Application Report
SPRAB27A—August 2009

Multicore Programming Guide

Communications Infrastructure and Voice/DSP Systems

David Bell
Greg Wood

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

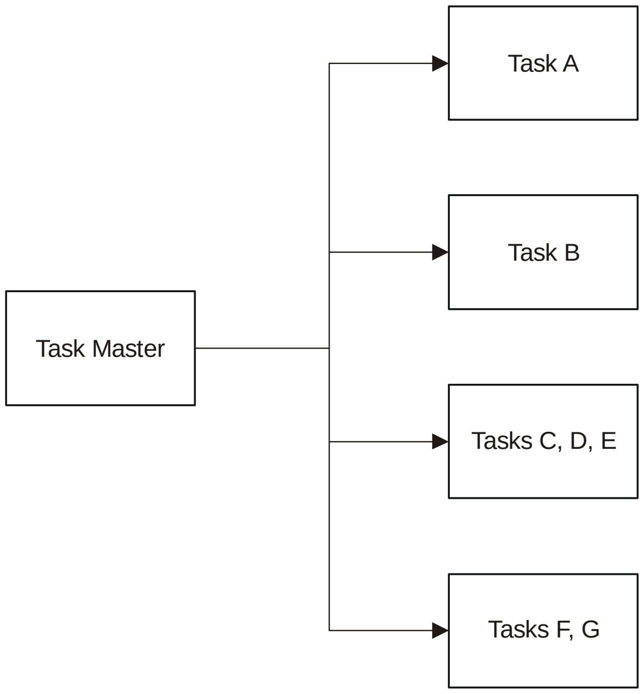
One of the first steps in mapping an application to a multicore processor is to identify the task parallelism and select a processing model that fits best. The two dominant models are a Master/Slave model in which one core controls the work assignments on all cores, and the Data Flow model in which work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

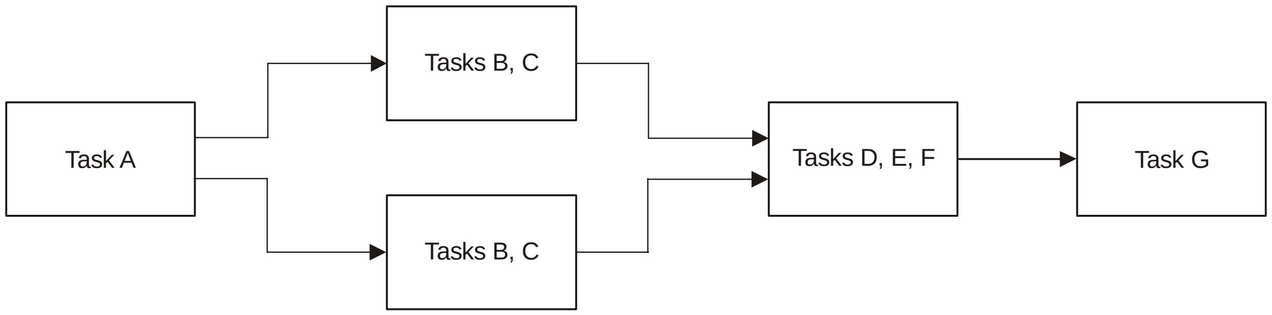
One or more execution threads are mapped to each core. Task assignment is achieved using message-passing between cores. The messages provide the control triggers to begin execution and pointers to the required data. Each core has at least one task whose job is to receive messages containing job assignments. 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 are 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 real time 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 effect 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 are passed between cores using shared memory or DMA transfers.

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 of 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.

