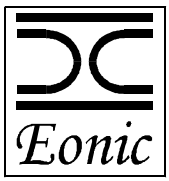


Virtuoso user guide: book 1
for Version 4.1

Introduction to Virtuoso
and crash course in programming
Virtuoso applications

This page intentionally left blank

Contents

This book starts you off working with Virtuoso. If you finish it, you'll have a good grasp of what it takes to write Virtuoso applications.

This page intentionally left blank

Chapter

1 Introduction

Book 1 of this **Virtuoso user guide** is for new users. It starts with a list of the documents and other help available to you as a Virtuoso user and a customer of Eonic Systems.

If you haven't installed Virtuoso yet, it's probably a good idea to do it now, so you can follow what's in the guide. The **Virtuoso installation guide** will help you install it correctly.

In chapter 2, there is a crash course in developing simple applications. At the end of the course, you'll have code that uses some of the more basic, but interesting, features of Virtuoso, and it will work!

You might want to look at the other books in the user guide:

- Book 2 is a programming reference
- Book 3 contains configuration and file information
- Book 4 is a processor supplement, which contains essential programming and configuration information for the processor you are using

It is essential that you read book 4!

Finally, if this is your first taste of Virtuoso, you might want to browse through the glossary at the end of this chapter. The glossary is a list of terms that you may not be familiar with, or which are used in a specific way in Virtuoso.

1.1 What you get with Virtuoso

- **License details**
- **Virtuoso installation guide**
- **Virtuoso user guide** with a processor supplement describing how Virtuoso works with your specific processor/board (if it is a board we support)
- **CD** containing Virtuoso software, examples and documentation

1.2 License

Virtuoso will not run on a networked system without the license manager software. See the **Virtuoso installation guide** for details.

1.3 Help for Virtuoso application developers

There are several kinds of help available for you if you're developing applications with Virtuoso.

Online help

There are help files for all Virtuoso components, including the kernel services. The help files are in the **VIRTDIR\bin** folder and are accessible from Virtuoso tool windows using the Help option and the F1 key.

Hard copies of the guides

These guides are included:

- Virtuoso installation guide
- Virtuoso user guide
  Book 1 Introduction and crash course
  Book 2 General programming reference
  Book 3 Configuration and file reference
  Book 4 Board-specific programming reference
- Virtuoso Host Extension Kit (HEK) user guide
  This is included only if you request the HEK

Soft copies of the guides

Acrobat versions of the user guides are in the **VIRTDIR\docs** folder. You will find a self-extracting archive of the Acrobat reader in the **VIRTDIR\Acrobat** folder.

Eonic support

If you have a problem, please try the help files or user guides first. If they can't help, you can call on Eonic support. Contact details are on page 2 of this guide and in the help files. We will acknowledge receipt of your problem report, and will normally reply within 1 working day. Before you contact us, make sure you have these files ready to describe or to send to us:

- project file (.vpf)
- NLI file (.nli)
- source code (including C and header files)

Please get the information for this form. We use the form in our problem tracking system – filling it out will help us to solve your problem quicker.

About you:
Your name
Your organization
About the product you have a problem with:
Name
Version Serial number
About your host OS
Name Version
About your DSP hardware
Vendor/board name
No of processors
Links
Interface
A brief description of the problem
Error messages

1.4 Virtuoso tools

As an application developer, you will be interested in these Virtuoso tools:

- Virtuoso Project Manager (VPM)
- Virtuoso Host Server
- Virtuoso Task Level Debugger
- Virtuoso Tracing Monitor
- Virtuoso Host Extension Kit (if ordered)

The Eonic admin, control and query options that you'll find installed relate to license use. See the Virtuoso installation guide for details.

1.4.1 Virtuoso Project Manager

The Virtuoso Project Manager is mainly for creating and editing Virtuoso project files (VPFs). These are the software configuration files you need to supply with the source code in order to generate a Virtuoso application. VPFs replace the sysdef files used in previous versions of Virtuoso.

You can also use the project manager to:

- edit source code, using the built-in editor or another one of your choice
- build Virtuoso applications
- call the Virtuoso Host Server to download and run the completed application

See book 2 of the user guide for more details of the project manager.

1.4.2 Virtuoso Host Server

You use the Virtuoso Host Server to download your application to the target board(s) and execute it. The Host Server can also accept requests for host services from the application running on the target processors (for standard I/O or graphics output for example). See book 2 of the user guide for more details.

If you are writing a hostless application, you can still develop it on your PC with the Host Server. When development is complete, a small edit of the [root]netload function will make it a hostless system. See “ROM booting” in book 3 for more details.

1.4.3 Virtuoso Task Level Debugger

With the Virtuoso Task Level Debugger, an application running in debug mode can be paused to take a snapshot of the system objects. The information provided can help you find out if the application is running efficiently.

Each tab in the debugger window monitors usage of a class of system objects, such as tasks. The task monitor shows the stack usage of all the tasks running on the target board at the point the application was stopped. You’ll find the debugger great for looking at resource and queue usage when your application appears to hang or run slower than you want.

See book 2 of the user guide for more details.

1.4.4 Virtuoso Tracing Monitor

While the application is running in debug mode, information about semaphore signaling, resource locking and other system operations is saved in a circular buffer. When the application is paused, this information is displayed by the Virtuoso Tracing Monitor. You can see the duration of all transactions occurring in the system, and the connections between them. The transactions are grouped by node, by task and by time.

The Tracing Monitor is installed as part of the Task Level Debugger. The buffer size for the Tracing Monitor, and the types of transactions it records are set in the project file.

See book 2 of the user guide for more details.

1.4.5 Host Extension Kit (HEK)

If you want to extend the functionality of an existing Host Server, you need to write a host service module using the Host Extension Kit (HEK). See the HEK Manual for more details.

You can use the HEK to:

- port the Virtuoso Host Server to different host OS’s
- write board support packages (BSPs)
- add host service modules (HSMs)
- integrate the Host Server into an application

The HEK is supplied free of charge to Virtuoso customers, but it is not part of the standard Virtuoso package, and must be ordered separately.

For more information, contact us. Contact details are on page 2 of this user guide.

1.5 Glossary of terms

Application kernel

Formerly known as the microkernel.

The application kernel provides the main C programming API for Virtuoso, offering a number of *services* and data structures for use by *tasks*. The application kernel is a *process* controlled by the *system kernel.*

Architecture file

Also known as a linker command file, linker description file or linker definition file.

You must create an architecture file to support your application. The architecture file defines the layout, distribution and heap size for the processor's memory.

For initial testing, you can copy any architecture file from the Virtuoso examples folder, but an optimal configuration for a particular application can only be reached by trial and error

See book 3 of the user guide for an example of an architecture file.

Blocking

Putting in a waiting state, either by descheduling a *task* or by putting it in a queue waiting for a system event.

Board drivers

The *target boards* Virtuoso supports are listed in the Virtuoso Certified Configuration (VCC) list – check the Eonic web site http://www.eonic.com/ or mail info@eonic.com for the latest list.

Unfortunately it is not possible to make an installation procedure that will work with all boards. The VCC list shows whether to use a vendor driver or the Eonic-supplied *WinRT* driver.

Board support package

Board support packages are used for interfacing Virtuoso DSP systems with the *Virtuoso Host Server*.

Eonic Systems already supports many *target boards* from companies such as Blue Wave Systems, TI and others – check the Eonic web site http://www.eonic.com/ or mail info@eonic.com for the latest list. However, if you are using a less common target board, or you are designing your own, you may have to write a board support package (BSP). The BSP contains information for direct booting.

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 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 using Eonic's Host Extension Kit (HEK).

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 new C++ classes that are derived from base classes provided by Eonic. Only a few virtual functions need to be implemented for these new classes of the custom board: the functions are specified in the template source code 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 HEK package.

Board vendor utilities

Utilities provided by the manufacturer of the board. These usually include drivers for accessing the board, a C library for use in your own host programs, monitoring and testing software, and software to download executables onto the board.

Bootlink

A description of a link between two processors, A and B, that can be used to boot processor B when only processor A is connected to the boot device (JTAG or host). For example, SHARC processors can be bootlinked from link port 4. A bootlink is not necessarily used as *netlink* after booting.

Bootlinks are also used as a means of synchronizing the network. Even when all the *nodes* in the system are booted directly, bootlinks must be declared in order to specify the path of the boot packet.

Byte

A byte is assumed to be 8 bits.

Channel

The means of communication between *processes*. See book 4 of the user guide for more details.

COTS board

Commercial off-the-shelf board, ie. a commercially available board, as opposed to a custom board.

Dependency

Used in explanations of the *makefile*.

If a file is a dependency, it means that the *target* will be rebuilt if the timestamp of any dependent file is later than the timestamp of the target built by the previous make.

Direct booting

Booting the processor directly from the boot device (JTAG or host) rather than via *bootlinks*.

Driver

Drivers are low level programs that provide generic or particular access to hardware. They have an associated startup program that installs any *ISRs*, starts *processes* and so on. The startup program must be declared in the *project file* driver definition.

The default drivers are declared in nodetype.h, which is included in the node#.c *intermediate file* produced by Virtuoso's generate utility. So you do not normally need to include the nodetype.h separately.

The default drivers are described in the processor supplement in book 4 of the user guide.

Event

An event is a system *object* similar to a *semaphore* except that it has two states, high and low, and is local to a *node*. Only one *task* can be associated with an event. An event handler (a C function) can be called when the event occurs. The handler cannot call any kernel *services*, but it can indicate that the task waiting on the event is to be rescheduled.

Event signaling is the simplest and most efficient way to *synchronize*, and is the preferred way for an *interrupt service routine* to communicate with the *application kernel* or with a task.

Execution

Executions are transactions that occur when Virtuoso starts processing *requests.*

Fifo

Formerly known as a **queue**.

