capabilities as much as possible to ease the complexity of debugging realtime problems in the development, test, and field environments. The following sections outline the system software instrumentation required to generate trace captures and logs for a particular problem.

8.2 Trace Logs

Fundamentally, the code running on each of the cores must be instrumented and the available hardware emulation logic configured to generate a *trace* of the software and data flow of the device execution. This process supports debugging any problems found during development or after deployment of the system. Trace logs can be enabled all the time or just during debug sessions, and can include any of the following data items:

- **API call log:** The target software incorporates logging functionality to record all API calls of interest. API calls can be recorded in memory with an ID, timestamp, and any parameters of interest.
- **Statistics log:** Chip-level statistics can be captured periodically to provide a picture of the activity through the SCR switch fabric over time. Statistics include bus monitors, event counters, and any other data of interest. This is typically resident in the system, although different/additional statistics may be optionally captured during debug.
- **DMA transaction log:** DMA transfers of interest can trigger a statistics capture, including timer values, chip registers, and data tables. This is typically resident in the system, although different/additional events and transactions may be optionally captured during debug.
- **Core trace log:** Core advanced emulation trigger (AET) can trace system events of interest, correlated to the CPU time. This is typically resident in the system, although different system events may be traced during debug. Also, PC trace may be added to the trace log. If data trace is desired during debug, it requires disabling the event trace.
- Other events/data can be recorded in a log buffer, as desired, by the CPU or DMA. The usage here is entirely customer-specific.

Historical information can then be used to construct a standalone test case using the same control and data flows that reproduce a scenario in the lab for further analysis.

8.2.1 API Call Log

The API call log is based on software instrumentation within the target software. Multiple logs may be correlated with respect to time either on the same core or across cores. The API call log is recorded by software directly into device memory.

Each of the API records will be accompanied by a timestamp to allow correlation with other transaction logs. The content of the logs may be useful in understanding both the call flow as well as details about the processed information at various times during execution.

8.2.2 Statistics Log

The statistics log consists of chip statistics taken at regular intervals that give a high-level picture of the device activity. The DDR, receive accelerator (RAC), and antenna interface (AIF) modules all have built-in statistics registers to keep track of bus activity. These statistics can be captured at regular intervals to record the activity to those modules during each time window. The log can then be used to give a high-level view of the data flow through the SCR switch fabric during each time window.

Statistics recorded by software in memory can also be recorded in the statistics log.

In addition to the statistics values, a chip time value must also be recorded to allow correlation with other transaction logs.

There is some flexibility in the statistics to be captured by the system, so the configuration of the statistics capture is left to the application. The required format is for the log to contain a time value following by the statistics of interest. Multiple logs are possible, provided each holds a time value to allow correlation with the others.

NOTE—For C66x generation devices, the Statistics Log can also be captured via System Trace. See "System Trace" on page 50 for more information. System Trace is not implemented on the C64x+ generation of devices.

8.2.3 DMA Transaction Log

Given the amount of data traffic that is handled within the SCR switch fabric by the EDMA controller, it is useful to record when certain DMA transactions take place. EDMA channels can be configured to trigger a secondary DMA channel to record statistics related to its activity: an identifier and reference time. Each DMA channel of interest can have a transaction log in which the transfer identifier, time of transfer, and any relevant information surrounding the transfer can be recorded. The number of transaction logs is flexible, and is limited only by the number of EDMA channels that can be dedicated to performing the recording.

The time value recorded with each entry should have a relationship to the time value used in the other transaction logs to allow correlation with other chip activity.

8.2.4 Event Log

The event logs are provided by each core through their event trace capability. Event trace allows system events to be traced along with the CPU activity so that the device activity in relation to the processing being performed within the CPUs can be understood. The trace data output from each of the cores can be captured off-chip through an emulator or on-chip in the embedded trace buffers. Event logs do add additional visibility to the state of the processor over time, but also use additional free-running hardware and could be a power consumption concern in a deployed system. During development, however, the event trace can be used in conjunction with the other transaction logs for greater visibility.