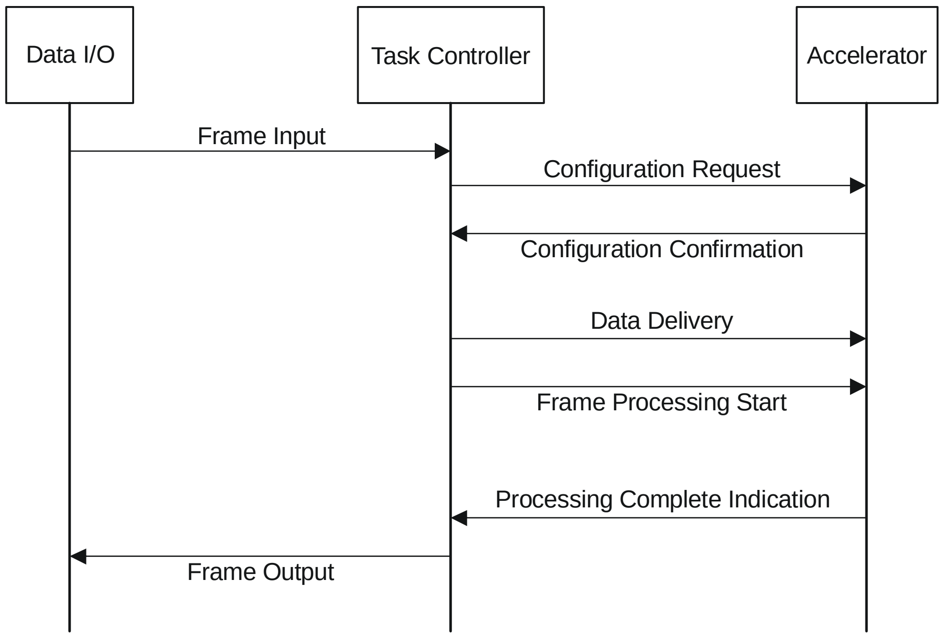
Partitioning the application into modules or subsystems is a matter of finding the breakpoints where coupling is low and cohesion is high. If a module has too many external dependencies, it should be grouped with another module that together would reduce coupling and increase cohesion. It is also necessary 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.

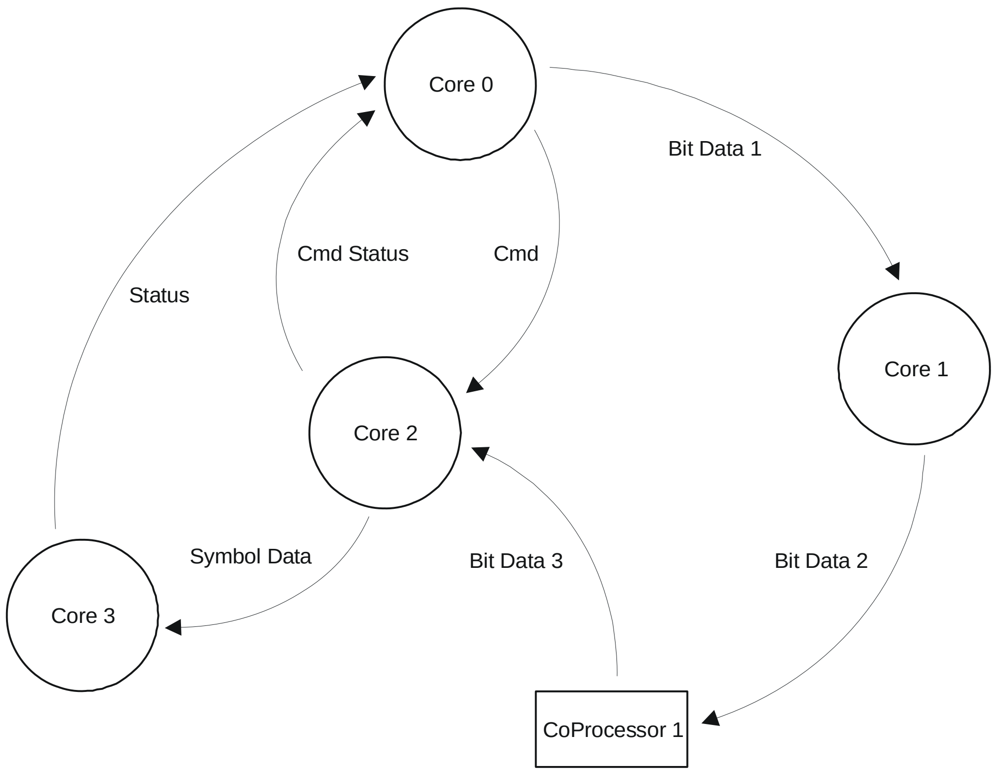
Control flow diagrams represent the execution paths between modules. Modules in a processing sequence that are not on the same core must rely on message passing to synchronize their execution and possibly require data transfers. Both of these actions can introduce latency. The control flow diagrams should be used to create metrics that assist the module grouping decision to maximize overall throughput. Figure 3 contains an example of a control flow diagram.

