Wind River | *Technical brief*

VSPWorks™

The RTOS for DSP and ASIC Cores

The VSPWorks real-time operating system (RTOS) supports a broad range of applications. Its architecture provides inherent scaling of the kernel to complement the complexity of the application. The virtual single processor (VSP) model greatly simplifies the challenge of distributing an application across multiple processors while retaining hard real-time performance. Use of VSPWorks ensures greater maintainability and portability of the application code while lowering engineering support costs.

WindRiver®
HOW SMART THINGS THINK™

VSPWorks is a versatile, powerful software package for developing real-time applications for single- and multiprocessor platforms. It has been designed for true real-time performance and provides preemptive multitasking and high-speed interrupt support on a range of DSP and ASIC core processors using innovative kernel architecture.

Multiprocessor configurations can benefit from VSPWorks' virtual single-processor (VSP) model. Data objects and tasks can be moved from processor to processor transparently, with VSPWorks handling all the underlying inter-processor communication. Changes to the processor topology do not affect the deterministic behavior of the application. This unique design allows a multiprocessor system to be programmed as if it were a single processor. Objects defined on one node, such as semaphores and mailboxes, can be accessed from any other processor in the network, as though they were local to it.

The virtual single-processor model (VSP) combines the power of parallel processing with the simplicity of traditional multitasking programming.

The VSPWorks kernel features a multilayered design allowing development with the highest level of abstraction and portability, while implementing systems with optimized low-latency performance. Two kernel levels are dedicated to handling interrupts. One level is for lightweight tasks called nanokernel processes. The higher level of abstraction provides priority driven, pre-emptive tasking for C tasks. Code written at this task level is portable between VSPWorks systems running on different platforms or target processors. This rich feature set is available without comprising VSPWorks' key design parameters of providing a small footprint kernel with very low context switch times.

The VSPWorks kernel is modular. At compile time, the system definition tools automatically strip out all unused parts. The result is an application-specific, real-time operating system (RTOS) with minimal memory overhead, tailored to each processor in the system. A minimal configuration can be as small as 200 words on a typical device and scales to approximately 10,000 words for a complete, full-featured configuration. The average application typically uses between 2,000–3,000 words.

A suite of graphical tools to simplify and accelerate single- or multiprocessor application development complements the VSPWorks kernel. The system generation tool provides a graphical way of configuring the processing nodes in a system, greatly simplifying the job of defining where code and data are stored and executed. Other tools in the project manager help automate the process of building an application and loading it to target processors for execution. The task-level debugger provides information about all the tasks and kernel objects in the system at runtime, and the event monitor captures a system behavior log, including timestamps. A host server manages all communications between the target processor(s) and the host. During development, the host server manages the interface between the tools task-level debugger and the target. Once development is complete the host server can be built into the final application, providing control and communication capability from either another embedded application running the VxWorks® RTOS or the host development platform.

VSPWorks architecture

VSPWorks has been designed for use in DSP environments – where speed of response to interrupts is usually critical – and scalable multiprocessing environments – where ease of programming is important. The requirements for each of these environments are quite different, but VSPWorks' layered approach allows developers to choose the appropriate combination of speed and ease of use for each section of the application.

DSP programmers have traditionally had to work in assembly language at the interrupt level to get the required performance. However, programming in this way makes it virtually impossible to write large multitasking applications. Conventional RTOSs have provided the scalable, multitasking required by large applications, but have traditionally been too slow at handling-time critical interrupts.

With parallel systems the situation is even more complex. This is because the interprocessor communications require even more interrupts to be handled at the system level, so short interrupt latencies are critical. For example, on a Texas Instruments (TI) C40 device, more than 10 different sources of interrupts can be generated even before any application-specific hardware is connected.

VSPWorks solves this problem of combining fast interrupt handling with scalable multitasking by using a multilevel approach. At the heart of the system is a highly optimized nanokernel that can manage a range of processes, switching rapidly between them as required. Below the nanokernel are the interrupt service routines (ISRs) dedicated to high-speed interrupt handling, while the microkernel sits above the nanokernel and handles pre-emptive multitasking C/C++ tasks.

Interrupt handling – levels 1 and 2

Processors vary significantly in their response to an interrupt; some immediately disable all other interrupts while others leave them enabled to allow higher priority interrupts to be serviced. VSPWorks uses two ISR levels (called ISR0 and ISR1) on processors that disable interrupts, such as TI DSP devices, to allow prioritized interrupt handling to occur. On processors that support prioritized interrupt handling in hardware, such as the ADI SHARC, only

one level is required (equivalent to ISR1). Obviously, interrupt handling is always processor specific, to some extent, but the basics are outlined in the next paragraphs.