The event log allows the recording of PC discontinuities, the execute/stall status for each CPU cycle, and up to eight system events (user programmable). In order to correlate the event traces of multiple cores with one another, and with the other transaction logs, one of the eight system events must be a time event common to the other logs.

8.2.5 Customer Data Log

Additional instrumentation of the application software is possible and should follow the guidelines outlined for the other transaction logs to record a timestamp with each entry to allow correlation to other chip activity. The contents of each entry can be anything meaningful in the customer system.

8.2.6 Correlation of Trace Logs

As mentioned in Section 8.2, a common system time event needs to be used to correlate the multiple trace logs collected by the system to build a complete view of the program and data flow on the chip. All API, statistics, DMA, and data logs must include a count value that corresponds to the window in time for which the log data was collected. The recording of the time value may be different depending on the type of log it is, but provided that the counts are taken from the same base and with a common period or relationship, the logs can be merged together.

The count is recorded as described in Table 4.

Table 4 Event Log Time Markers

Log	Time Event(s)	Recorded	Relationship to Log Data
API Call	System time	With each API call	Reflection of the point at which the call was made
Statistics	System time (at interval $x \times p$)	At time of statistics collection	The end of the time window for which the statistics are valid
DMA Transaction	System time	After each DMA transaction of interest	Reflection of the point at which the DMA transfer took place
Event	System time interval ($y \times p$) marker	Within the event stream	Marker at each time window boundary
	Program Counter	With each event recorded	Reflection of the PC value at the time of arrival of the event to the core
Data	System time	With each data record	Customer defined

In Table 4, the time intervals are shown as an integer (x or y) times a common period p. The integer multiples should all be integer multiples of one another (for example, there could be four statistics log windows for every DMA transaction log window).

For the API call log, the time value itself is recorded with each API call. Because the log recording is under CPU software control rather than DMA control, recording a window marker would require an interrupt and does not provide any additional information because the window can be determined by the count value divided by the window period, p.

The Statistics log gets a timestamp recorded in memory. Every $x \times p$ UMTS (universal mobile telecommunications system) cycles in time an event is asserted to the DMA to capture the time value and all statistics of interest. In addition, the statistics registers must be cleared to begin collecting over the next time window because the statistics represent events during the current window. The time value recorded along with the statistics data serves as the start time of the next window.

The DMA transaction log is similar to the API call log in that the time is recorded with each transaction or multiple chained transactions of interest. The time value is captured by a DMA channel that is chained to the transfer(s) of interest along with information necessary to identify the transaction(s). As with the API call log, the window to which the transaction records belong can be determined by dividing the value recorded by the period, p.

The Event log contains UMTS timing markers and CPU program counter (PC) markers. The UMTS time interval marker is used to correlate the event log to the other logs and serves to distinguish the collection windows. The time value represents the beginning of the time window. The CPU PC value is recorded with each time event and can be used to indicate the processing activity occurring during each time window. It may provide insight as to what caused some of the information collected in the other logs.

The customer data log is customer-defined, but should map to one or more of the above definitions. Examples of correlating different logs are shown in Table 5 and Table 6.