A fifo is a system *object* suitable for transferring small amounts of data, such as *task* control information, in an asynchronous, buffered and time-ordered way. Fifo entries must be of a fixed size, with a maximum size of ten 32-bit words.

A fifo might be used by a task filling an area of memory with data, for example, to tell the task processing the data that the area is full. The processing task could use another fifo to tell the memory filling task that the data has been retrieved and that the area can be cleared.

For most purposes, we recommend the use of *mailboxes* over fifos because they are more flexible, but the choice depends mostly on the size of the data to be transferred.

File Generation directory

The file generation directory is where all generated code files are stored. It defaults to the directory containing the *project file*. You can change it using the menu option (Tools | Options) in the main window of the *VPM.*

Host Extension Kit

A package supplied separately from Virtuoso that you can use to:

- write *board support packages*
- port the *Virtuoso Host Server* to different host OS's
- add *host service modules*
- integrate the Host Server into an application

The HEK contains libraries, header files, templates and examples. For more information mail info@eonic.com.

Host Server

There are two versions of the Virtuoso Host Server. One is started from the command line, the other is started from the *project manager*. Both versions have the same functions:

- resetting the *target board*
- booting, ie. downloading executables to the target board
- hosting, ie. handling interactions between the target board and the host, such as standard I/O and graphics output to the host screen

The Host Server has four cooperating aspects:

- a generic core
- a *processor support package*
- a *board support package*
- *host service modules*

Eonic Systems supplies a standard Host Server for each supported board – check the Eonic web site www.eonic.com/ or mail info@eonic.com for the latest list. Customers can write their own board support packages and host service modules using the Host Extension Kit.

Host service module

A host service module is a part of the Host Server that supplies specific services. For example, the Virtuoso implementation of standard I/O is written as a host service module, as is the graphics interface.

Intermediate files

The Generate Files option in the *project manager* uses the *project file* (VPF) to create intermediate files: a source file (node#.c) and header file (node#.h) for each *node* in the system, and another header file called allnodes.h.

These files provide the system startup routines for each node in the processor network, and define the data *objects* on each node as required. The node#.c files contain the main() routine which initializes the node, *synchronizes* it with the rest of the system and starts up user *tasks* in any *Task Groups* that were defined to startup at boot in the project file definition (the EXE *Task Group* always starts at boot).

Interrupt service routine

An ISR is a function that is called when an interrupt arrives from a hardware device. ISRs are normally written in assembly language. In Virtuoso, ISRs can use *events* or *semaphores* to *synchronize* with application kernel *tasks.*

Linkbooting

Booting your processor using links between processors, instead of direct booting. For example, an Analog Devices Sharc can be booted from Sharc link port 4. Linkbooting is supported by the *Virtuoso Host Server*, and can be supported in ROM booting.

Makefile

Makefiles are standard files that are used in building the processor executables from your source code. They supply configuration and other information to the compiler and linker, and contain a number of rules about creating *targets*. The rules define the dependencies of the target and the tools and options used to make the target.

There is a reference document for creating makefiles at the Computer Science Department of Aberdeen University, UK:

http://www.csd.abdn.ac.uk/facilities/sw/texi2html/make_toc.html

Virtuoso's makefile information is held in three separate files:

- VIRTDIR\<processor>\<compiler>**Base**
  This contains the processor and compiler makefile requirements
- VIRTDIR\<processor>\<compiler>\<board>**Tools**
  This contains the board-specific makefile entries
- VIRTDIR\<processor>\<compiler><board>\<category>\<program>**makefile**
  This contains application-specific makefile entries. Note that the name *makefile* must be all lower case

Makefiles are processed by the **make** utility, which the *Virtuoso Project Manager* calls automatically when you select the Build or Rebuild options. You can also call the make utility directly. It takes no parameters. The utility is located in the VIRTDIR/bin directory.

See book 3 of the user guide for more information.

Mailbox

A mailbox is a system *object* suitable for sending large or small amounts of data around the system. It is the preferred way for *tasks* to communicate.

A mailbox message is composed of two parts, a header and a body. The header says who the message is from and how big it is, while the body contains the message data. The mailbox stores only the header – the message data is handled separately.

Messages can be any size and may be prioritized. For faster messaging, if the sending and receiving tasks are on the same processor, then a pointer to the message data can be passed via a field in the message header

For sending small amounts of data, you could also consider using a *fifo.*

Memory allocation

Virtuoso *memory pools* and *memory maps* are designed to reduce the effects of three major problems with memory allocation routines.

The first problem is that a **time lapse** of indeterminate length can occur while the allocation function (malloc) searches for a block large enough to satisfy the request.

The second problem is that standard memory allocations are usually **not re-entrant**, and are therefore not safe to use in a multitasking environment.

The third problem is that **fragmentation** occurs when requests for memory of different block sizes are made to a common 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.

Memory map

A memory map is a system *object*, consisting of an area of memory split into blocks of a fixed size, and is local to the processor on which the requesting task is running.

We recommend that new applications are written using *memory pools* in place of memory maps.

Memory pool

A memory pool is a system *object*, consisting of an area of memory with a block size determined and allocated dynamically within predefined maximum and minimum sizes. It is local to the processor on which the requesting task is running, but blocks can be freed by tasks on remote processors.

Memory pools are the preferred method of memory allocation for Virtuoso applications.

Microkernel

The former name for the *application kernel.*

Monitor

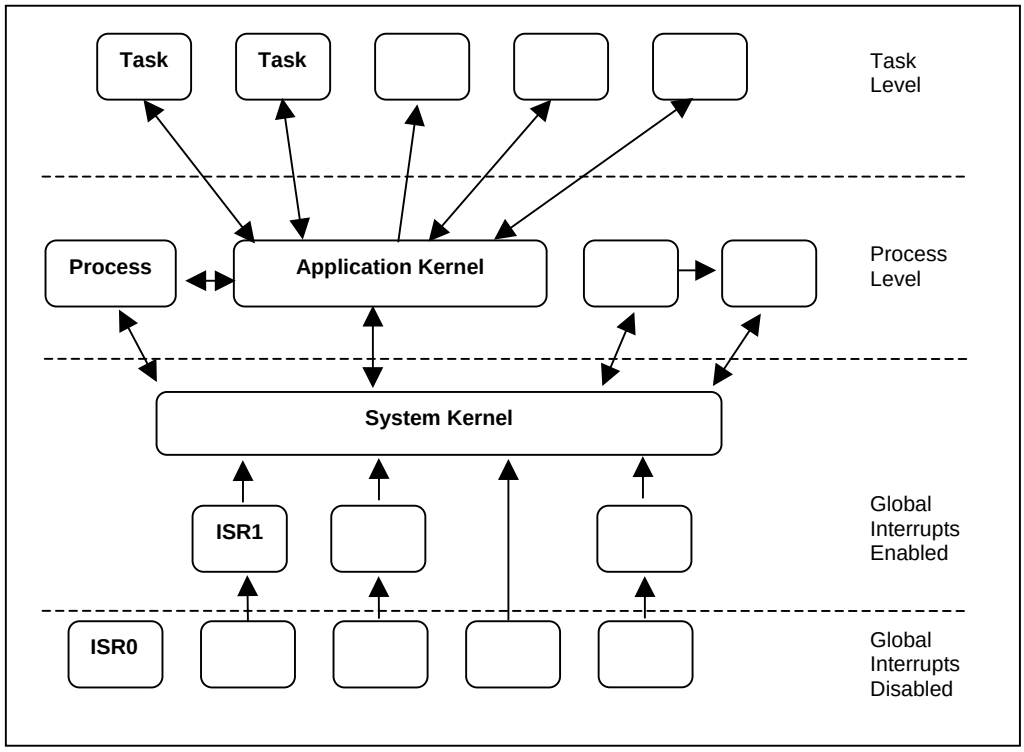
Used in explanations of the *Tracing Monitor* and the *Task Level Debugger*.

Each tab in the main window of the Tracing Monitor shows trace data for a particular node. The tabs are called monitors because they are monitoring activity on the *nodes*.

Each tab in the window of the Task Level Debugger shows debug data for a particular system *object* – tasks, fifos, mailboxes, and so on. The tabs are called monitors because they are monitoring activity on the objects.

Multi-level programming

Virtuoso *tasks* are written at the *application kernel* level. Many applications can be written entirely at this level. However, there are three other programming levels available to cater for specialized needs.



The *system kernel* manages very small, lightweight tasks called *processes*. Processes are fast but have some restrictions: non-pre-emptive scheduling; a limited number of registers; and code which is not portable. It is best to write code at this level in assembly language, although there is a set of C routines provided. Some processors do not allow programming at this level, so your system may not have the system kernel available.

Below the system kernel are the interrupt levels. Code written at these levels is strictly assembly language. Here are the *interrupt service routines* that run in response to interrupts received from hardware. They start up and run very fast.

There are two interrupt levels in a full Virtuoso system, but on your system you might only have one – this is very low level and entirely processor-dependent. If your processor doesn't support it, you don't get it.

An interrupt can be serviced at ISR0, ISR1 or even system– or application kernel levels, depending on the processor and on how time-consuming the processing is.

Multitasking

A Virtuoso application is built as a collection of *tasks*, each with its own thread of execution and set of system resources. These tasks are written in C. They communicate and *synchronize* using calls to Virtuoso kernel *services*. There are currently about 80 services.

Since more than one task is normally loaded on a processor, Virtuoso is managing several tasks at once – it's a *multitasking* system.

Nanokernel

The former name for the *system kernel.*

Netlink

A link between two processors that is used exclusively by the Virtuoso kernel for inter-processor communications when the application is running. The physical carrier for a link can be any type of hardware (e.g. twisted wire, cable, bus connector, shared memory).