Figure 3 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. They also show 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 4 contains an example diagram.

Figure 4 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. It 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. Once all 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 individual 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 consisting of multiple levels of cache and off-chip memory. It is preferred to operate on data in cache to minimize the performance hit of 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.

3 Inter-Processor Communication

The Texas Instruments TCI64xx and C64xx multicore devices offer several architectural mechanisms to support inter-processor communication. All cores have full access to the device memory map, meaning 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 3rd-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.

Inter-core communication consists of two primary actions: data movement and notification.

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. The sending and receiving cores choose different mechanisms.

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 accessible by both sender and receiver, with each responsible for their 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 DSP/BIOS message queue transport, developed for TCI64x and C64xx multicore devices to send messages between cores, makes use of 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 local memory for their data, and overhead is reduced by keeping the data local. 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, then it may be advantageous for the receiver to control the data transfer to allow tighter management of its memory.

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 mentioned 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.

3.2 Notification

After the communication message data are 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.

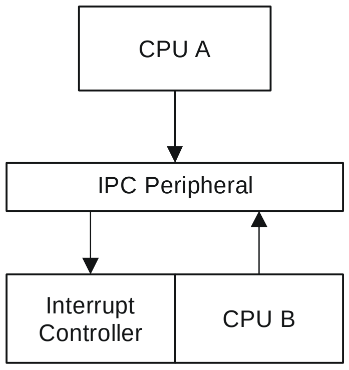
3.2.1 Direct Signaling

The devices support a simple peripheral that allows for a core to generate a physical event to another core. This event is routed through the cores' 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 5.

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 5 Direct IPC Signaling



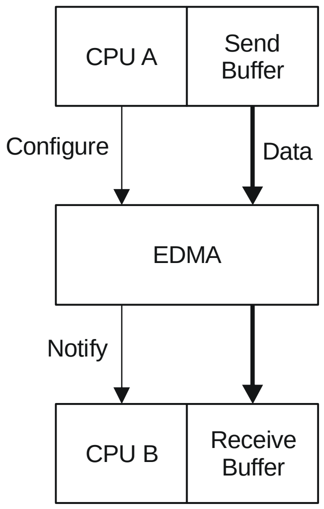
3.2.2 Indirect Signaling

If a 3rd-party transport, such as the EDMA controller, is used to move data, then the signaling between cores can also be performed through this transport as well. In other words, the notification follows the data movement in hardware, rather than through software control, as shown in Figure 6.