Table 5 Trace Log Correlation

CPU 0		DMA Log		System Trace	
Cycle	**Event**	**Entry**	**Data**	**Entry**	**Event**
10000	GLOBAL_TIME	0	GLOBAL_TIME = 10020	0	GLOBAL_TIME = 10080
10203	DMA_INT	0	ValueX	0	Transaction log
11150	EMAC_INT	0	ValueY	1	GLOBAL_TIME = 10110
11601	DMA_INT	1	GLOBAL_TIME = 10108	1	Transaction log
		1	ValueX	2	GLOBAL_TIME = 10220
		1	ValueY	2	Transaction log
				3	GLOBAL_TIME = 10280
				3	Transaction log
				4	GLOBAL_TIME = 10340
				4	Transaction log
				5	GLOBAL_TIME = 10400
				5	Transaction log
				6	GLOBAL_TIME = 10488
				6	Transaction log
12000	GLOBAL_TIME	2	GLOBAL_TIME = 12096	7	GLOBAL_TIME = 12060
12706	DMA_INT	2	ValueX	7	Transaction log
13033	EMAC_INT	2	ValueY	8	GLOBAL_TIME = 12120
13901	GPINT	3	GLOBAL_TIME = 13330	8	Transaction log
		3	ValueX	9	GLOBAL_TIME = 12180
		3	ValueY	9	Transaction log
				10	GLOBAL_TIME = 12240
				10	Transaction log
				11	GLOBAL_TIME = 12300
				11	Transaction log
				12	GLOBAL_TIME = 12360
				12	Transaction log
14000	GLOBAL_TIME	4	GLOBAL_TIME = 14100	13	GLOBAL_TIME = 14120
15006	DMA_INT	4	ValueX	13	Transaction log
15063	EMAC_INT	4	ValueY	14	GLOBAL_TIME = 14180
		5	GLOBAL_TIME = 14200	14	Transaction log
		5	ValueX	15	GLOBAL_TIME = 14240
		5	ValueY	15	Transaction log
				16	GLOBAL_TIME = 14300
				16	Transaction log
				17	GLOBAL_TIME = 14360
				17	Transaction log
				18	GLOBAL_TIME = 14420
				18	Transaction log

As described in the preceding sections, the trace logs can be correlated with one another using common time events. The core event trace has a PC value with each event, and the GLOBAL_TIME is the marker that is common to the other trace logs. The DMA log is recorded with every GLOBAL_EVENT (or multiple), and the UMTS Time recorded with the log entry shows which window in time. The timestamp recorded with each API call in the call log is the actual time.

Table 6 Core Event Trace Correlation

CPU 0		CPU 1		CPU 2	
Cycle	**Event**	**Cycle**	**Event**	**Cycle**	**Event**
		10161	SEM_INT	10115	DMA_INT
10000	GLOBAL_TIME	13001	GLOBAL_TIME	11061	GLOBAL_TIME
10203	DMA_INT	13070	DMA_INT		
11150	EMAC_INT	13404	GPINT		
11601	DMA_INT				
12000	GLOBAL_TIME	15001	GLOBAL_TIME	13044	GLOBAL_TIME
12706	DMA_INT	15390	DMA_INT	13910	DMA_INT
13033	EMAC_INT	16012	DMA_INT		
13901	GPINT				
14000	GLOBAL_TIME	16804	GLOBAL_TIME	15036	GLOBAL_TIME
15006	DMA_INT	17506	DMA_INT	16690	DMA_INT
15063	EMAC_INT	18029	DMA_INT		
16000	GLOBAL_TIME	19001	GLOBAL_TIME	17876	GLOBAL_TIME
16079	DMA_INT	19740	DMA_INT	18101	DMA_INT
		20406	DMA_INT		
				20485	GLOBAL_TIME
				20496	DMA_INT
				20500	GPINT
				21028	DMA_INT
				22008	GLOBAL_TIME

With the core event traces, each entry in the logs is referenced to the PC value of the core that is performing the trace function. Given that each core can stall independently of the others, the logs need to be correlated to one another using common time markers. The Global_TIME shown for each log is the same and matches that used for correlation to other trace logs.

Using the above information it is possible to build summaries per time window of the device operation, as shown in Table 7. This information provides details into the activity on each core as well as the system loading through the device interfaces and important events from user-defined sources.

Table 7 Time Window Trace Log Summary

Time Window 0					
Start	**UMTS Time 0**				
Core 0 Event Trace		**Core 1 Event Trace**		**Core 2 Event Trace**	
10000	TIME_EVENT	11500	TIME_EVENT	14350	TIME_EVENT
10203	DMA_INT0	11620	DMA_INT3	14440	DMA_INT6
11150	DMA_INT1	12110	DMA_INT4	14550	DMA_INT7
11601	DMA_INT2	12230	DMA_INT5	14590	DMA_INT6
		12950	DMA_INT3	14620	DMA_INT6
		12970	DMA_INT4	14680	DMA_INT6
		12970	DMA_INT5		

Statistics Summary			
Interface	**% Utilization**	**% Reads**	**% Writes**
DDR2	17.6	79.3	20.7
RAC (cfg)	3.1	5.0	95.0
RAC (data)	26.8	22.9	77.1
AIF	86.4	50.9	49.1

General Stats	
User Stat 1	8493
User Stat 2	26337

Similar trace and debug capability is integrated in the Data Visualization Tool (DVT) which works for the multicore C64x+ and KeyStone families of devices.