Contrast with *bootlink*, *rawlink.*

NLI file

The Network Loader Information file. This file contains information about:

- the *target board* type
- the number and type of processors on the board
- the root *node*, used to communicate with the host
- *bootlinks*

Node

A node is a description of a processor on the *target board*, with its frequency, local memory, communications ports and other processor-specific information. It is defined in the *project manager*.

The **root node** is the processor directly connected to the host (usually via shared memory or a fifo type link). When the application is running, driver tasks running on the root node interface to the *Host Server*, providing services for the other processors. Tasks that need fast access to the Host Server, such as those required by the *Task Level Debugger,* should run on the root node if possible.

When booting, the root node is loaded first, followed by all other nodes. Loading code to the remote nodes is usually done using the comports or linkports.

Object

Virtuoso is a formally structured system that encourages its users to write structured applications. One of the ways it encourages structure is by using objects. An object is just a convenient term for data and code structures provided by Virtuoso that you can use in your application. Each object has a fixed interface accessible only through kernel *services*.

There are nine objects: event, fifo, mailbox, memory map, memory pool, node, resource, semaphore, task.

To use a Virtuoso object, you must define an instance of it in the *project file*. It must be given an alphanumeric name, the first character of which must be alphabetic. No embedded spaces are allowed but upper and lower case can be used. All names must be unique (ie. a semaphore cannot have the same name as a fifo, even though they are of different types). When the system is generated, the kernel objects are assigned a 32 bit identifier. The most significant 16 bits represent the node identifier, while the least significant 16 bits represent the object identifier.

Packet

Data is sent round the system in packets. The number of packets to be made available is specified for each *node* in the *project file*.

There are three types of packets:

- command packets: used by the *application kernel* for service information
- data packets: used to buffer data on **intermediate nodes only** when you transfer data between indirectly connected nodes
- timer packets: used by the application kernel for timer information

Process

A process is the *system kernel* equivalent of a *task*. It is very processor-specific - see book 4 of the user guide for more information.

Processor support package

A processor support package contains information for linkbooting specific processors and for parsing the *node* information in the *NLI file.*

Virtuoso project file

A Virtuoso project file (VPF) can be created in a text editor or by using the *project manager*. The VPF links the hardware to the software and defines all the *drivers* and *objects* used in an application. Because the distribution of *tasks* on a network of processors is defined in this file, separately from the source code, an application can be developed on a single processor and then run on multiple processors with only small changes to the project file and no changes to the source code.

The Virtuoso generate utility uses the project file (with the *NLI file* and *architecture file*) to generate *intermediate files*, which in turn are used to build the executables.

The project file syntax follows some conventions of C. As a result, you can define numbers as symbolic names, use include files, and use comments.

Project Manager (VPM)

The Virtuoso Project Manager (VPM) is an application that runs on the development host computer. It is a complete Integrated Development Environment (IDE), that enables Virtuoso users to create, edit, build, and run applications. The Virtuoso project files it creates are simple text files, so you can create and edit them directly if you want.

You use the **main window** for creating, saving or loading a project; for checking the consistency of projects; for generating the application source files; and for linking and compiling your application into executables. The consistency check involves checking that the specified netlinks enable each *node* to be reached from all the other nodes.

You use the **node view** and **object view windows** for specifying your application. Both windows allow you to create application *objects*: the node view shows all objects that live on a selected node, whereas the object view shows all objects of a selected type (semaphore, mailbox, etc).

Rawlink

A link between processors that is used exclusively by the application. The data transfer is a raw bit protocol. The physical carrier for a link can be any type of hardware (e.g. twisted wire, cable, bus connector, shared memory).

There are two rawlink services, KS_LinkInW and KS_LinkOutW.

Contrast with ***netlink*.**

Request

Requests are transactions that occur when setting and clearing requests for a kernel *service*, such as a KS_FifoPut. The kernel carries out the request with one or more *executions.*

Resource

A resource is a system *object*. It is a logical device used to control access to a physical device that is in demand by several *tasks*. The device is locked by the task that gains control of the resource to prevent higher priority tasks taking control before the current task has finished executing.

The priority of the task that currently owns a resource is increased to match the highest priority of any waiting task in order to avoid the priority inversion problem. You can limit the priority increase via the CEILING_PRIO global parameter in the *system folder*.

Scheduler

Virtuoso has three layers of operation: interrupt, *system kernel* and *application kernel*.

Interrupts are serviced by the lowest layer. The associated *ISR* is started as soon as the interrupt arrives and runs until it completes.

System kernel *processes* are prioritized, the programmer assigning a priority level (between 0 and 62) to each routine. When a process has the highest priority and is ready to run, it is started and runs until it either completes, calls a yield or blocking service, or an interrupt arrives.

Application kernel *tasks* are prioritized in the same way as system kernel processes, but a running task can be swapped out if a lower level routine or a higher priority task becomes ready to run. Swapping out is called **descheduling**. Being made ready to run is called **rescheduling**.

Application kernel tasks can also be time-sliced. This involves setting a limit to the length of time a task can run without being swapped out, using the KS_SchedulerSetSlicePeriod service. If a task is not swapped out by a higher priority task, it continues until the end of the time slice, when it is swapped out and the next task of equal priority is swapped in. If no equal priority task is waiting, the original task is swapped back in for another time slice.

Semaphore

A semaphore is a system *object*, enabling tasks to *synchronize* their activities. A task signals a semaphore, which increments a count value. Another task tests the semaphore and if it is not signaled can either wait with a timeout, or return. A semaphore is similar to an *event*.

A semaphore can be accessed globally, and may be used as a counter.

Service

Virtuoso provides more than 80 *application kernel* services for the application programmer to work with. They provide inter-task communications, interfaces to the data *objects*, and other operations.

SIZEOFUNIT_TO_OCTET

This is a macro that transforms the result of a sizeof operation (size_t) into 8 bit bytes (octets).

Synchronize

If an application is written as separate *tasks*, each task may need to know what stage of processing the other tasks are in. In Virtuoso they can do this by signaling and testing *event* and *semaphore* objects. The signal/test process is called synchronizing.

By the same principle, a task that sends a message may or may not need to know if the message has been received. If the sending task does need to know, Virtuoso can deschedule it until the receiver appears for the message, when the sender is released. Similarly, a receiving task may be descheduled while it waits for a message to appear.

Because each task knows what stage of processing the other is at, this is known as a **synchronous** message or transfer. If the task simply sends a message and carries on without waiting for the receiver, it is known as an **asynchronous** message.

Sysdef

The former name for the *project file*.

System folder

This provides system-wide information such as values for global parameters. You can change the values using the node view window of the *project manager*.

As supplied, the values in the system folder are:

- CEILING_PRIO **5**
  The maximum priority to which *tasks* can be boosted when they become owners of a *resource*
- DATALEN **16384**
  The length of data *packets* in bytes
- DRIVER_PRIO **0**
  The priority of the *netlink* drivers
- KERNEL_PRIO **0**
  The priority of the *application kernel*

- TICKFREQ **1000**
  Ticks per second of the virtuoso timer. This affects timeouts on application kernel *services* and the KS_TaskSleep service
- NLIFILE **hw.nli**
  The path to the *NLI file* for the application. If no path is specified this defaults to hw.nli, which is the default NLI file supplied by Eonic Systems
- EXE, FPU, SYS
  The default *task groups*

System kernel

Formerly known as the nanokernel.

The system kernel controls all *processes*, including the *application kernel*, and provides an interface for *ISRs* to communicate with the rest of the system.

The system kernel manages a collection of lightweight tasks called *processes*, which communicate with each other using *channels*. A process normally only uses a subset of the processor's registers, which means a context switch between two processes is very quick. The system kernel provides both an assembly language API and a C programming API.

Target

Used in explanations of the makefile.

The target is the output, which can be just a name (a **fake** target) or a file. The timestamps of the dependencies determines whether or not the target is rebuilt when the makefile is processed.

Target board

The board containing the processors on which the Virtuoso application will run.

Task

In a Virtuoso system a task is a code *object*: an independent module that performs a well-defined function or set of functions. It will typically talk to

other tasks using *service* calls to other objects: semaphores, fifos, mailboxes, and so on.

Tasks:

- are defined in the *project file*
- have an entry point, a priority and a stack size,
- can belong to a *task group*
- have a *task state*

Task ids are allocated by Virtuoso at compile time. The id has two parts:

- processor number
- task id within that processor

For example, the task id 0x0002000A represents task 0x000A running on processor 0x0002.

Task group

Virtuoso *tasks* can be grouped so that all tasks in the group can be operated on by group *services* (such as KS_TaskGroupStart). The group identifier is a 32 bit word, where each bit identifies membership of a different task group.

Group membership is defined in the *project file*, and can be altered dynamically at runtime using task services. A task can belong to as many groups as necessary, up to a maximum of 32. There are three predefined task groups:

- EXE. Tasks assigned to this group are started automatically when the application is loaded
- FPU. Originally related to use of the floating point unit of a Transputer, this group now is used on SHARC boards for tasks that need circular buffering enabled
- SYS. Tasks assigned to this group continue to run when the system is paused for the *Task Level Debugger* – so this group should contain only the tasks required by the Task Level Debugger

Task entry point

The entry point is the name of the function where execution starts (equivalent to main() in a normal C program). It is normally the same as the *task* name, except that the task name is UPPER CASE and the entry point is lower case.

The entry point is set at compile time, but may be changed before starting execution by a *service* call, to provide simple dynamic tasking.

Task priority

Every task has a priority, whose initial value is set in the *project file*. The priority determines which task should be run next. Priorities are in the range 0 to 62 where 0 is the highest. Priority 63 is reserved for the background task and should not be used by an application.

The priority can be changed dynamically using a *service* call.