The lowest level of interrupt handling is ISR0, which is used to process interrupts coming directly from the hardware. During processing of ISR0 interrupts all other interrupts are disabled (in all other levels of VSPWorks, interrupts are enabled). Processing an interrupt at the ISR0 level can be very quick, typically less than a microsecond on a C40. The recommended technique is to acknowledge the interrupt at this level and pass the processing of the interrupt on to one of the higher VSPWorks levels. This minimizes the time during which global interrupts are disabled and allows bursts of interrupts to be handled at rates up to 1MHz on a C40. Code running at this level must be written in assembly language, and it is the programmer's responsibility to ensure the appropriate registers are saved on the stack. VSPWorks provides macros to simplify this procedure. If the amount of processing required by the interrupt is small, then it can be done at the ISR0 level. However, if the interrupt processing is more complex – and hence takes more time – then it is better to move the processing to the ISR1 level, where interrupts are globally enabled.

The ISR1 level can be entered from the lower ISR0 level through a system call. The level handles the interrupt in the same way as ISR0, but with global interrupts enabled. Normally the ISR would signal an event, which could then be processed by a waiting process or task. By re-enabling the global interrupts, the ISR1 level allows the nesting of ISRs.

Nanokernel processes – Level 3

The nanokernel is a unique feature of VSPWorks that makes it particularly suitable for DSP environments. Nanokernel processes are usually written in assembly language with a reduced context (that is, using fewer processor registers), which can be swapped in and out of the processor very rapidly – typically around 0.5 microseconds on a SHARC. Each process is assigned a priority, which dictates when it is scheduled to run. Several types of data objects called channels are available for synchronization and communication between nanokernel processes. Level 3 is ideal for writing device drivers for low-level hardware interfaces. A comprehensive assembly language API is provided to simplify accesses to the nanokernel data objects. Nanokernel processes can also be written in C, and they use the C API to access nanokernel objects such as semaphore channels and linked list channels.

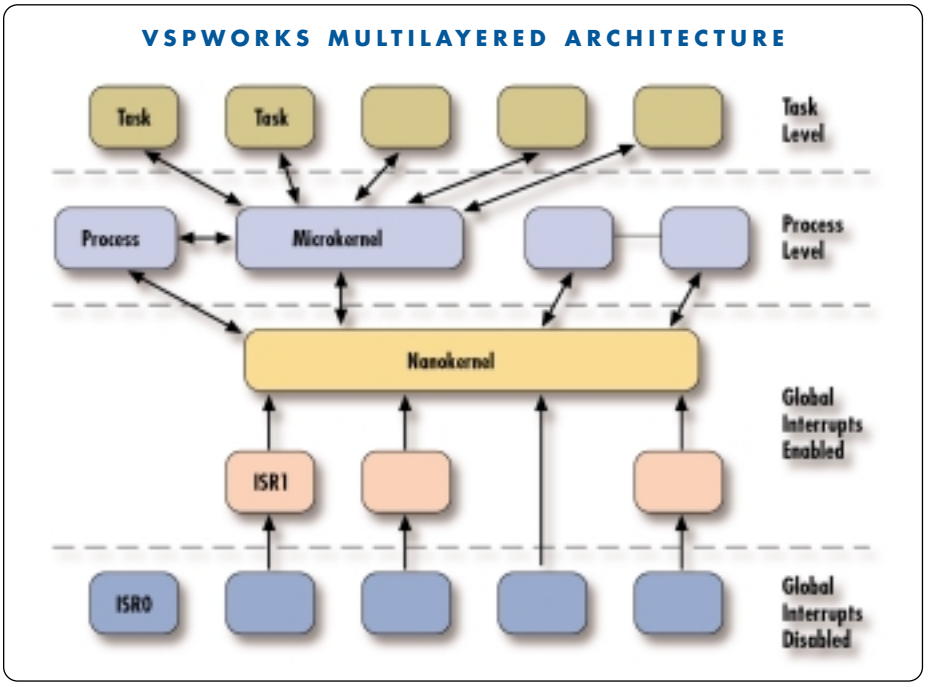

Figure 1: VSPWorks' architecture permits high-level application design without compromising performance

Microkernel tasks – Level 4

Microkernel tasks are written in C and can access over 100 kernel services. Tasks at this level are fully pre-emptive, and scheduling is priority-driven. This means that a high-priority task that becomes ready to run can immediately take over the processor from a lower priority task. Applications are built as a collection of tasks, each with a complete processor context, that communicate and synchronize using the microkernel objects. Programming an application at the microkernel level is very flexible; the code can be ported to other processors, or processor network topology can be altered with no change to the source code required. A typical application may still need some code to be written at the lower levels where a faster response or higher performance is important. However, the bulk of an application should be written using the microkernel interfaces to take full advantage of the portability and scalability they offer.

Scheduling

Most RTOSs simply offer task- or ISR-level code, but VSPWorks' layered approach provides the programmer with a wider range of options. The lower levels always pre-empt the higher levels and operate more efficiently due to the smaller context switches required. Figure 1 illustrates the interaction between the different VSPWorks levels. The lower levels have a higher priority than the layers above, and can pre-empt any activity operating at a higher level.