DVT has following capabilities:

- Graphical view of the execution of the tasks in different cores
- Graphical view of the synchronized execution of tasks in all the cores of the SoC
- Graphical view of the CPU loading for each of the cores
- Storing of logs in external memory for offline processing (generating graphs)

DVT uses mechanisms similar to those described previously. The method of capturing the timing information of the task execution is by instrumenting the Entry and Exit points in the task/code block with a function to capture the local core's current time.

In addition to a timestamp, DVT records a 32-bit Tag field that contains the CPU ID, process ID, process type, and location information, as follows:

- CPU ID: In this example, CPU ID specifies the core ID for a C6670 device, ranging from (cores) 0-3.
- Process ID: For the task to be profiled, this ID can be mapped to a string of the task name to display in DVT.

- Location: Debug instrumentation location; for example, the start of task, end of task, or intermediate position in the task.

There are standard macros defined in DVT that can be placed in the code as per the general guidelines described above. For example:

```
void function()
{
    /*variable initializations*/
LTEDEMO_DBG_TIME_STAMP_SWI_START(LTEDEMO_DBG_ID_SWI_SOFT_SYM);

    /* body of code
    */

    LTEDEMO_DBG_TIME_STAMP_SWI_END(LTEDEMO_DBG_ID_SWI_SOFT_SYM);
}
```

In the above example, LTEDEMO_DBG_ID_SWI_SOFT_SYM is an ID.

LTEDEMO_DBG_TIME_STAMP_SWI_START is a macro, defined as follows:

```
    #define LTEDEMO_DBG_TIME_STAMP_SWI_START( id ) \
    LTEDEMO_DBG_TIME_STAMP( LTEDEMO_DBG_PROC_SWI, LTEDEMO_DBG_LOC_START, id )
```

Unique IDs are defined for each task being instrumented. Different macros could be used according to where the macro is placed inside the code block.

All this information (CPU ID, Process ID, Location) is OR'd bitwise to get the 32-bit tag information. DVT stores the 32-bit tag information and the corresponding time read from the TSCL register of the core. This information can be dumped in standard CCS DAT file format which is used by the tool to generate the graphs.

The timestamp that DVT captures is based on the *local* TSCL register associated with each core. Therefore, the data collected from each core is not synchronized with the other cores. A BIOS interrupt task with a different process ID is used for synchronization. Reference time is recorded on each core when the interrupt is received. Each time entry from a given core will be adjusted by a specific formula to achieve synchronization. For more information about this formula, see the online help supplied with the DVT installation.

The examples in Figure 13, Figure 14, and Figure 15 show the graphical information generated for each of the different scenarios.