Task stack

Each task has its own stack which is used for local storage of function calls and variables. On most processors, the task stack must also be large enough to handle worst-case *ISR* nesting. This is because the stack of the current task is used for saving the registers used inside each ISR.

The stack size is defined in the *project file*. It is defined as the number of entries, since the actual size is processor-dependent. When developing a new application, you should make the stack larger than you need to begin with and reduce its size later based on the stack usage information in the *Task Level Debugger.*

Task state

Enqueue: the task performed an enqueue operation on a FIFO

Dequeue: the task performed a dequeue operation on a FIFO

Send: the task initiated a message transmission

Receive: the task initiated a message reception

Sema: the task performed a KS_SemaTest(W(T))

Semalist: the task performed a KS_SemaGroupTest(W(T))

Resource: the task performed a KS_ResLock(W(T))

Memory: the task performed a KS_MapGetBlock(W(T))

Temp: used by the kernel to adjust task priority

Network: used by TLDEBUG task when requesting information form other node; or regular task performing KS_SemaStatus

Halted: task is halted

Terminated: task is terminated

Suspended: task is suspended

Blocked: used by debugging system to suspend tasks during debug session

Timer: task issued KS_TaskSleep

Driver: task issued driver call

Data: task issued KS_MemCpy

Event: task issued KS_EventTestW

Task Level Debugger

The Task Level Debugger is an application that runs on the host computer. It takes a snapshot of a running application and provides information about the state of the application *objects* so you can find out if the application is running efficiently.

Each tab in the debugger window monitors usage of a particular class of system object, such as the task class. The task monitor shows the stack usage of all the tasks running on the *target board* at the point the application was paused; the packets monitor shows allocated and used packets; and so on.

Tick

A tick is a unit of time. Its value is determined by the TICKFREQ variable in the *system folder* .

Timer driver

The Virtuoso timer driver is a low resolution driver with a frequency defined by the TICKFREQ variable in the *system folder*. The availability of high resolution drivers depends on the *target board* supplier.

The Virtuoso timer driver is essential for most applications. You must include it in your *project file* if you want to use KS_TaskSleep, or the timeout version of any services (KS_xxxxWT) .

Tracing Monitor

The Tracing Monitor is an application that runs on the host computer.

While the application is running in debug mode, information about semaphore signaling, resource locking and other system transactions is saved in a circular buffer. When the application is paused, this information is displayed by the Tracing Monitor. You can see the duration of all transactions occurring in the system, and the connections between them. The transactions are grouped by *node*, by *task* and by time.

VIRTDIR variable

This is an environment variable defined in the Virtuoso install process. It specifies the path to the directory where Virtuoso is installed (normally c:\Virtuoso). You should avoid using white spaces in the path name.

It is used throughout this user guide to mean the Virtuoso root directory.

WIN_DIR variable

This variable is only required by Win95 and Win98 systems and should be set to the Windows directory – normally C:\Windows. It is used by DOS commands like attrib – see the Win95 version of the base *makefile* for an example.

WinRT drivers

Some *target boards* use drivers from the board manufacturer, and some use WinRT drivers shipped with Virtuoso. See the current list of board drivers on the Eonic web site www.eonic.com/.

The WinRT drivers are developed by Blue Water Systems. For more information, check their web site: www.bluewatersystems.com/.

This page intentionally left blank

Chapter

2 Crash course

This chapter uses simple examples to introduce some of the major features of Virtuoso. If you are new to Virtuoso or have used previous versions of the software without the GUI interface, then it is worthwhile spending some time to familiarize yourself with the new features. Not all the features of Virtuoso can be covered in this chapter, but once you understand these basic facilities, the rest will be easily understandable.

The source code for all the examples is supplied on the Virtuoso CD, with all the necessary files required to create and run them. See the folder:

VIRTDIR\<processor>\sources

2.1 The key points

This crash course is meant to get you to the point where you can start programming real applications. Before you start:

- we assume that you have installed your system correctly
- we assume you know the basics of C programming
- it would be helpful if you knew a little about the make utility and makefiles – but it's not essential. We change the makefile during the crash course, but the changes are very simple

If you find a word you're not familiar with, or a familiar word that we seem to be using in a strange way, take a look at the glossary on page 1-11. You'll also find the glossary in the help system.

This chapter is a basic introduction. If you run into problems that are not described here, look in the reference sections of the guide. If they can't help you, don't hesitate to contact Eonic support. Contact details are on page 2 of this user guide.

2.2 Preparation

2.2.1 Where to find the example files

The first thing is to find the MyApp example. Look for the **VIRTDIR\<processor>\<compiler>** folder (where <processor> is the name of your processor, eg. Analog Devices Sharc, and <compiler> is the name of the compiler you're using, eg. VDSP).

The folder contains one subfolder for each supported board you installed, and below the board folder, subfolders for the example applications. The example is in the **VIRTDIR\<processor>\<compiler>\<board>\MyProjects\MyApp** subfolder (where <board> is the name of your DSP board, eg. Blacktip). Each of these subfolders contains the configuration for a particular application. They will also hold the temporary files used in building the application, and eventually the executables too.

You might want to browse through the description of the folder structure in book 3 of the user guide, because you'll need to know it reasonably well later.

2.2.2 Preparing source files

The **MyProjects\MyApp** example is an empty application configured to use standard I/O and to run in debug mode. You are going to add a task that does a printf. Then you'll compile and run the application.

The usual place to keep source files is in a subfolder under the **VIRTDIR\<processor>\sources** folder, and that's where the example in this tutorial puts it. If you want to keep it somewhere else, you'll have to make some small changes to the makefiles – see "Makefiles" in book 3 of the user guide for details.

If you configure your application to look in **VIRTDIR\<processor>\sources\hello** all the source you add will automatically be compiled into executables.

So to start:

- create a new folder **VIRTDIR\<processor>\<compiler>\<board>\MyProjects\hello**

- copy the contents of
  VIRTDIR\<processor>\<compiler>\<board>\MyProjects\MyApp
  to the new folder
- create a new folder **VIRTDIR\<processor>\sources\hello**
- using your favorite editor, open …**MyProjects\hello\makefile** and change:
  SOURCEPATH=VIRTDIR\<processor>\sources\MyApp
  to read:
  SOURCEPATH=VIRTDIR\<processor>\sources\hello
- save the updated makefile

2.3 Hello world...

Using the project manager

Now to start using the Virtuoso tools:

- click Start>Programs>Virtuoso>Virtuoso Project Manager

The main window of the Virtuoso Project Manager (VPM) appears. The VPM provides an easy way to configure applications. When you start it, the last project file you edited is loaded, or, if this is the first time you have started the project manager, a new project is created.

- select File | Open
- select
  VIRTDIR\<processor>\<compiler>\<board>\MyProjects\hello\hello.vpf

This opens the project file for your new application. It already contains a node entry and some other things, but you have to add some more specific entries. First, add a task.

(By the way, *node* is defined in the glossary on page 1-11).

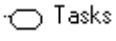
Creating a new task

If you read chapter 1 of this book you'll remember that Virtuoso applications are written as small communicating tasks. A task is just C code. The name of the task has to be added to the project file.

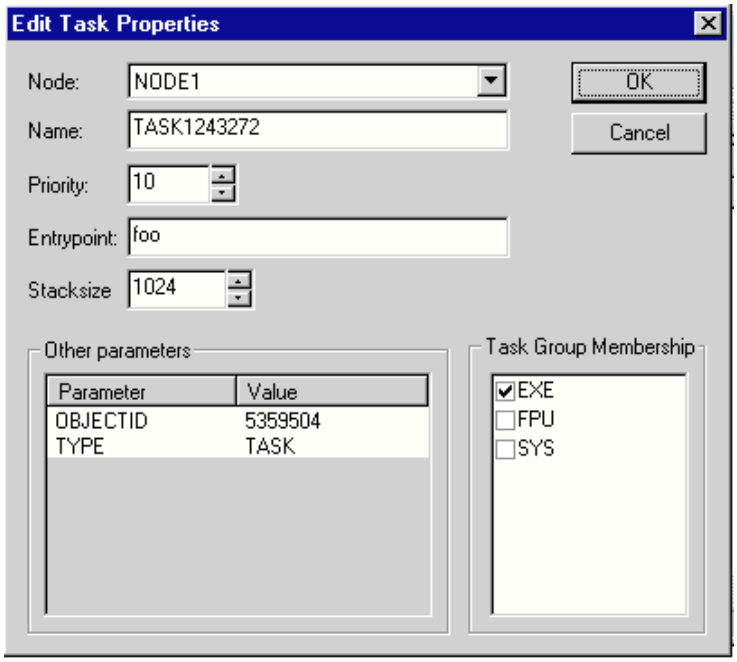
You'll be working in the object view window. To find the window, the easiest way is to:

- click the  icon on the main window, twice

When the window appears, to create a new task:

- in the **left hand pane** click the  icon
- select Edit | New | Task or click the toolbar button 

The Edit Task Properties dialog appears.



- type the properties of the new task

Node	NODE1
Name	HELLO [in upper case]
Priority	10
Entrypoint	hello [in lower case]
Stacksize	1024 entries. The actual size is processor-dependent
Task group membership	EXE

You can ignore “Other parameters” for now.

That’s it. When you finish typing, check your entries. If they’re correct, click OK and the new task is added to the project file.

Using upper case for Name and lower case for Entrypoint

While it is a good idea to make the name of the task the same as the entrypoint, they must be different cases. This is because in the **allnodes.h** file – an intermediate file output by Virtuoso’s generate utility – you will find a list of all kernel objects and their respective numbers as defines. For example:

#define RECVTASK 0x00010001

The symbol corresponding to the entry point of a function is the name of the function itself. So if you were to use the same name and case for the kernel object and the function, the c preprocessor would also replace the function name by 0x00010001

Getting the stacksize right

It is very important to get the stack size right. Stack overflow is a very common cause of application failure. You should always make the stack as big as possible (at least 1024 entries). You can then monitor stack usage with the Task Level Debugger, and decrease it as necessary.