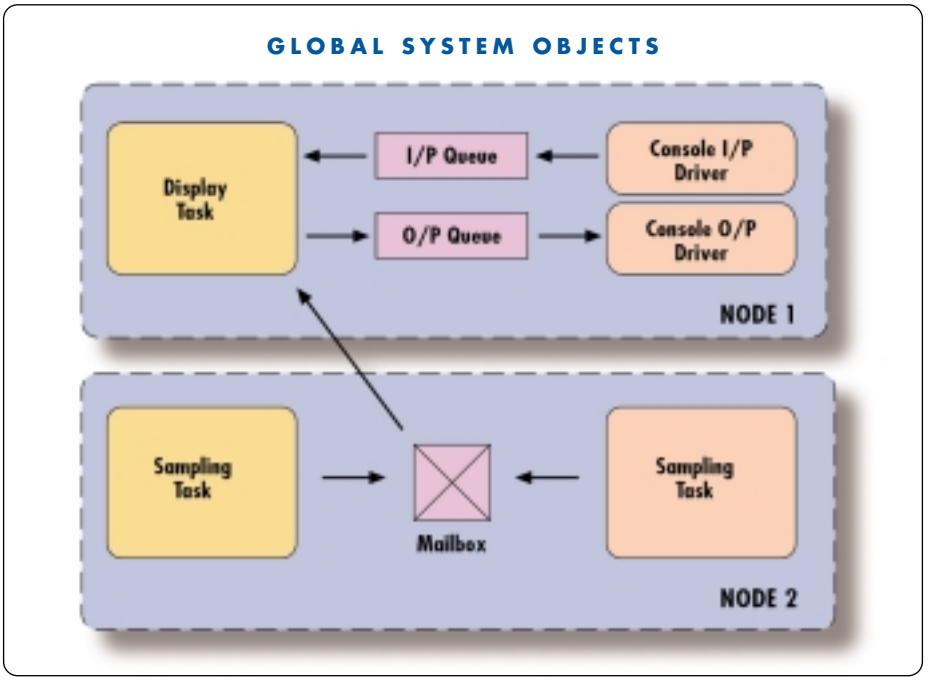

Figure 2: VSPWorks tasks and objects in a multiprocessor environment

However, interrupts may become globally disabled during ISR0 processing, depending on the developer's programming methodology and the particular processor used.

When several tasks are competing for the processor, the system must find a way of allocating the processor fairly among the tasks so that each can meet its deadlines. The solution is to assign a priority to each task, based on its relative importance within the system. The programmer can assign these priorities in several ways. One approach is to assign high priorities to tasks that need to respond rapidly to external events and low priorities to tasks that are not time-critical. Another would be to assign high priorities to tasks that need to run most often and lower priorities to tasks that run rarely. Once the priorities have been set initially, there is normally some hand tuning required at the debugging stage to get the system running most efficiently. VSPWorks tasks' priorities can also be altered dynamically at runtime, using a microkernel service call to cope with special situations occurring in the application.

Tasks are granted execution time on the processor in strict order of hierarchy – the scheduler gives access the processor only to the highest priority task that is ready to run. While a task is running, it may be interrupted and some action within the ISR may cause a higher-priority task to become ready to run. The lower-priority task is temporarily suspended and the higher-priority task starts to run in its place. Eventually, control returns to the lower-priority task, which resumes at the original point of interruption. If no tasks are in a ready-to-run state – they are waiting for external events, semaphores, and so on— then a system task called the idle or null task runs on the processor. This task simply soaks up the spare processor capacity, and the VSPWorks workload monitor can assess the loading on each node in the processor network.

Kernel objects

VSPWorks provides the real-time system designer with the ability to organize the functions of an application as a collection of communicating tasks. These tasks synchronize and communicate with each other through a set of microkernel objects. The microkernel objects are split into a number of classes that handle semaphores, queues, linked lists, and so on. Figure 2 shows the relationship between a number of tasks and the way they use kernel objects to synchronize and communicate.

In a single-processor system, these tasks and objects would be stored on the same processing node, but in a multiprocessing environment VSPWorks' VSP technology allows them to be assigned to any processor in the system. Obviously, some tasks that require specific external hardware can only be assigned to the processing node for which they were designed. Tasks and data objects can be moved from one processor to another by simple changes to the system definition file; no changes to the source code are required.

Tasks

Each task in a VSPWorks application is an independent module that performs a well-defined function or set of functions. Although a task exists independently of other tasks, it probably talks to other tasks using semaphores, message queues, and so on. A task starts running on a processing node when the scheduler determines that the resources required by it are available – in other words, it is not waiting – and it is the highest-priority task in this state. Once a task starts running it has complete control of the processor, but the multitasking kernel gives the impression that several tasks are all running simultaneously. This is achieved by continually switching between tasks to keep the processor busy on each node.