Figure 13 Execution Related to Cores

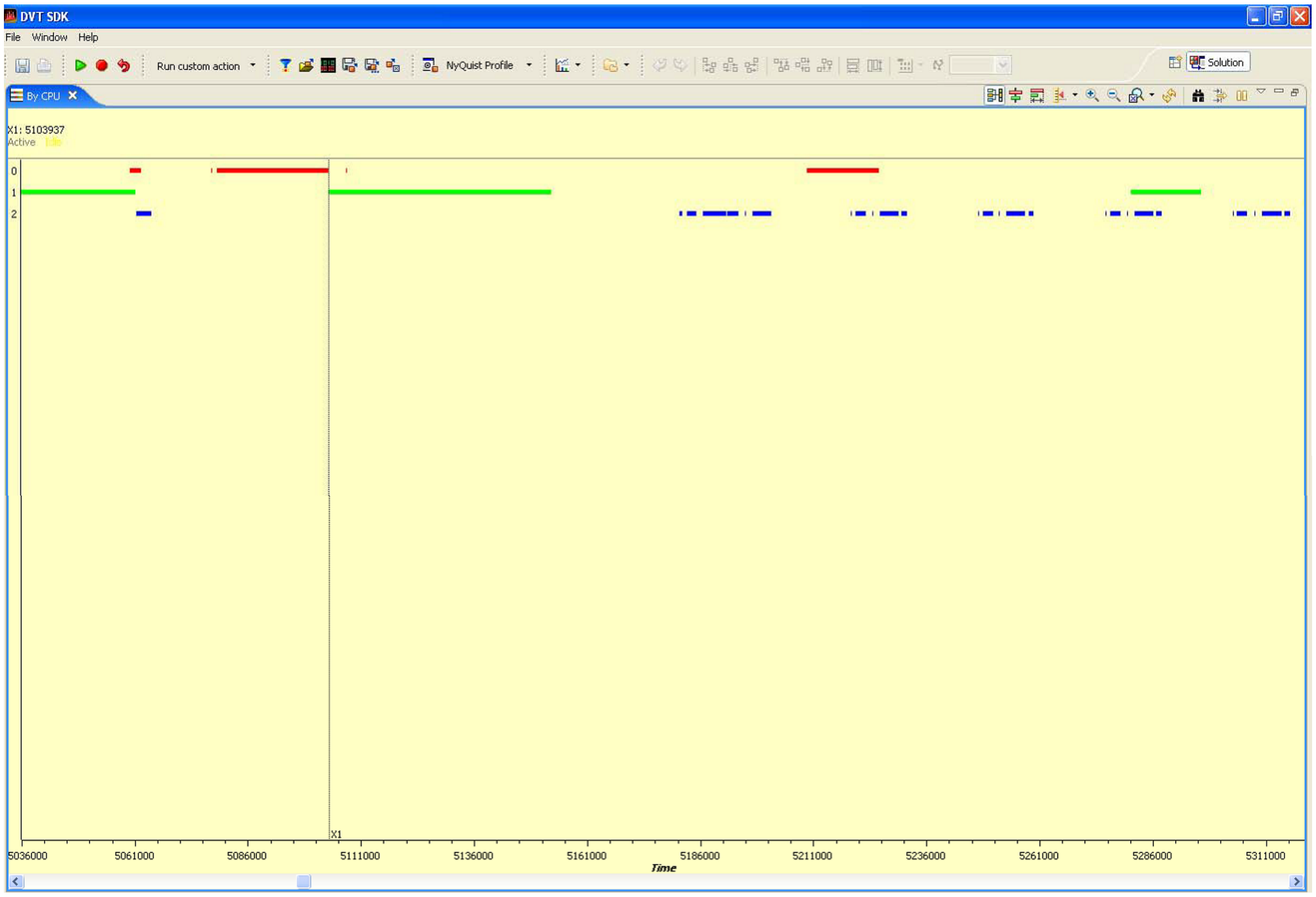


Figure 13 shows the activity on three cores, as follows:

- The x-axis is time (in microseconds).
- The activity on each core is represented by a unique color; for example, Core 0 activity is visualized in red.
- The length of time that the color across the “0” line is red indicates the time intervals that the CPU is active; that is, a task is being executed on that core. Otherwise, it is in Idle state.