Host communication objects

You'll notice there are already some Virtuoso objects in the MyProjects\hello project file:

- HOSTIODRV, a task that handles host communications
- HOSTMBX a mailbox used by HOSTIODRV
- HOSTRES, a resource used by HOSTIODRV
- a host driver for your board

Virtuoso always needs these for applications that use a host, and for development of hostless applications. Click the appropriate objects (task, mailbox, resource) in the left hand pane and look at their properties. Notice that the host drivers always run on the **root node**.

The root node is the processor directly connected to the host (usually via shared memory or a fifo type link). When the application is running, driver tasks running on the root node interface to the Host Server, providing services for the other processors. Other tasks that need fast access to the Host Server, such as the Task Level Debugger, should also run on the root node if possible.

Creating a new source file

Now you have to provide the task code that you've just told Virtuoso is part of your application.

See hello.c in **VIRTDIR\<processor>\sources\Man_ex_1** So:

- click the VPM main window
- select File | New Source file
- write the program! You need to include the Virtuoso standard I/O library **_stdio.h** (don't leave out the underscore)

Your new task is just a C function of type:

void hello(void)

Remember that the function name should be the same as the entry point specified in the project file.

Your HELLO task will be called automatically when the application is loaded, because in the project file you specified the EXE group as its task group. You don't need to call any task entry point functions yourself.

Once you've written the code:

- select File | Save
- save the source as hello.c in **VIRTDIR\<processor>\sources\hello**

Generating intermediate files

To generate intermediate files:

- click the  (save) button on the main window toolbar
- click the  (generate) button on the main window toolbar

Virtuoso's generate utility creates some intermediate files using the entries in the project file. The intermediate files are:

- node#.c – one per processor (eg. node1.c, node2.c)
- node#.h – one per processor (eg. node1.h, node2.h)
- allnodes.h

The header files define the global and local data objects for the nodes. The node#.c files contain the main() routine which initializes the node, synchronizes it with the rest of the system and starts up user tasks in any Task Groups that were defined to startup at boot in the project file definition (the EXE Task Group always starts at boot)

The intermediate files are used in the build process as input to the makefile.

Rebuilding the project

Now make the executable:

- click the  (save) button on the main window toolbar
- click the  (rebuild) button

This rebuilds the executable file. All files in the source folder are compiled and task entry points are mapped to the functions specified in the source files.

The build process relies on **makefiles** which supply configuration and other information to the compiler and linker. Virtuoso's makefile information is held in three separate files:

- VIRTDIR\<processor>\<compiler>**Base**, which contains the processor and compiler makefile requirements
- VIRTDIR\<processor>\<compiler>\<board>**Tools**, which contains the board-specific makefile entries
- VIRTDIR\<processor>\<compiler>\<board>\<category>\<example>**makefile**, which contains application-specific makefile entries.

Note that the name *makefile* must be all lower case.

See book 3 of the user guide for descriptions and examples of the makefiles. If you've already browsed through Virtuoso's folder structure and are wondering why there are application makefiles at several other folder levels in the Virtuoso structure, book 3 will tell you.

Makefiles are processed by the **make** utility, which the Virtuoso Project Manager calls automatically when you select the Build or Rebuild options. You can also call the make utility directly. It takes no parameters. The utility is located in the VIRTDIR/bin directory

Test the application

Once the build process has been completed there is a separate executable file for each processor in the network. Since we have defined only one processor for our "Hello World" example, there is only one executable. The executables are named for the nodes available: test1.exe for node 1, test2.exe for node 2, and so on.

To test the application, you can run the Virtuoso Host Server:

- click the  (Host Server) button on the main window toolbar

This downloads the executable to the target processor and runs the application. If everything is OK, the console window pops up and says "Hello world!"

Problems?

So, what if the console window doesn't pop up and say "Hello world!" ?

Compiling

… Virtuoso doesn't compile my hello.c source

- did you change the correct makefile? Is SOURCEPATH pointing to the folder VIRTDIR\<processor>\sources\hello?
- is your source in the VIRTDIR\<processor>\sources\hello folder ?

Running

… nothing prints, there's no console window

- try to run a pre-compiled example (e.g. KSBench)

If this fails, either your board or the Virtuoso software has an installation fault. Check the board first – check especially that you have installed the correct driver.

If the pre-compiled example runs:

- check that your source is similar to:
  #include <_stdio.h>
  void hello(void) { printf("/n Hello world ! /n"); }

… there's garbage on the screen or the console window is empty

- *make sure* you included _stdio.h (with the leading underscore)!

Comment

There's nothing in your code to tell the program to stop (unless you've added it yourself), so you have to stop it:

- click the  (Host Server) button on the main window toolbar

Your program stops.

The Virtuoso standard I/O library (_stdio.h) can be used by tasks running on any node. Remember to include the leading underscore(_), because you

need Virtuoso's version. The I/O requests will then automatically be routed through the root node to the host system (remember, the root node is the processor wired directly to the host).

If you don't include the leading underscore in _stdio.h, the compiler uses the printf function from its own stdio.h library. There are some limitations to _stdio.h compared to the compiler's version – see the #include of _stdio.h.

The project manager uses two databases to construct the intermediate files:

- the NLI file, which holds information about the target processor on which Virtuoso will be running
- the project file (VPF), which holds all the project related information

At startup, the system loads the default NLI file from the VIRTDIR\default folder. If this is not the correct NLI file you can change it using the system folder, accessible via the Node View Window. See "Changing global parameters" in book 2 of the user guide.

2.4 Inter-task communication

One task

So, your task runs. You may be surprised that the source looks like all the other C programs you've written. Of course it's extremely basic, but actually a Virtuoso task differs from traditional programs only in that it's usually shorter and doesn't contain many *if* or *case* statements. It's kept simple.

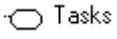
And that's the crux: a task does a very specific job – it may take characters from one place and put them in another, for example, or manipulate them in *one* way – and then hands over to another task.

When you break your traditional application into small tasks, you need to think about how the tasks communicate, and how they are scheduled. Although this is slightly more work for you than before, the result is a faster and more flexible system that is easier to maintain. And the scheduling part is not hard – remember you gave your HELLO task a priority of 10? That's your main contribution to scheduling.

Many tasks

Creating the task definitions

We'll add two more tasks for your first task to talk to. It's the same process we just went through for the first task:

- find the object view window
- in the **left hand pane** click the  icon
- select Edit | New | Task or click the toolbar button 
- type the properties of the new task

Node	NODE1
Name	WORLD [in upper case]
Priority	10
Entrypoint	world [in lower case]
Stacksize	1024
Task group membership	EXE

- check the values, then click OK

Now repeat for the third task, which has the same properties except:

- name: EXCLAM
  entrypoint: exclam

Synchronizing tasks

This sounds very grand, but is just a matter of defining some semaphores. A semaphore is a data object whose value is zero or a positive integer. One task *signals* the semaphore when it finishes a piece of work, and this signal increases the semaphore's value by 1. Another task wants to know if the work has been done, so it *tests* the semaphore. If the semaphore is zero, the second task knows the work isn't done and waits until the first task signals. Otherwise, the test returns with a standard message (RC_OK) and the semaphore value is decreased by 1.

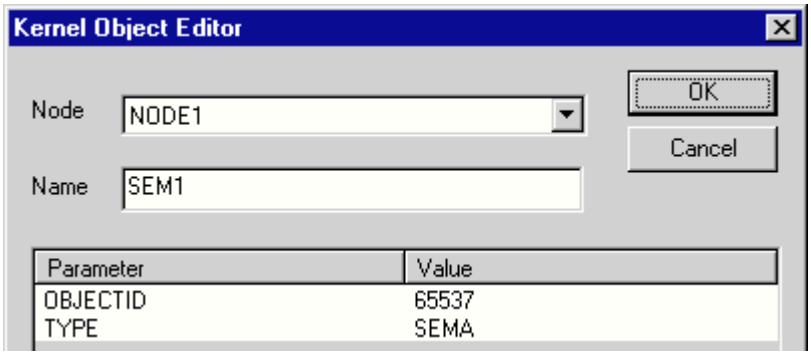
This is task synchronization. You use it instead of the kind of *if* statement within an ordinary application that says:

```
if k > 50 then <do this>
```

What will happen instead is that the part of the application that makes k greater than 50 signals a semaphore when it reaches 50. The part of the processing depending on the *if* is moved into a separate task, and tests the semaphore.

So, in our little example, we'll add some task communication. A semaphore is just another object, like a task, and you define it in a similar way, so:

- find the object view window
- in the **left hand pane** click the Semaphores icon
- select Edit | New | Semaphore or click the toolbar button 



- type the properties of the semaphore

Node	NODE1
Name	SEMA_HELLOWORLD

- check it, and click OK

Now add a second semaphore called SEMA_WORLDEXCLAM.

Changing the source

Tasks are the only objects you have to write code for. There's no code to write for the semaphore: it's an object you can use just by defining it in the project file and then calling an appropriate Virtuoso service in your task code.

So now we'll change the source of HELLO to use the semaphore, and add code for the two new tasks. You can use any editor you want to change the source.

Edit hello.c to read:

```
#include <_stdio.h>
#include <iface.h>
#include <allnodes.h>
void hello(void) {
printf("Hello ");
KS_SemaSignal(SEMA_HELLOWORLD); }
```

The reason for including iface.h and allnodes.h is given in the "Comment" section on page 1-46. Now add a new task WORLD to the same folder.