Obviously, it is impossible for a single processor to run several things at once (it is a sequential device) but the kernel scheduler can interleave separate tasks onto the processor to give the impression of concurrent operation. Once a task has been assigned to run on the processor, it remains running until one of the following occurs: the task finishes; the task has to wait for an event or data; the task has to wait for a resource to become available; the task is

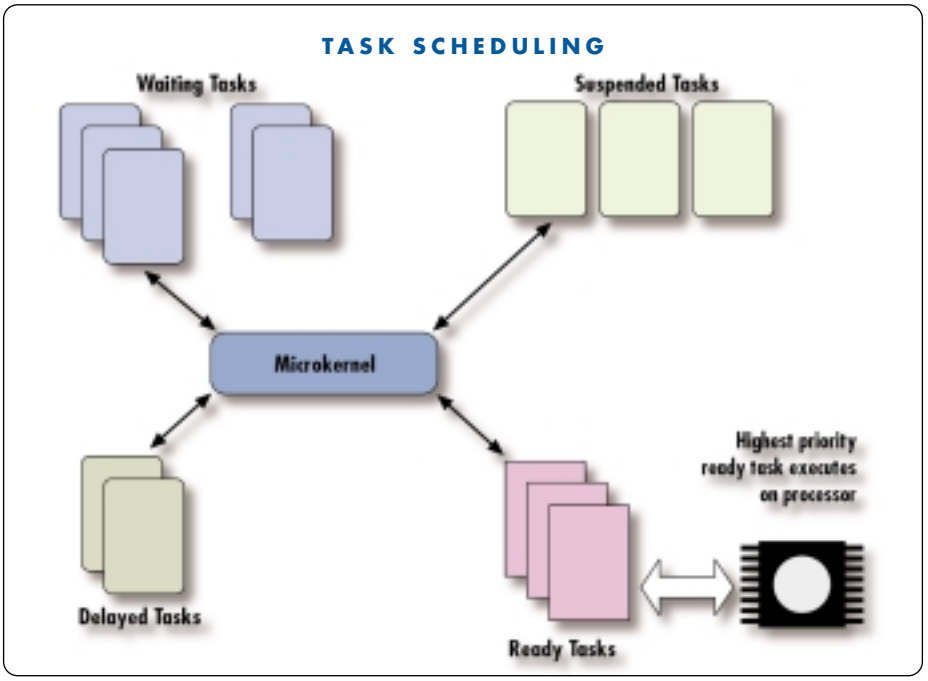

Figure 3: The microkernel manages a number of task queues for scheduling

interrupted by an ISR.

As soon as the running task stops to wait for an event, data, or a resource, the scheduler can assign another task to the processor. This produces a very efficient system, since the processor is always kept busy with tasks that are ready to run. Figure 3 shows a typical VSPWorks application during execution. The microkernel operates by managing queues of tasks. For example, there is one queue for tasks that are ready-to-run while another queue may contain all the tasks waiting for a particular semaphore. As the application executes, tasks are moved between queues depending on their current state such as ready or waiting for a message.

Counting semaphores

Tasks use counting semaphores primarily for synchronization; that is, indicating that something has happened in the system. Each counting semaphore has an initial count of zero. A task or ISR signals the semaphore, which increments the count value. Another task can wait for the semaphore to be signalled by calling a KS_SemaTestXX() operation on the semaphore. The waiting task can optionally choose to wait with a timeout or return immediately if the semaphore has not yet been signalled. Groups of semaphores may also be signalled and tested simultaneously, using the KS_SemaXxxxxG() API calls.

Events – binary semaphores

Events are simple semaphores that can be used to synchronize between two tasks or between an ISR and a task. An event can only have two states: signalled and not signalled. A task can wait for the event to be signalled with a timeout, or it can choose to wait indefinitely. An application may install an event handler, which can be used to buffer information. If installed, the event handler is called every time the event is signalled, and may then choose whether to signal a waiting task. For example, a developer may have an I/O device that generates an interrupt every time a word of data is available and an associated ISR that signals an event. An event handler could be installed to buffer 256 words from the I/O device – that is, 256 interrupts – and only signal a waiting task when the buffer is full.

Mailboxes

A message is composed of a header and a message body. A message can be sent between tasks using a mailbox. The header contains information about the sender and size of the message; the message body contains the message data. The mailbox is used to store the information in the message header only – the body of the message is handled separately. To improve the efficiency of the system, VSPWorks uses a two-step process to transfer a message using the mailbox. In the first step, the sending task places the message header in the mailbox; the message data is not copied at this stage. When a receiving task performs a read message command on the mailbox, then the data is copied from the sending task's data area to the receiving task's data area. This could involve a simple memory copy if the tasks are on the same processor, or an interprocessor transfer if not on the same processor (only on a multiprocessor system). Turning the transfer into a two-step process means that data is copied to the receiving task only when the task is ready for it. Consequently, routing buffers on the interprocessor communications channels are not needlessly filled with data that no task requires.