Figure 14 Execution Related to Tasks

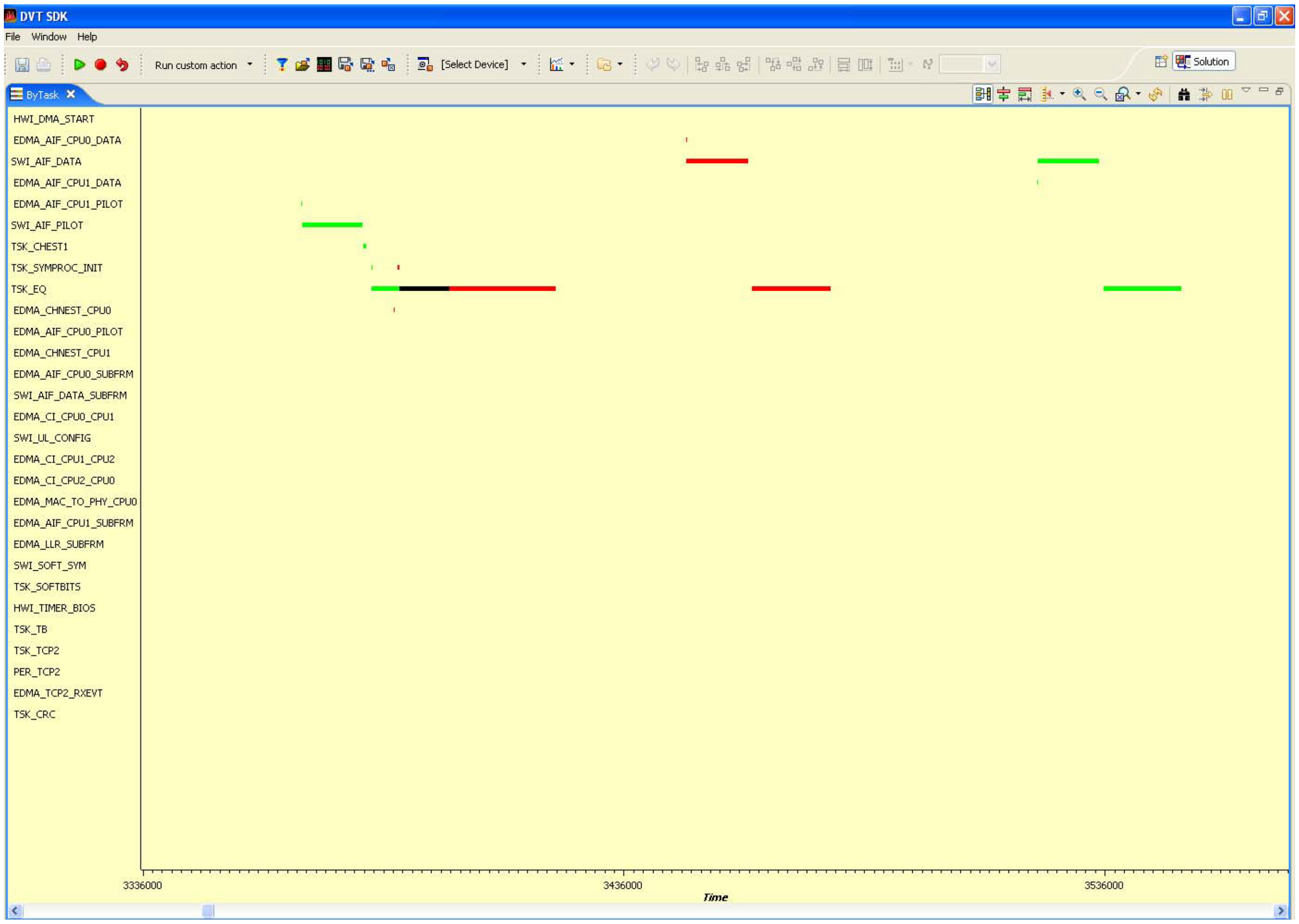


Figure 14 shows activity in terms of tasks at the SoC level in different cores, as follows:

- The x-axis shows time in microseconds. The y-axis lists different tasks.
- Consider a single task. It would have different colors across it, representing the active state of the task on the corresponding CPU core. For example, from time t1 to t2, the task was executed on Core 0 (green); from t2 to t3, it was executed on both Core 0 and Core 1(black); and from t3 to t4, it was executed on Core 1 (red).
- Small lines are also visible in the graph, which represent either interrupts or points of interest in the task execution.