- Open a new file and type:

```
#include <_stdio.h>
#include <iface.h>
#include <allnodes.h>
void world(void) {
KS_SemaTestW(SEMA_HELLOWORLD);
printf("World ");
KS_SemaSignal(SEMA_WORLDEXCLAM); }
```

- Save the file as world.c

Now add a third task EXCLAM to the same folder:

```
#include <_stdio.h>
#include <iface.h>
#include <allnodes.h>
void exclam(void) {
KS_SemaTestW(SEMA_WORLDEXCLAM);
printf("!/n"); }
```

The HELLO task prints “Hello” on the screen then signals the HELLOWORLD semaphore. The WORLD task tests the HELLOWORLD semaphore, whose value will be 1, and which therefore returns RC_OK. The WORLD task then prints “World” and signals the WORLDEXCLAM semaphore; and so on.

After saving the new source files, add the two new tasks to the project file, in the same way you added the first task – see page 1-35 for details – and save the project file.

When you’ve edited and saved the three source files, and edited and saved the project file, you can recompile the application and run it like this:

- click the VPM main window
- click the  (rebuild) button
- click the  (host) button

If everything is still OK, the console window pops up and says “Hello, world!” just the same as the first version.

Click the  button again to stop the program.

Comment

It is not essential for the source code of the three tasks to be placed in three separate files, but it is better if they are separate.

If the three tasks are separate, they are compiled into three separate object files/executables. This means that only the code required on each node is downloaded to it. If the code for the three tasks was in one source file, only one executable would be produced which would have to be loaded onto each processor. Each processor would then have superfluous code, resulting in a waste of target memory space.

The header file **iface.h** that we included in the two new tasks enable you to use the KS_* functions (called *services*) supplied with Virtuoso. These services are the way tasks, semaphores and other objects are accessed and manipulated.

The header file **allnodes.h** that you included is generated from the project file. It is part of the SoftStealth system, and contains definitions of the system-wide kernel *objects* used in your program. In our case it contains

definitions of semaphores and tasks. You already know that a semaphore only needs a name for it to be used by a task. The header file **node#.h** is another generated file. It is included by allnodes.h, and contains node specific objects such as memory maps.

There are nine kernel objects in all, and they are made available simply by specifying them in the project file. Some, like semaphores, need only a name. Others, like tasks, need a name and a few more values. Once they are specified in the project file, you can use the object's KS_* services in your code to manipulate them. For example, the kernel object *fifo* needs these values in the project file:

- name
- depth (number of entries)
- width (size of each entry)

And *fifo* has four services:

KS_FIFOGet[W|WT]	Get the oldest entry from the fifo
KS_FIFOPurge	Clear the fifo of all entries and waiters
KS_FIFOPut[W|WT]	Put an entry in the fifo
KS_FIFOStatus	Read the number of entries

Now for a little useful information about tasks. When you specify a task, you include a *group* membership, an *entrypoint* and a *priority*.

The group membership enables you to operate on many tasks simultaneously. Any task in the predefined group EXE is started when the application is run. Any task in the predefined group SYS continues running when the Task Level Debugger is in use. You can create your own groups.

The task entrypoint is the starting address of the C function that implements the task. It is normally the same as the task name, but in lower case. Because the entrypoint function is specified in the project file, you don't need to call it yourself.

Tasks are scheduled by the microkernel according to their priority. The details are given in book 2 of the user guide. You have some influence on the scheduler with some of the KS_Task* calls.

For example you can:

- make tasks available or not available for scheduling
- change task priorities

One more thing: it's a good idea to think of tasks as black boxes. When two tasks interact, each should only know what goes in to the other task and what comes out. Then communication between the two tasks is straightforward.

Problems ?

Compiling

Are you sure you included iface.h and allnodes.h ?

Running

…garbage on the screen ?

- make sure the printf function comes after the KS_SemaSignal and before the KS_SemaTestW. Otherwise, the tasks will try to do a printf simultaneously, in which case the ouput of the printf calls will be mixed up. (Try it and see!). To stop this kind of interlacing, you can use resource locking: if you want, try using the KS_Reslock and KS_ResUnlock services on the STDIORES resource. Details of these services are in book 2 of the user guide
- make sure you typed KS_SemaTestW with the W. The W stands for *wait*, and it means that the task will wait until the sema is signaled. KS_SemaTest (without the W) returns immediately with a return value that indicates whether the semaphore value was zero or positive
- did you include "_stdio.h" in all three source files ?

2.5 Inter-processor communication

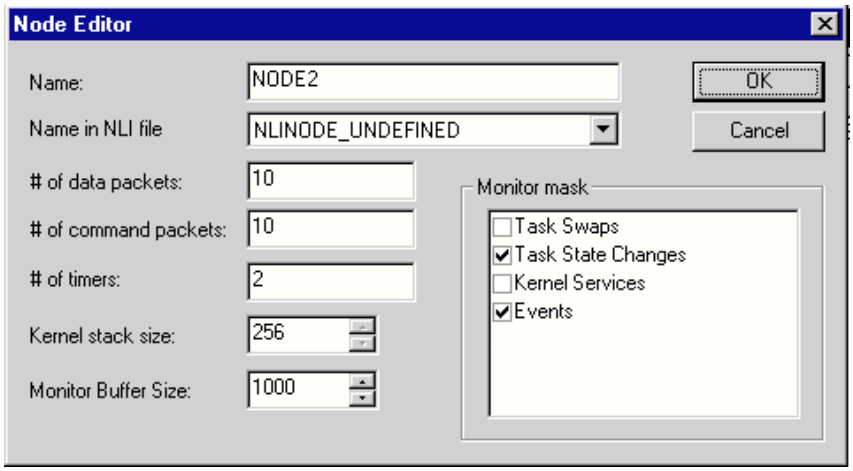
The examples in this section only work with target boards that have more than one processor, or with systems with more than one target board. Also, some processors do not allow multi-node operation. You'll have to look at the technical reference guide for your board to find out. If it supports multi-node operation, you can continue.

The purpose of the example in this section is to show how easy it is to move tasks and other objects from node to node (processor to processor). Moving tasks and other objects to different nodes can help to solve bottlenecks by distributing the processing load more evenly.

Add a new node

Nodes are treated like kernel objects in that adding a node is just a matter of defining its name and some other attributes in the project file:

- find the object view window
- in the left hand pane click the node group icon
- select Edit | New | Node or click the toolbar button



- change these two attributes of the node. You can use the default values of the other attributes

Name	NODE2
Name in NLI file	NODE2

- check the values, and click OK

You might take some time out at this point to check out what the attributes of a node object mean. They are described in the project manager help file and in book 3 of the user guide, along with the attributes of the other Virtuoso objects.

Add a netlink driver

At this point you should have read your processor supplement (book 4 of the user guide). You need the information in book 4 to complete this section.

We are now at the point where the differences in boards starts to make things a little complicated. Not for you, if you have just one or two boards, but for us, because we support many different boards and processor configurations.

Netlinks are used by the Virtuoso kernel for communications between nodes. A netlink is a description of a single physical link between two processors, and may be uni- or bi-directional. Directionality may be important. See page 1-64 for more information.

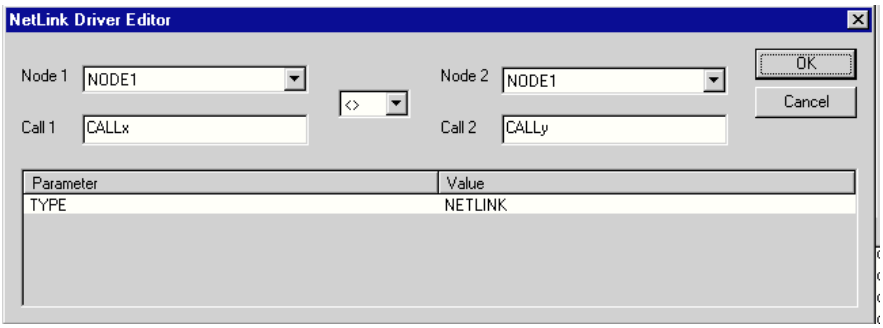
For each physical link, you have to specify at least two netlinks. Just as you have to tie a hammock to a tree at both ends, you have to specify a netlink on each processor (tree) connected by the physical link (hammock). A Virtuoso application can use as many processors as you have, as long as each processor can be reached from all other processors (either directly or via intermediate processors) . On some boards, you need to add two netlinks for each physical link – this is made clear in the processor supplement.

Like nodes, you can think of netlinks as kinds of kernel objects. To add a netlink you just define its name and some other attributes in the project file:

- find the object view window
- in the **left hand pane** click the 

icon

- select Edit | New | Netlink or click the toolbar button, eg. 



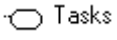
- leave the Node 1 value as NODE1
- change the Node 2 value to NODE2
- type the name of a driver initialize function (CALLx) for NODE1 and a driver initialize function (CALLy) for NODE2. The names of the driver initialize functions are given in your processor supplement
- in the dropdown textbox select the symbol **>** (ie. the direction of the netlink is from Node 1 to Node 2)
- check the values and click OK

Now repeat this procedure, still with NODE1 as Node 1 and NODE2 as Node 2, but specifying the direction as < (ie. the direction of the netlink is from Node 2 to Node 1).

Having specified the second node and the communication route between them, you can now move some tasks to the second node.

Move tasks 1 and 3 to the new node

Now you'll move two of the three tasks you've created to node 2:

- find the object view window
- in the **left hand pane** click the 

icon
- doubleclick HELLO and change the NODE attribute to NODE2
- click OK

- doubleclick EXCLAM and change the NODE attribute to NODE2
- click OK

It's that simple. Almost. We just have to make a small adjustment to the makefile and the nli file, which are configured for just one processor.