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 6 Indirect Signaling



3.2.3 Atomic Arbitration

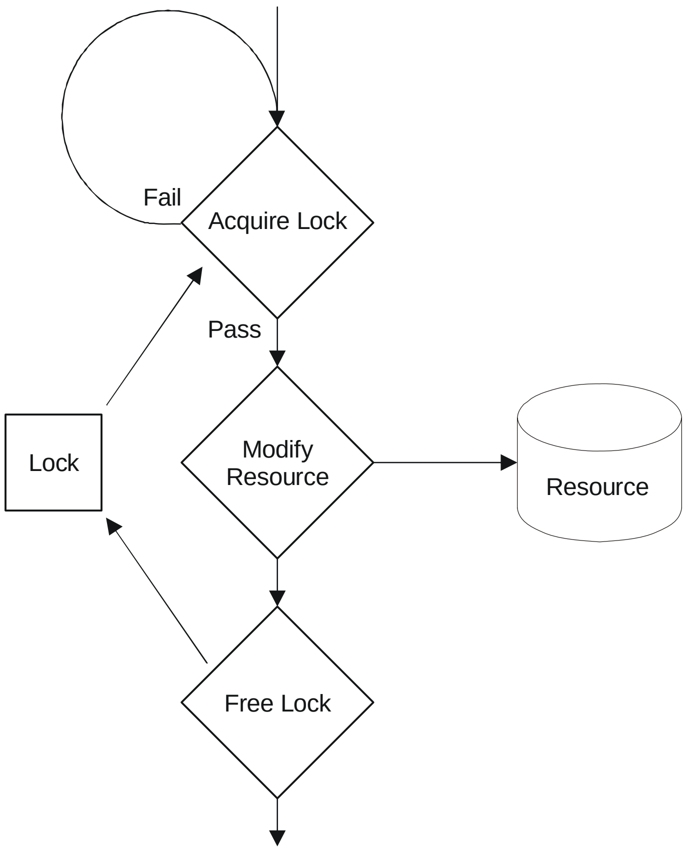
Each device includes hardware support for atomic arbitration. The supporting architecture does vary on different devices, but the same underlying function can be achieved easily. Atomic arbitration instructions are supported with hardware monitors in the Shared L2 controller present on the TCI6486 and C6472 devices, while a semaphore peripheral is present on the TCI6487/88 and C6474 devices because they do not have a shared L2 memory. 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 7.

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 7 Atomic Arbitration



4 Data Transfer Engines

Within the device, the primary data transfer engines on current Texas Instruments TCI64xx and C64xx devices are the EDMA (enhanced DMA) modules. For inter-core communication between devices, there are several transfer engines, depending on the physical interface selected for communication:

- **EDMA:** The EDMA is an integrated DMA transfer engine that can be used to transfer data between any two memory locations on the device, and is required for using some of the peripherals (device-specific).
- **Antenna Interface (TCI6487/88 and C6474 only):** The EDMA is used in conjunction with the AIF to transport data.
- **Serial RapidIO:** There are two modes available — DirectIO and Messaging. Depending on the mode, the EDMA or built-in DMA control are available.
- **Ethernet:** There is a built-in DMA controller for handling all of the data movement.

4.1 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 prior to accessing the resource.

4.2 Ethernet

The peripheral allows for up to 32 MAC addresses to be serviced by up to eight channels. These can be dedicated to a given core and used for broadcast or multi-cast. Each core can have a dedicated receive channel that is independently controlled. This allows for any number of the cores directly consuming Ethernet traffic to a given MAC address. Once a channel is allocated to a core, that core may access it directly without arbitration. Likewise, each core can control outbound Ethernet traffic directly.

4.3 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 four of these, each independent from the others, and each can submit transactions on 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 for multiple transfer channels. When using messaging protocols, each core is responsible for managing its own messaging traffic, and ownership can be assigned for in-bound messages based on RapidIO mailbox numbers. If using DSP/BIOS MSGQ over RapidIO, this is transparent to the user.

4.4 Antenna Interface

The AIF is serviced by multiple EDMA channels, whose timing is controlled by the frame synchronization (FSYNC) module. EDMA channels are set up once based on the system timing, board topology, and allocation of antenna streams to cores. Data can be directed to or from any core, which, in turn, dictates the EDMA channel allocation and programming. If multiple cores are using the AIF, then each core is required to allocate EDMA resources according to Section 4.1. In addition, the synchronization of the selected EDMA channels is critical. If circuit-switched transfer mode is selected, then the FSYNC module must be programmed to notify each core's associated EDMA channel(s) at the appropriate time to ensure data is correctly transmitted across the link. If packet-switched (PS) transfer mode is used, then the cores must control the EDMA resources efficiently to pass data through the PS FIFO.

5 Shared Resource Management

When sharing resources on the device, it is critical that there is a protocol followed uniformly by all of 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 3.2 mentioned signaling in the context of message passing. The same signalling mechanisms can be used for general resource management as well. One can use direct signaling or atomic arbitration between cores. Within a core, one can use a global flag or an OS semaphore. It is not recommended to use a simple global flag for inter-core 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.