Figure 15 CPU Load Graph

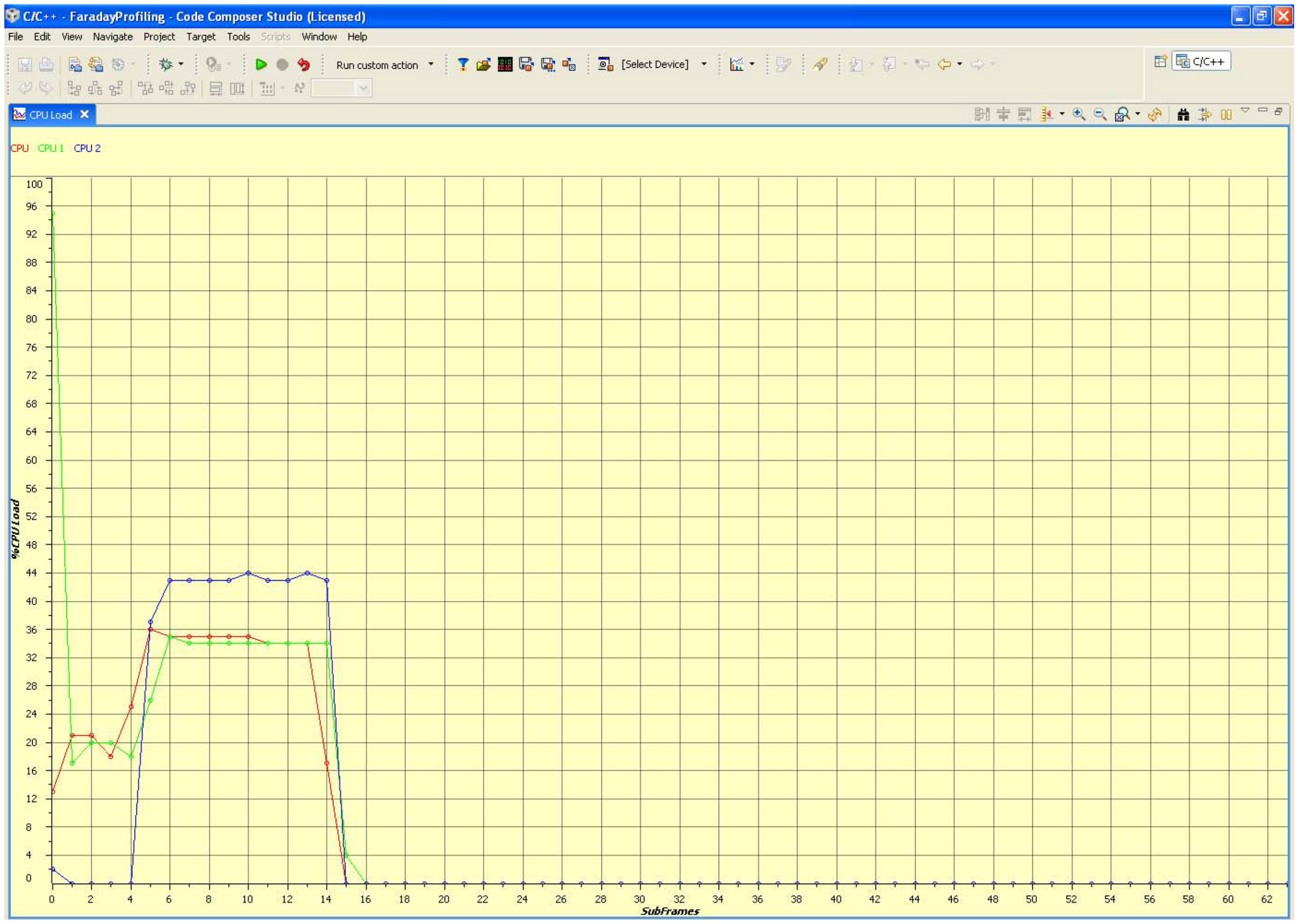


Figure 15 shows the CPU load plotted using a line graph:

- The x-axis is the number of subframes. The y-axis is the CPU percentage load. For example, subframe 10, Core 0 was 40 percent loaded, Core 1 was 30 percent loaded, and Core 2 was 30 percent loaded.
- The CPU load information is obtained for each of the cores separately. In a given MIPS (Million Instructions per Second) window, the idle time of a core is calculated. The active time is determined by subtracting the idle time from the MIPS window. This gives the CPU load information for a given core.

DVT comes under the System Analyzer which is installed by default with CCS. See the CCS help for more information about DVT and System Analyzer.