- open the file **makefile** in the folder **VIRTDIR\<processor>\<compiler>\<board>\MyProjects\hello** using your favorite editor
- find the line that says:
  ```
  all: test1.exe
  ```
- change it to read:
  ```
  all: test1.exe test2.exe
  ```
- add a compilation line for the test2.exe:
  ```
  test2.exe: $(DEPENDENCIES) $(ACHFILE) test2.lnk NODE2.o
  $(LN) -i test2.lnk -o $@ -a $(ACHFILE) –m
  ```

(that's all on one line)

- save the makefile

For the NLI file you have to add another node declaration, plus the bootlink path to the new node from the root node. Since the node declaration is processor dependent, the best way to see what is needed is to look at an NLI file for a two-node example in the folder **VIRTDIR\<processor>\<compiler>\<board>\<category>\<example>**.

When you've edited the NLI file:

- recompile the application using the Rebuild button in the VPM, and run it using the Host Server button

Congratulations! You have now produced a highly optimized parallel-processing multitasking application!

Problems ?

If the previous examples were OK, you shouldn't get any compiler errors.

… application doesn't boot

- test your board with a pre-compiled multi-processor-example, e.g. KSBench.2p

If it works (as it should if the previous examples worked), then go through the example again and check for editing errors. If it doesn't work, you should recheck your netlinks and the physical links with help from the technical reference manual for your board.

…application boots but doesn't start

- check the netlink definition

Comment

Configuration

Configuration issues are important in a multi-processor application, and separating the configuration information from the code makes life a little easier.

Virtuoso application code just says what happens, it doesn't say where it happens. The project file says where it happens, and to change the location of an object such as a task or semaphore is just a matter of editing the object's Node attribute. It's only when increasing or decreasing the number of nodes that you need to change the makefile.

The easiest way to upgrade an application to run on more processors is to:

- locate a multiprocessor example in the Virtuoso example subfolders
- copy the example's NLI file and those parts of the project file containing the needed node and netlink entries

Since the project file is a text file, you can use an editor to amend it. The layout of the project file is described in book 3.

Kernel objects

We keep mentioning *objects* or *kernel objects*. That's just a convenient term for data structures (mostly) provided by Virtuoso that you can use in your application. Tasks and semaphores are kernel objects. Here's a complete list:

- event
- fifo
- mailbox
- memory map
- memory pool
- resource
- packet
- semaphore
- task

They are described fully in book 2. Nodes and drivers are defined like kernel objects in the project file, but they don't actually qualify as kernel objects because there are no services for them.

2.6 Summary

If you've gone through the crash course, you know enough about Virtuoso to begin creating real applications using the application kernel.

To summarise, you need to:

- set up one folder for your application, to contain an NLI file, a project file, a makefile, and an architecture file
- set up a second folder for the source
- create the project file using the project manager
- write the source, using Virtuoso services to manipulate the kernel objects
- use the VPM generate utility to create the intermediate files

- use the VPM build or rebuild utilities to compile and link the code and create executables
- use the VPM Host Server to run the application

If you need to write ISRs and system kernel processes, you need to read the description of the system kernel services in book 2 of the user guide, and the processor supplement in book 4, because low level programming is beyond the scope of this brief introduction.

2.7 Other examples

The hello world example is supplied as man_ex_1 in the **VIRTDIR\<processor>\<compiler>\<board>\tutorial** folder. That folder contains six more examples of simple code, described below, whose source and associated project files are supplied in the **VIRTDIR\<processor>\sources** folder so you can study them.

There are also ten example applications that give a taste of what Virtuoso can do in the subfolders under the **VIRTDIR\<processor>\<compiler>\<board>**.

2.7.1 Example code

Man_ex_2 - simple host I/O

This task prints a prompt on the screen and then polls the host keyboard waiting for a character key to be pressed. Polling is not usually a good idea in a real-time application as it wastes valuable processor resources, but we use it here to keep the example simple. Once a key is pressed, the value of that key is printed out on the host screen, and the task finishes.

Man_ex_3 – semaphores

The CLIENTTASK prints a prompt on the host (via the Host Server) and then waits for a key to be pressed. This task makes periodic calls to the KS_TaskYield service while polling for the keypress to allow other tasks of equal priority to execute. When a key is pressed, the SEMA1 semaphore is signaled, and CLIENTTASK returns and waits for another keypress.

The SERVER task waits for the SEMA1 semaphore to be signaled. As soon as CLIENTTASK signals the semaphore, SERVER becomes ready to run, and because it is of higher priority than CLIENTTASK, will be scheduled immediately. SERVER prints a message acknowledging receipt of the semaphore signal, returns, and waits for the next signal.

Both CLIENTTASK and SERVER are in continuous loops, so they will run indefinitely.

This example shows how a semaphore is used to synchronize the operation of two tasks. In this case the tasks are running on the same processor node, but the source code would look identical, even if the client and server were running on different nodes.

Man_ex_4 - mailboxes

In this example we use a mailbox to show how intertask data transfers occur. The mailbox.vpf file shows that we have two user tasks called CLIENTTASK and SERVERTASK and a single mailbox called MAILBOX1.

As in the previous example the client task waits for a keypress and the server waits for a message in the mailbox. When a key is pressed, the CLIENTTASK composes a message with the key value in it and sends it to MAILBOX1. Notice that the message can only be read by the SERVERTASK (due to the value in the rx_task field). SERVERTASK receives the message (it too had specified that it would only accept messages from CLIENTTASK, via the tx_task field in its message structure) and unblocks. It has a higher priority than the CLIENTTASK, so the scheduler starts to run it immediately. The contents of the message are printed via the Host Server and the SERVERTASK returns and waits for another message to arrive.

Man_ex_5 - timers

This example shows how a timer can be used to wake a task periodically.

The TIMERINIT task requests a new timer from the pool of timers available, and initializes it to go off initially after 5000 clock ticks (5 seconds @ 1KHz), and then every 2000 ticks (2 seconds @ 1KHz). Whenever the timer goes off, the TIM_SEMA semaphore is signaled. The TIMERINIT task finishes once the timer has been started.

The TIMERRESPOND task waits on the TIM_SEMA semaphore. As soon as the semaphore is signaled (i.e. when the timer has gone off) a message is sent to the host. The first message appears after 5 seconds, with subsequent messages arriving every 2 seconds.

Man_ex_6 - fifos

This example shows how easy it is to have several tasks all accessing the same fifo, without having to worry about corruption through simultaneous access attempts.

Two tasks place entries on the QUEUE1 fifo at different rates, where each entry includes the task ID of the sending task. The server task waits for entries to appear in QUEUE1, when it reads the entry and prints a message on the host to show where the entry originated.

When you run this example you will see messages on the Host Server display showing the rates at which the clients are generating fifo entries.

Man_ex_7 - multiprocessing

If your target board only has one processor you will not be able to run this example.

Example 7 is similar to example 6 except that it runs on two processors. You can see from the source file that the code for the client and server tasks is identical.

If you look at the project file, you see that there are now two nodes defined. The first contains the CLIENT1 task and the fifo, while the second contains the CLIENT2 task and the SERVER task.

2.7.2 Example applications

If you want to know more about Virtuoso, the best thing to do is to run the examples applications in the folder:

VIRTDIR\<processor>\<compiler>\<board>\category>

and study the source in the folder:

VIRTDIR\<processor>\sources

The example applications supplied with Virtuoso tell you something about your Virtuoso system, and they also demonstrate some useful programming techniques. The examples are board specific, so some examples may not be available, or others may be included. If the board is available with different numbers of processors, there will be examples for each variant. The examples are designed for a particular number of processors: the examples for single processors have the extension .1p; those for two processors .2p, and so on.

Here are some brief descriptions of the examples.

Benchmark examples (Benchmrk folder)

Event

This tests all functionality for event objects, including enable, disable, signal while enabled and disabled, and error checking. The results are output on the host console.

HstBench

This measures the throughput over the host interface in two ways. First it calls a simple function (call_server) from the root node which shows the speed of the direct connection to the Host Server. Second, it does the same test using another function (process_cmd) that sends a message to the host I/O driver task, which then calls the Host Server. This shows the power of the Virtual Single Processor feature, since the calling task resides either on the root node or on another node – but there is no difference in the source code. The benchmarks send output to the host console, and to a (Borland's Graphical Interface (BGI) graphic window.

KSBench

This tests most of the kernel services. First it enters and exits the kernel, showing the raw response times. It then does mailbox transfers, fifo transfers, semaphore signaling, resource locking, memory pool and memory map requests. This test is a good indicator of Virtuoso's kernel performance.

RawDMA and RawSPORT

These show the impact of data packet sizes on performance. One MWord is transferred between processors in increasingly large packets, starting with 1 Word packets and doubling in size until the packet size is 16K Words. The effect is to show transfers that start very slowly and get exponentially faster.

Stdio

This just reads a file and sends it to the host screen in different ways. It uses fstats among other things, and does a small benchmark of host I/O operations. It is a good test for proper operation of host service modules and other parts of the Host Server.

Txd and Txdr

These are two versions of a data transfer test using KS_memcopy. One MWord is transferred between processors in increasingly large packets, starting at 1 Word and doubling in size until it reaches 16K Words. The TXD task sends the data, the TXDR task requests the data. At the end of the test there is a graphic showing throughput and number of packets transferred.

Demonstration examples (demo folder)

Debug versions of these programs are in the demo.dbg folder.

Balls

This is a simulation of charged particles moving in a field containing a force perpendicular to the particle's velocity. The charge strength and the force are input as application arguments to the program in the Host Server. Try entering 5 for the charge strength and 12 for the force, ie.

The example shows how flexible mailboxes are. Each ball is a task that sends the ball position and velocity to a master task. The info field of the message structure contains the ball's id. (See book 2 for more information

about the message structure K_MSG). The master task puts the position, velocity and id of each ball into an array and sends it to each ball. So each ball knows about all the other balls and can calculate its own next position. The movements of the balls are shown in a graphic window.