Mailboxes provide automatic synchronization between the sending and receiving tasks, because the sending task resumes running only when the receiving task has collected the data. Messages can be any size, and message priorities can be assigned to allow important messages to take precedence over less important ones. The content of each message is entirely at the programmer's discretion.

Messages can also be passed asynchronously, using the KS_Post() service. A task calling KS_Post() does not have to wait for a receiving task to read the message before continuing, providing greater flexibility to the application designer. Message data that is to be sent using KS_Post() must be stored in a block from a message pool (see the "Memory management" section).

Passing a message from sender to receiver is guaranteed to work in a portable manner whether or not the sending and receiving tasks are on the same processor. However, if the sending and receiving tasks are known to be on the same processor, then a quicker (but less portable) alternative exists. The message header contains a user-defined field that can be used to store a pointer to the data being sent. The receiving task can read this field from the header and access the data directly or perform a memcpy() operation to transfer the data to its own local storage area.

Queues

Queues are used for transferring data between tasks in a buffered and time-ordered way. Synchronization is unnecessary between the sending task and the receiving task, so the sender can add an entry to the queue and continue processing without waiting for the receiver to read the data. For example, one task could collect blocks of data from an I/O device and place the blocks in a queue. Another task (probably of lower priority) could remove the blocks of data from the queue and process them. Provided the queue never gets full, this scheme provides some "elasticity" in the processing requirements of the system.

Queues can also be used as ports to communicate between I/O devices and processing nodes. For example, any processor node can write to the host console simply by queuing the data on the console queue. Accesses to a queue are normally protected by a resource (see the "Resources – mutex semaphores" section) to ensure the operation is not corrupted by another task trying to simultaneously access the queue. The queue services available in the microkernel are described in the following sections.

Memory management

In virtually all real-time systems, tasks compete for dynamically allocated memory from a global pool (using malloc() and free() type operations). In the past, operating systems have used a range of different schemes for allocating and deallocating memory. Some are fast, some are elegant, some are both (and some are neither!). Most suffer from two key flaws that make them unsuitable for use in a hard real-time environment: nondeterminism and fragmentation.

Nondeterminism – taking an unknown and uncontrollable length of time – occurs when the allocation strategy searches a linked list of free memory blocks to find the next block large enough to satisfy the request. The time taken to perform the allocation depends entirely on whether the search finishes with the first block on the list, the last block, or somewhere in between.

Fragmentation occurs when requests for memory of different size blocks are made to a common global pool. After a while, a request for a new block may fail, not because there is insufficient memory, but because there is no single piece of memory large enough to satisfy the request. There are many schemes for garbage collection and memory reorganization, but they are all nondeterministic and therefore unsuitable for use in a hard real-time environment.

VSPWorks has two methods of managing memory, called memory maps and memory pools. New applications should be written using memory pools, as these offer the best combination of flexibility and efficiency and also offer better support for multiprocessing environments.

Memory maps

System memory can be divided into one or more memory maps, with each map made up of several logical blocks. When the application makes a request for memory, using the KS_MapGetBlock() command, a map must be specified, and the application is returned the address of a logical block within that partition. If no logical blocks are available in the specified partition, then NULL is returned. When the application has finished with the block it can return it to the map for reuse, using the KS_MapReleaseBlock() command. Note that a request for memory may still fail if there are no blocks free in the specified partition. It is the application designer's responsibility to ensure that adequate memory is defined in the system definition file to ensure this never occurs.

Memory pools

System memory can also be divided into one or more memory pools. Memory pools allow variable size blocks to be allocated, but in a way that eliminates fragmentation and still provides a deterministic response. The maximum and minimum size of blocks that can be allocated from each pool is specified in the system definition file.

In a multiprocessor environment, memory pools on one processor can be accessed by a task running on another processor, with interprocessor memory copies taking place only where absolutely necessary. This means that memory pool blocks are a very efficient way of implementing applications where data must be passed between tasks independently of which processor they are running on.

Resources – mutex semaphores

The resource class contains routines to provide protection to a critical object – hardware device, driver, message queue, and so on – using lock and unlock routines. All tasks that need to access the resource must lock the resource using the KS_ResLockW() routine before accessing it, and release it using the KS_ResUnlock() routine after use. Any task that tries to access the protected resource while it is locked by another task must wait until the resource becomes free. These routines are equivalent to the mutual exclusion semaphores found in some other operating systems.

Timers

An important requirement of real-time systems is the ability to specify time periods. These time periods may indicate the rate at which certain

events occur or they may specify the time the application is prepared to wait for an event to happen. VSPWorks timers can be created to generate a timed event at a specific moment, called a one-shot, or cyclically such as every 0.5 seconds). When the timer expires, a semaphore is signalled that in turn could release a waiting task. The name of the semaphore is specified when the timer is started.