5.2 OS Semaphores

All multi-task 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 for using them for single-core arbitration, as 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 the message passing, direct signaling can be used for simple arbitration. If there is only a small set of resources being shared between cores, then the IPC signaling described in Section 3.2.1 can be used. A protocol can be followed to allow a notify and acknowledge handshake to pass ownership of a resource.

6 Memory Management

In programming a multicore device, it is important to consider the processing model. On the Texas Instruments TCI64xx and C64xx 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 explained later in this section.

In the case of each core having its own code and data space, the aliased 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 run-time, aliased address is usable for direct CPU load/store and internal DMA (IDMA) paging, though not EDMA or other master transactions. These transactions must use the global address.

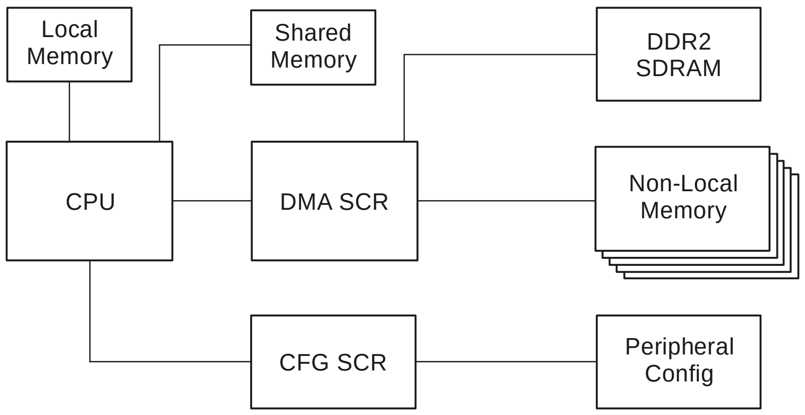
It is always possible for the software to verify on which core it is running as well, so the aliased addresses are not required to be used in common code. There is a CPU register (DNUM) that holds the DSP core number and can be read during run-time to conditionally execute code and update pointers.

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.

6.1 CPU View of the Device

Each of the CPUs has an identical view of the device. As shown in Figure 8, beyond each core's L2 memory there is a switched central resource (SCR) that inter-connects the cores, external memory interface, and on-chip peripherals through a switch fabric.

Figure 8 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 end-points, with priority assigned per master by user software for arbitration between all accesses at each end-point.

Each slave in the system (e.g. Timer control, DDR2 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 SPRU871, *TMS320C64x+ DSP Megamodule* [2].

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 [2]) 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 three 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. 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 run-time by a particular core, should always use the global address only.

6.2 Cache Considerations

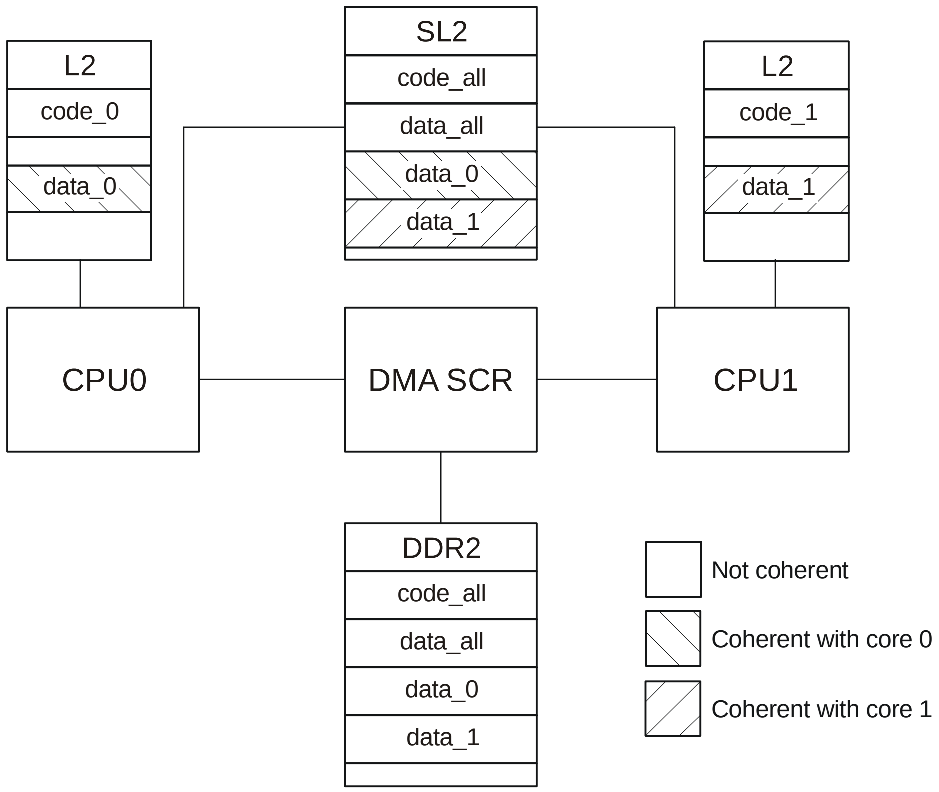
It is important to point out that the only coherency guaranteed by hardware with no software management is L1D cache coherency with L2 SRAM within the same core. The hardware will guarantee that any updates to L2 will be reflected in L1D cache, and vice versa. There is no guaranteed coherency between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