Latency

This is often used to show the real-time characteristics of the operating system. The example measures the delay between an interrupt coming in to a processor and the point at which the application begins to process the interrupt. The results are shown in a histogram. There are options in the graphic window to: activate the balls example in the background (to simulate a higher workload); to initiate interprocessor data transfers (to stress the communications network); to run calculations on each processor (to simulate a saturated system); to switch between the different levels of the kernel; to write the data to a file; and to display the min, P15 and max of the histogram.

Mandel

This is a simple example of a controlling task that farms out processing tasks. The master task divides the complex plane into subsections and puts packets of work onto a fifo. The processing tasks get the packets from the fifo, execute it, and send the output to a plot task that draws the Mandelbrot set on the screen.

Tim

This shows how to use the Virtuoso system timer services. A timer located on the first node is started with a cyclic period. Every time it expires, a semaphore is signalled. A master task waits for the semaphore, and when it has been signalled, the master task signals four other semaphores. There is a separate LED task for each semaphore which:

- waits for the semaphore
- increments an internal counter
- if the counter == modulo count, sets the counter = 0 and signals the semaphore for following LED
- updates the display

2.8 Some programming hints

Virtuoso has many good qualities. For example, it encourages task-centered programming: tasks are generally more efficient and more manageable than monolithic applications. It separates application code from configuration operations so that it is easy to run the same application on a different number of processors or even on different makes or types of processors. It has tools that make configuring, running and debugging applications easy.

But you still have to write the code.. here are some general techniques that will help you write better Virtuoso applications. Book 2 of the user guide also contains some tips about programming.

2.8.1 General

Reading the processor supplement

READ the processor supplement!

Using Virtuoso services

Always use Virtuoso services and functions from the Virtuoso libraries. Use of other functions will destabilize the scheduler.

Initializing Virtuoso objects

You don’t have to. The values of objects like semaphores, events and resources are reset to zero on boot up.

Locating tasks

While it is possible to locate tasks on any processor (node), you should be aware that all inter-processor communication incurs an overhead. Put related tasks and objects on the same processor if possible, to reduce the overhead.

Using internal Ids

Don’t use the variables KS_taskId and KS_nodeId instead of the task names in your code, they may change.

Similarly, don’t assume the root node is node 0 – it usually is, but doesn’t have to be.

Working with size parameters

Virtuoso supports processors that allow byte aligned addressing as well as processors that only allow word aligned addressing. This can cause a problem because the sizeof() function always gives the size of a char as the difference between consecutive addresses.

For example, on a processor like the Motorola 96K, a char is represented as a 32bit word (with the leftmost 8bits containing the char) and this has a size of 1. This can be a problem when transferring code between two different types of processors.

Defining variables

Define variables in the project file. The generator then inserts a #define for them in the header file and they can be used anywhere in the task code.

Understanding packets

Data is sent round the system in packets. The number of packets to be made available is specified for each node in the project file. There are three types:

- **command packets** are used to send commands between nodes, to implement timer operations and to build waiting lists. A command packet is 20 words. Usually, four or five command packets per node is a good choice
- **data packets** are used to buffer data **on intermediate nodes** when data is transferred between indirectly connected nodes. Each packet is assigned the priority of the sending task. In diagram 1 of the figure on page 1-65, data packets are only needed on processors 2, 3 and 4. The size of data packets is defined as a global variable in the system folder of the project file. Usually, four packets of 2k bytes is a good starting choice, but we recommend the use of the Task Level

Debugger to optimize packet use. While larger packets may be faster for sending single messages, they may block the channel while higher priority packets are waiting

- **timer packets** are used for kernel level timer operations. One timer packet and one command packet are required for each timeout operation requested by a task

In most cases, timeouts are executed on the node where the timed-out object resides; so timer packets should be allocated there, and not on the node where the requesting task is.

The exception is a wait request using KS_SemaGroupTestWT, where the timeout is executed on the requesting task's node.

Using structs

You can use structs such as K_FIFO in your code. If you pass the fifo name in the struct, it is translated from an integer to an UNS 32 in allnodes.h and node#.h. See book 2 of the user guide for information about Virtuoso structs.

Creating application folders

It is now a simple process to create applications in folders other than the default folder. To create an application in a different folder:

- create a new folder for the application
- copy an NLI file, a makefile and an architecture file to the new folder. You can use the NLI file from the **VIRTDIR\default** folder; and a makefile and architecture file from any of the examples, although the architecture file may not be optimal for your application. You should use a makefile from an example designed for the number of processors on your target board
- Start the Virtuoso Project Manager
- Select File | New and specify the NLI file in the new folder
- Use the Project Manager to specify the application
- Select File | Save and specify the new folder

Using the command line

Much development can be done from the command line. For example:

- you can run the Virtuoso editor (vseditor.exe)
- you can edit Virtuoso project files, which are text files just like the NLI, ach, and make files
- you can run the generate utility with the command
  `generator infile`

where *infile* is a pre-processed project file.

2.8.2 Links and netlinks

Using onboard links

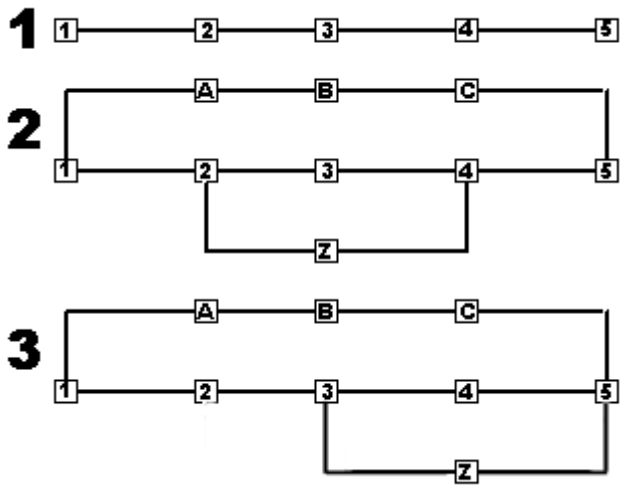
Use onboard direct links whenever possible if you want the fastest possible communication. Use offboard links only to communicate with other boards and external devices.

Virtuoso does not use the cluster buses at the present time.

Creating the correct topology

When Virtuoso reads the NLI and project files it establishes the shortest paths between non-adjacent processors using the netlinks declared in the project file. It just counts the number of intervening processors. Once determined, those paths are static unless the project file is changed.

Virtuoso then uses the shortest path when transferring data between processors. If there are two or more paths that are equally short, all are used and the data is divided equally between them.



These three diagrams show some netlinks joining processors. All paths between processors 1 and 5 are equally short so:

Diagram 1: 1-2-3-4-5 gets 100% of the traffic

Diagram 2: 1-A-B-C-5 gets 50%, 1-2 gets 50%, 2-3-4 gets 25%, 2-Z-4 gets 25%, 4-5 gets 50%

Diagram 3: Here, directionality has an effect. Data traffic is divided according to the links available on the originating processor. So, if all links are bidirectional, with data originating on processor 1:

1-A-B-C-5 gets 50%, 1-2-3 gets 50%, 3-4-5 gets 25%, 3-Z-5 gets 25%

And with data originating on processor 5:

5-C-B-A-1 gets 33.3%, 5-4-3 gets 33.3%, 5-Z-3 gets 33.3%, 3-2-1 gets 66.6%

2.8.3 Drivers

Most embedded applications need you to write drivers as well as applications. Drivers are low level programs that provide generic or particular access to hardware. They have an associated startup program

that installs any ISRs, starts processes and so on. The startup program must be declared in the project file driver definition. The default drivers are declared in nodetype.h.

If you need to write your own driver, you need to program on the lower levels of Virtuoso:

- ISR level – the interrupt handling part of your application
- system kernel level – supporting processes

The system kernel offers very high performance and very low code size. Its functionality is limited, and processes are normally programmed in assembly language, although, if necessary, processes can be written in C. See the description of the system kernel in book 2 for more details.

In the crash course you used two kinds of drivers:

- a host driver: You can only do the crash course if you have a board we can support out of the box, ie a commercially available (COTS) board. If you have a custom board you normally need to write your own host driver if you want to use the Virtuoso host capabilities (_stdio.h etc.), although you can use a COTS board as a gateway to your custom board – see the chapter about Boot methods in book 3
- a netlink driver: We can only support netlink drivers that we supply

There are three other types of driver:

- rawlink driver: This is a driver used by an application for sending or receiving data over a hardware channel (linkport, serial port, and so on). We supply some rawlink drivers, and you can write your own
- timer driver: We can only support timer drivers that we supply
- user driver: This is entirely application-specific

For the list of drivers already implemented for your processor, see the processor supplement.

2.9 Services that do not cause a task switch

Misc	KS_Linkin	KS_IRQSetHandler
	KS_Linkout	KS_SchedulerSetSliceperiod
	KS_ISREnable	KS_MemCpyA
	KS_ISRDisable	
Event	KS_EventSetHandler	KS_EventDisable
	KS_EventEnable	
Fifo	None	
Mail	KS_MsgPutA	
Memory pool	None	
Memory map	KS_MapStatus	
Resource	None	
Semaphore	KS_SemaReset	KS_SemaGroupReset
Task	KS_TaskID	KS_TaskGroupLeave
	KS_TaskPrio	KS_NodeID
	KS_TaskGroupMask	KS_TaskSetEntry
	KS_TaskGroupJoin	KS_TaskSetAbortHandler
Timer	KS_HighTimerRead	KS_LowTimerGet
	KS_LowTimerRead	KS_LowTimerStart
	KS_LowTimerFree	KS_LowTimerRestart
	KS_LowTimerElapsed	KS_LowTimerStop
Workload	KS_WorkloadSetPeriod	KS_WorkloadRead

1 End of book