Many microkernel calls allow the calling task to specify a timeout period during which the task is willing to wait. If the event has not occurred before this timeout period expires, then the task kernel call returns and passes an error code back to the calling task. The kernel uses timer objects internally to handle these situations efficiently.

Link drivers

Some processors that VSPWorks supports have been designed for use in multiprocessing systems and have communications channels commonly called links for transferring data efficiently between processing nodes. Examples would include the Texas Instruments C40 and the Analog Devices 21060 SHARC device. The processor topology is specified in the system definition file. Each link is listed in the file as either a NET link or a RAW link. Net links are used to transfer data between processing nodes in response to kernel requests, for example, sending messages between processors. They may not be used by the application directly. Raw links are never used by the kernel and are available for direct use by application tasks. The microkernel contains a set of routines for accessing these raw links in a generic – non-processor specific – way. The routines work with blocks of raw data, and so it is the application's responsibility to interpret the data in the correct way when it arrives.

Nanokernel classes

The previous section introduced the calls available in the microkernel. The bulk of an application should be written to interface to the microkernel level. However, in areas where speed is particularly vital to the application, such as with a device driver, some code may need to be written, perhaps in assembly language, to interface directly to the nanokernel. Code written to interface to the nanokernel is called a process. The nanokernel services offer both C and assembly language APIs, as follows:

- Services prefixed with prlo_ are designed to be called from the applications main() routine or from a microkernel service.
- Routines prefixed with _prhi_ should be called from within a nanokernel process written in assembly language.
- Routines of the form prhi_...C should only be called from within a nanokernel process written in C.

Process control

In VSPWorks a process is a lightweight task handled at the nanokernel level. A process normally uses only a subset of the processor's registers, which means a context switch between two processes can be very quick. However, to minimize the context used, these processes must be written in assembly language, which lengthens the development process. Alternatively, processes may be written in C, with a larger number of registers used, but this lengthens the processor time taken to switch from one process to another. The nanokernel provides routines to simplify the procedure of setting up and starting processes and to allow one process to voluntarily yield the CPU to another.

Synchronization

Nanokernel processes can synchronize and communicate with each other using three different channels: semaphore, linked-list, and stack. Each channel is defined by a data structure and a collection of nanokernel routines that operate on it. Each instance of the data structure defines a separate channel. The operations available for a channel are usually:

- Wait: The process waits on a channel.
- Signal: The channel is signalled and a waiting process is released.
- Test: The channel status is tested and a status indication returned.

Semaphore channels

The nanokernel semaphore channel is a simple counting semaphore. It is usually used by an ISR to signal a process that is waiting for an event to occur. Every time the semaphore channel is signalled, the count associated with it is incremented. This is important when two signal operations occur in quick succession. Each operation causes the channel count to be incremented and so the waiting process is released twice; no signal operations are lost.

Linked-list channels

The linked-list channel creates a linked list of memory blocks, using the first word in each block as a link pointer. New blocks are added to the linked list using a PUT command and data is removed using a GET command. The microkernel itself uses linked list channel objects to store free command packets, data packets and timers. Two versions of linked list channels are implemented, called LIFO and MWFIFO. The LIFO channel allows only one process to wait on the channel and offers the following API.

The MWFIFO channels allow multiple waiting so that several processes can simultaneously wait on a channel. The waiting processes are queued in order of priority, such that the highest priority process receive the first piece of data that arrives on the channel.

Stack channels

The stack channel uses a block of memory as a data stack. Data is pushed on the top of the stack using a PUSH operation and removed in a LIFO fashion using the POP operation. No checks are made for stack overflow, so the system designer must be sure the defined stack size is large enough.

ISR management

Interrupt service routines can be invoked at any time, caused by changes in the external environment, such as hardware interrupts, or activity within the processor, such as a communications buffer emptying. The ISR for the event is executed by the processor and may need to communicate to a nanokernel process that the event has occurred. This is usually achieved by using a nanokernel channel (the exact semantics of channel services are specific to the processor being used).

VSPWorks project manager

The VSPWorks project manager is a Windows tool (NT or 2000) that simplifies the development of a VSPWorks application. It allows a developer to open windows to display the tasks and data objects assigned to every processor in the network. New tasks and data objects can be created and assigned to a processor using the comprehensive toolbar. Objects can also be moved between processors using a simple drag-and-drop interface.

Windows can also be displayed to show the whereabouts on the system of every data object. For example, all semaphores defined in an application can be listed and the processor where they are stored is shown. Multiple windows can be displayed to show many different types of data objects simultaneously.

Once the system has been defined using the project manager, the system generation tool can be used to create all the system files and basic header and source files required by the application. The project manager can then be used to build the application and start the host server/network loader if required.

VSPWorks host server

The VSPWorks host server includes the network loader, a Windows application that can download a VSPWorks application to a target processor network, and then optionally service requests to the host from the VSPWorks application on the target(s). If enabled, it accepts commands from the target processor(s) for access to the host file system, console, and so on. In a multiprocessing system, any processor can make a request to the host server, for example, to print text on the screen. Host server requests from remote processors are routed through the processor network until they reach the root node that is in direct contact with the host. The host server can also be used to start the task-level debugger.