The TCI64xx and C64xx devices do not support automated cache coherency because of the power consumption involved and the latency overhead introduced. Real-time 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.

This coherency can all be managed through software. For shared L2 cache (SL2), coherency is not maintained between cores, so for any shared writeable data sections, the user must manage coherency as with external memory. If a section of the SL2 is used for writeable data by a single core only, then that core’s L1D cache is guaranteed to be coherent with this portion of the SL2. This is summarized in Figure 9.

The DSP/BIOS operating system provides a BCACHE module that includes API functions to perform cache coherency operations including cache line invalidation, cache line writeback to stored memory, and a writeback-invalidation operation. DSP/BIOS also provide RapidIO and shared memory Message Queue Transports (MQT) that maintain cache coherency for the application using them.

Figure 9 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 TCI6486 or C6472 device, it is important to consider the processing model. As shown in Figure 9, each core has local L1/L2 memory and a direct connection to the shared L2 memory (if present in the device), plus equal access to the external DDR2 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 in preceding sections of the document, 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.

External data sections or unique data tables within the SL2 data sections cannot simply be placed using the linker. Instead, the per-core addresses must be determined at initialization time and stored in a pointer (or calculated each time they are 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, such that 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, such that 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.

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 run-time control, it is up to the software to determine when a particular core is to initialize a peripheral. In addition, rules must be provided either at build time or system-initialization time to allow the routing of information received from a peripheral to the correct core. For example, one set of EMAC addresses are uniquely assigned to each core, a unique Utopia VPI field value is assigned per core, or a unique mailbox number for SRIO messages is assigned.

For each of the DMA resources on the device mentioned above, it is the up to the software architecture to determine 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 TCI6486 or C6472, as summarized above, all peripherals have multiple DMA channel context that allows for peer control without requiring arbitration. That is to say that each DMA context is autonomous and no considerations for atomic access need to be taken in to account.

It should be noted that because a subset of the cores can be reset during run-time, the application software must own re-initialization of the reset cores in a way that avoids interruption of the cores 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).

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 on whether to perform a global initialization or simply a local initialization.

6.5 Data Memory Placement

Data buffers may reside in any of the device memories, but typically are brought into L1D SRAM (for critical sections) or L2 SRAM for processing. Low-priority data may reside in DDR2 SDRAM and be accessed through the cache.

6.6 Data Memory Access

Memory selection for data is dependent 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. Often, however, there is a space limitation in the internal DSP memory that requires some code and data to reside off chip in DDR2 SDRAM.