8.3 System Trace

System Trace is a technology for gathering system-level execution data with limited or no intrusiveness to the application. System Trace was initially deployed on the C66xx generation of multicore devices and is not available on previous generations (C64xx). With System Trace, the statistics traditionally captured by instrumented code can now be captured automatically using the logic built into the SoC without consuming precious system resources.

System Trace provides the ability to capture messages on-chip and pass them to an external emulator or store them in an on-chip embedded trace buffer. Each message that is output via System Trace is allocated a system-level timestamp, which enables synchronization across the entire system. System Trace provides the ability to generate two types of messages: Hardware and Software.

8.3.1 Hardware Messages

Each device that supports System Trace has a set of statistics counters called Common Platform Tracers (CP Tracers). The CP Tracers are located on slave interfaces on the device, such as the DDR interface. The statistics counters can be configured to measure access statistics over a specified time. When that time expires, the measured statistics are automatically output in the System Trace stream. The captured data can then be used to visualize these measurements throughout the execution of the application to locate processing bottlenecks. The data captured here is similar to what is available in the Statistics Log, but without the need for code instrumentation or the consumption of CPU cycles to capture the data.

8.3.2 Software Messages

Software messages are STM messages that are generated through software execution.

These functions give `printf`-like capabilities without the intrusiveness that a traditional `printf` requires. In addition, each message is given a system-level timestamp, which allows users to instrument their code for debug purposes and view cycle-accurate logs. The flexibility of the software messages enables users to visualize their application in many ways. Simple examples might be generating a cycle accurate, multicore thread execution graph, or diagnosing error conditions to determine where messages are being lost between the cores.

9 Summary

In this paper, three programming models for use in real-time multicore applications are described, and a methodology to analyze and partition application software for a multicore environment is introduced. In addition, features of the Texas Instruments KeyStone family of TCI66xx and C66xx multicore processors used for data transfer, communication, resource sharing, memory management and debug are explained.

TI TCI66xx and C66xx processors offer a high level of performance through efficient memory architectures, coordinated resource sharing, and sophisticated communication techniques. To facilitate customers achieving full performance from these parts, TI has included hardware in the devices to allow the cores to both execute with minimal overhead and to easily interact with each other through signaling and arbitration. These devices also contain hardware that provides trace and debug visibility into the multicore system.

There are tools available like System Analyzer which uses those hardware capabilities and can allow customers to debug at the SoC level in the multicore environment.

TI's multicore architectures deliver excellent cost/performance and power/performance ratios for customers who require maximum performance in small footprints with low power requirements. As the leader in many applications that require high-performance products, TI is committed to multicore technology with a robust roadmap of products.

10 References

See the following documents for additional information regarding this application note.

1 Ankit Jain, Ravi Shankar. *Software Decomposition for Multicore Architectures,* Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL, 33431. Internet. http://www.csi.fau.edu/download/attachments/327/Software_Decomposition_for_Multicore_Architectures.pdf?version=1

2 TI user guide SPRUGR9, *KeyStone Architecture Multicore Navigator User Guide* http://www.ti.com/lit/pdf/sprugr9

3 TI user guide SPRUGW0, *TMS320C66x CorePac User Guide* http://www.ti.com/lit/pdf/sprugw0

4 TI user guide SPRUGW7, *Multicore Shared Memory Controller (MSMC) User Guide* http://www.ti.com/lit/pdf/sprugw7

5 TI user guide SPRUEA7, *TMS320TCI648x DSP Bootloader* http://www.ti.com/lit/pdf/spruea7

6 TI user guide SPRUG24, *TMS320C6474 DSP Bootloader* http://www.ti.com/lit/pdf/sprug24

7 TI user guide SPRUGY5, *Bootloader User Guide for KeyStone Devices* http://www.ti.com/lit/pdf/sprugy5

8 TI user documentation, *MCSDK 2.1 Addendum* http://processors.wiki.ti.com/index.php/BIOS-MCSDK_2.1_Addendum

9 *OpenMP Specification* http://openmp.org/wp/openmp-specifications/

IMPORTANT NOTICE

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		