Task-level debugger

VSPWorks includes a standalone task-level debugger that can be used to examine the state of any VSPWorks object on any processing node. The task-level debugger can be called up from either the keyboard on the host, or by using a call within a task in the application. The debugger is especially useful for tuning the overall performance of the system, or tracing problems caused by the interaction of two or more tasks. It is not designed to replace other debuggers, such as the source-level debugger that is often supplied by the processor manufacturer, but to provide a system level view into the workings of the system.

The debugger uses the screen and keyboard of the target's host PC or workstation. Transfers between the target processors and the host can be sent using a standard host interface, based on shared memory or a FIFO, or using RS232C, depending on the type of target hardware being used. During the period when the debugger is active, all other tasks in the system are suspended, although interrupts continue to be serviced. The system clock (TICKS timer) ignores its interrupt, so the tick count is not incremented. This means that time spent in the debugger is effectively invisible at the task level. The debugger can be removed from the final system by relinking the application with the makefile's DEBUG switch disabled.

The debugger provides access to all the VSPWorks objects in the system, with relevant information about each type:

- Tasks: the task name and number are displayed together with the current priority of the task and its execution state, such as waiting for timer or waiting for semaphore. The stack usage of each task is also available, including current size, maximum size, and largest size used so far (high-water mark).
- Queues: a snapshot of all the queues (including system queues for console accesses) is shown including queue name, current queue size, high-water mark, maximum queue size, and names of tasks waiting on the queue.
- Semaphores: the names of all the semaphores, along with their current count value and the names of tasks waiting on them are displayed.
- Resources: for each resource, the number of lock requests, the number of conflicts – the number of times the resource was needed by one task while it was locked by another – and the names of tasks waiting on it are shown.
- Memory partitions: for each memory partition, the debugger displays the number of blocks currently used, the maximum number of blocks used so far, the total number of allocation/deallocation operations so far, and the names of any tasks waiting for blocks to become available.
- Mailboxes: information is displayed about the current entry in each mailbox along with the names of tasks waiting for send or receive message operations to complete.
- Timers: all user- and system-defined timers are displayed, showing time to expiration, action on expiration, such as wake up a task, and the name of the object to which the action applies, such as a kernel object or user task.

Workload monitor

The kernel provides two operations for determining the workload of a processing node. The KS_Workload() command returns a value indicating the time during which a processor node is busy, to the nearest 0.1 percent. The KS_WorkloadSetPeriod() command defines

the rate at which the workload is measured, between 10 msecs and 1 second. The workload measuring routines rely on a high-precision timer being available to the processor. Workload monitoring is based on the principle that at any given time the processor could be doing one of three things: running a kernel operation, a user task, or the null task (the processor is idle).

The null task is created automatically by the system and has the lowest priority of all tasks. This causes it to be scheduled by the kernel when there is absolutely nothing else to do. The null task loops round counting how many iterations it has completed so far. By reading the counter twice within a defined time interval, the time spent running the idle task can be determined. This value is used to calculate the processor loading when KS_Workload() is called.

Tracing monitor

The tracing monitor forms part of the task debugger and saves information in a circular buffer about system behavior. Analyzing this buffer allows the recent history of the system to be traced back, which is especially useful in the case of a system failure. The size of the buffer can be specified at system startup along with the type of information to be logged.

Once data has been recorded in the buffer, the task debugger can be used to interpret and display it. Each event is timestamped using a high-resolution timer, providing a highly accurate record of when the event occurred.

Board support packages

Each target running VSPWorks needs a board support package (BSP), and each host needs the VSPWorks host server. Wind River supplies BSPs for most leading boards and host servers for many common platforms, including those utilizing PCI, ISA, VME, and RS232C communication architectures. Please contact Wind River for the latest availability information.

A developer using a less common target board or designing his or her own board has to produce the BSP. The most important decision when writing a BSP is choosing the method of transferring data between the host and the target. Boards usually provide either a Dual-Port RAM or a FIFO attached to a processor link for host I/O. A new BSP using either of these methods can easily be produced.

The BSP consists of two sections, with part running on the target and part running on the host. The host code needs to be written in the form of some new C++ classes that are derived from base classes that Wind River provides. These new classes need only a few functions, and example source code is provided. On the target side, the BSP writer needs to generate two new C functions and an interrupt handler. Source code examples for the target functions are also provided in the standard package.

Adding user commands

The standard host server handles requests for services from the root node using a set of command packets. To extend the server functionality, a developer can define custom command packets on the target and then add extra routines to the host server to handle them. Any task on any target node can make a request for the new function by placing a new command packet on the host queue. This command packet is transferred to the host, where a new packet processor function must be written to handle it. The packet processor is created by writing a new C++ class that contains some predefined functions. The host server already includes base classes that simplify writing a new packet processor.