Typically, data for run-time 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. In the instance that run-time data must be placed off-chip it is often preferred to page data between L2 SRAM and external memory rather than access through the cache. The tradeoff 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 Image

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

DSP/BIOS provides configuration platforms for all Texas Instruments TCI64xx and C64xx devices. In the DSP/BIOS configuration for the TCI6486 and C6472, there are separate memory sections for local L2 memory (LL2RAM) and shared L2 memory (SL2RAM). Depending 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 (all cores execute the same program), then only a single project is required for the device, and likewise, a single DSP/BIOS configuration file is required. As mentioned in the previous sections, 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 DDR2 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 Image

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 DSP/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 Image with Shared Code and Data

In this scenario, a common code image is shared by different applications running on different cores. Sharing common code between 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 for 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, and 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. Memory requirement may increase.
- Trampolines are disabled. All code needs to be within a 21-bit boundary.
- .cinit and .pinit can not be placed in the partial image.

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

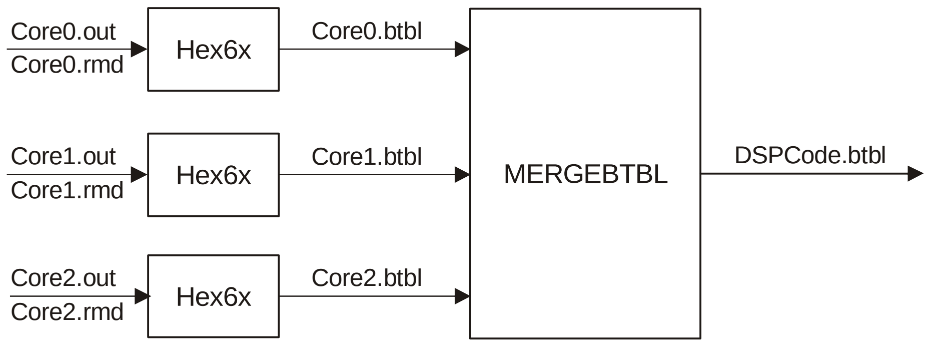
Because the DSP/BIOS code contains data references (.far and .bss sections), these sections need to be placed in the same memory location in non-shared memory by the different applications that will link with this partial image. To allow this to work correctly, each core must have a non-shared memory section at the same address location. For the TCI6486 and C6472, these sections must be placed in local L2 memory for 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 10 contains an example of how these utilities can be used to build a single boot table from three separate executable files.

Figure 10 Boot Table Merge



Once a single boot table is created, it can be used to load the entire DSP image. As mentioned in previous sections, 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 a single 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 properly loaded in the memory map to all cores' start addresses (which is configurable).

Details on the boot loader are available in TI user guides SPRUEA7, *TMS320TCI648x DSP Bootloader* [3] and SPRUG24, *TMS320C6474 DSP Bootloader* [4].

8 System Debug

The Texas Instruments TCI6486 and C6472 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 a single timeline.

8.1 Debug and Tooling Categories

There are hardware and software tools available during runtime that can be used to debug a particular 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 addressed 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. run-time), 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 run-time 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 occurring 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 real-time 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 debug of problems found during development or even problems that arise in a deployed 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 to memory with an ID, timestamp, and any parameters of interest.
- **Statistics log:** Chip-level statistics can be periodically captured to provide a picture of the activity going 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, though 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, though 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, though 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 stand-alone 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 time stamp 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 on the processed information at various points in time of 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 statistics registers built into them 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 window of time.

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 be recorded as well, 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 that each holds a time value to allow correlation with the others.

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 TCI64xx and C64xx 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 time stamp 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 in order to put together 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 off of 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 (e.g. 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. Since 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 time stamp 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 are referenced to the PC value of the core that is performing the trace function. Given that each core can stall independently from 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

9 Summary

In this paper, two 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 TCI64xx and C64xx multicore processor families used for data transfer, communication, resource sharing, memory management and debug are explained.

TI TCI64xx and C64xx 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.

TI's multicore architectures deliver excellent cost/performance and power/performance ratios for customers requiring 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 reference guide SPRU871, *TMS320C64x+ DSP Megamodule*
http://www.ti.com/lit/pdf/spru871

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

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

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless-apps