Stdio library

The functions in the VSPWorks Stdio library allow a task running on any processor in the network to perform standard I/O through the host. They have been designed to support the multitasking and distributed processing environment of VSPWorks and operate in conjunction with the host server program. For example, any task can use the console for printing or access the host file system for storing data.

Graphics

The optional graphics server supplied with VSPWorks can be used with a wide range of PC-hosted boards to generate sophisticated graphics on the PC screen. The routines can be called from any task on any processing node in the network, and they communicate with the graphics server running on the PC host. Application tasks generate command packets and pass them to the graphics server task running on the root node through two systemwide queues. This root task passes the commands to the host graphics server, based on a Borland graphics engine, which plots the output on the PC screen.

Briefly, the graphics commands included in the driver provide the following functions:

- Select graphics mode and display page.
- Set color palette, text style, text size, and so on.
- Draw pixels, lines, circles, and shapes (line only or filled).
- Write text.
- Load and save screen portions from/to disk.

Easy multiprocessing

VSPWorks is available in two versions: single processor (SP) and virtual single processor (VSP). The SP version is tailored for single-processor targets, which do not need the distributed processing features of VSP. On an SP system all tasks and data objects are automatically on the same processor, so there is no requirement for moving data and synchronizing tasks between processors. However, in a multiprocessor environment, the VSP model includes extra kernel code for transparently moving data and command packets around the processor network. For example, this allows a task on one processor to signal a semaphore stored on a second processor, which is being tested by a task on a third processor.

VSPWorks is designed for use on a wide range of processing topologies, from a single embedded processor with no host communications, to a large multiprocessor network controlled by a host typically running an OS such as UNIX or VxWorks. The VSPWorks programming philosophy is the same either way, with the emphasis on portability combined with maximum speed of operation. Applications originally written for a single-processor environment may be transferred quickly and simply to a multiprocessor network, because of the common microkernel API in the SP and VSP versions. From a software development standpoint, the VSP model allows a network of processors to be treated like one powerful processor. The application programmer simply codes the application as if it were running on a single processor, and VSPWorks takes care of the interprocessor communications. Tasks and other kernel resources are simply assigned to processing nodes in the network during system configuration, and data objects such as semaphores and message queues are automatically accessible to all processors. All the data copying required to allow this model to work is performed by the kernel in the background, leaving the programmer free to concentrate on the real application. Changes may be made to the processor network topology or the assignment of tasks among the processing nodes, without any changes to the source code.

Routing

The VSP model relies on a systemwide naming scheme for microkernel objects such as semaphores, queues, and even tasks themselves. This allows the programmer to treat the processing network as a single real-time processing unit. Whenever an access is made to a microkernel object, VSPWorks verifies whether the object is available locally to this processing node or whether it is stored on a remote node. If the object is available locally, then the operation on the object is handled just as though it were a single-processor system. If it is not available locally, then the microkernel service request is transformed into a remote procedure call, which must be sent to the appropriate processing node. The microkernel contains an embedded router, which can locate the processor where the resource is located and determine the shortest path to reach the destination with a given communications packet. If several equally optimal paths can be used to access the remote processor, then data is sent in parallel down all of them to minimize the transfer time. In a large system, the information may need to be forwarded by several processors in the network before reaching the final destination.

The router manages a pool of message buffers that are dynamically allocated to store command packets when required. Each message buffer contains a priority field, whose value is copied from the priority of the task that is requesting the service. This priority inheritance means that high-priority message transfers take precedence over messages already queued by low priority tasks. The router is used for all system data transfers, including remote memory operations, so a memory copy is handled in the same way as routing a message. The speed of message transfers is comparable with the speed of calling a microkernel service; in a typical system, tasks are not held up waiting for message routing to occur. The benefit of this is that the programmer can view the memory of the processing network as one large common address space.

The VSP model for application development is so powerful that VSPWorks can run on systems with a mixture of processor types. The underlying message protocols are common between different processors and so the only requirement is the presence of a fast communications method. This could be shared memory (such as across a VME backplane), a serial link, or even dual-port RAM. This feature is particularly useful in systems that have to perform a wide range of functions and use the most suitable processor for each type. For example, a system may have a basic microprocessor at its core, a DSP processor for computationally intensive tasks, and an I/O processor for communications with the outside world. VSPWorks could be run on each processor to provide a uniform programming interface on each device while handling all interprocessor communications, dramatically shortening the development timeframe.

For more information

Please contact a Wind River sales representative for additional information and a demonstration.

Wind River Worldwide Headquarters
500 Wind River Way
Alameda, CA 94501 USA
Toll free 1-800-545-WIND
Phone 1-510-748-4100
Fax 1-510-749-2010
Inquiries@windriver.com
Nasdaq: WIND

For additional contact information,
please see our web site at www.windriver.com.