Virtuoso ™

The Virtual Single Processor Programming System

User Manual

Covers :

Virtuoso Micro ™

Virtuoso Classico ™

Version 3.11

Table of Contents

Alphabetical List of Virtuoso microkernel services P2 - 25

Part 3: Binding Manual P3 - 1

Virtuoso on the Analog Devices 21020 DSP ADI - 1

Alphabetical List of nanokernel entry points ADI - 30

Introduction

Virtuoso™ is a family of real-time processing programing systems. As this family is expanding it became necessary to differentiate between the different product offerings. The general philosophy however is the same: ease of use and portability with no compromize on the performance.

Most of the products are available in three different implementations.

SP : Single Processor implementation.

These implementations do not assume the presence of any other processor in the system. The Virtuoso kernel provides multi-tasking with preemptive scheduling.

MP : Single Processor implementation with multi-processor extensions.

The multi-processor extensions enable fast and easy interprocessor communication, all in about 500 instructions. With minimum set-up times, it provides for maximum performance by using the DMA engines when available. Communication is point-to-point between directly connected processors.

VSP : Virtual Single Processor implementation.

The true solution for parallel processing is to implement the communication as part of the kernel service, hence providing fully transparent parallel processing. The VSP implementation provides this feature by way of fully distributed semantics, permitting to move kernel objects and/or changes to the target topology without any changes to the source code.

The current product offerings are as follows :

Virtuoso Nano is based on the core nanokernel of the Virtuoso product range. It can be very small (200 instructions) but is ultrafast. It provides for true multitasking and interprocess communication services. The VSP implementation is still not larger than 1000 instructions.

Virtuoso Micro features a small but fast microkernel that provides preemptive scheduling for a number of prioritized tasks. Well suited when preemptive scheduling is needed and the appication has moderate interrupt requirements..

Virtuoso Classico : a tight integration of Virtuoso's nanokernel and Virtuoso's microkernel. The microkernel provides fully distributed high level semantics with no source code changes when kerrnel objects or user tasks or moved in a processor network or when the system topology is changed.

Virtuoso Modulo 0 to VI contain a complete range of libraries in optimized assembly covering vector and matrix functions, filters, FFT, EISPACK (eigen value functions), BLAS (Basic Linear Algebra Subroutines, and 2-dimensional image processing. All written in optimized assembly. Part of the libraries were developed by Sinectoanalysis from Boston, MA and adapted for Virtuoso. The package is complemented by a board specific host server that boots the target network and provides standard I/O, PC graphics and heap allocation functions.

For the rest of the manual, we will often use the term Virtuoso as the context makes it clear what product offering is being discussed.

Virtuoso currently supports the following processors :

ARM, INMOS T2xx,T4xx, T8xx, T9000, TEXAS INSTRUMENTS TMS320C30/C31/C40, Motorola 68xxx, 68HC11, 68HC16, Intel 80x86 (real mode), MIPS R3000, Motorola 96K, 56K, Analog Devices 21020, 2106x, Pine & OakDSPCore. (contact Eonic Systems or your distributor for a list).

Not all processors supported by Virtuoso are supported with all possible implementations. A choice was made depending on the specific processor architecture as well as on the typical use made of the processor. In all cases is the microkernel available offering an identical interface from single 8bit microcontrollers to mixed parallel processing networks of 32bit DSPs and other processors.

As not all processor versions are fully upgraded to the latest version. Refer to the release floppy or the previous manual if there is a inconsistency. Eonic Systems is upgrading all versions to be source level compatible as much as possible.

Eonic Systems International Inc. has taken up the challenge to continually improve its product by further streamlining the code, by extending the functionality and the flexibility of the kernel and by adding tools that will support the designer during the development phase as well as during the rest of the life cycle of the applications developed. Therefore, the current version is subject to modification and will be upgraded on a regular base.

For more information : info@eonic.com

For support, contact : support@eonic.com

WEB page : http://www.eonic.com

Release notes

V.3.01 September 1992

This version does not contain many changes. In particular :

1. The I/O library was revised and extended;
2. Terminal type I/O is now a separate library;
3. The compile and development cycle has been shortened by streamlining the makefiles and library decomposition;
4. The KS_Alloc kernel service was modified to allow deallocation when a task is aborted;
5. A new universal network loader.

We made a lot of efforts to support even better DSP applications. In particular :

1. The ISR structure has been reviewed permitting to eliminate most of the interrupt disabling times.
2. Light context tasks were introduced. These are used internally by the kernel but can be defined and programmed as well by the user.

V.3.05 January 1993

The major novelty is the introduction of the nanokernel for the distributed version. This consists of several light context tasks and enables very fast interrupt servicing.

Following enhancements were added :

1. The router will use multiple paths, if possible.
2. A new service, the KS_MoveData was introduced
3. The transputer and C40 version can be used transparently on mixed networks.
4. Introduction of an Application Development Support Package consisting of a Vector, Matrix and FIter library (separate product).

V.3.09 September 1993

The major change is the updating of the manual documenting the nanokernel. For the single processor versions, the same hostserver and netloader is being used as with the multiprocessor versions to improve the portability of the applications. Following changes were made :

1. An improved *.NLI file format;
2. An improved tracing monitor.

V.3.09.1 November 1993

In this release all nanokernel services were implemented as traps. This has the benefit that the total interrupt disabling time was reduced and that the nanokernel code can be placed on any memory bank in relation to the program code.

The manual was largely updated and covers the Virtuoso support packages in a single manual. Virtuoso Nano is not yet documented in its VSP implementation

From this release on, every license includes free of charge a binary version of Virtuoso Micro for use with Borland C under DOS, permitting an easy cross development at the microkernel level.

V.3.11 September 1996

This release adds a C++ API to the microkernel services.

This reference part of this manual has been extensively revised,.

This version is the first version of Virtuoso to be tested production versions of the Analog Devices 2106x processor.

Implementation-Specific Features

As these manuals are generic, not all the software versions will correspond fully with it. Some advanced features might be missing or implemented differently depending on the actual target processor. Note however that later versions are always supersets of the previous ones unless serious technical reasons dictated syntax changes. Refer to your interface libraries for a correct definition of the syntax and the read.me files.

Trademark Notices

Virtuoso™ is a trademark of Eonic Systems Inc.

12210 Plum Orchard Drive, Silver Spring, MD 20904

Tel. (301) 572 5000, Fax. (301) 572 5005

e-mail: info@eonic.com. For support : support@eonic.com

WEB : http://www.eonic.com

Europe :

Nieuwlandlaan9, B-3200 Aarschot, Belgium.

Tel. : (32) 16.62 15 85. Fax : (32) 16.62 15 84

Virtuoso Nano **™** is a trademark of Eonic Systems Inc.

Virtuoso Micro **™** is a trademark of Eonic Systems Inc.

Virtuoso Classico **™** is a trademark of Eonic Systems Inc.

Virtuoso Modulo **™** is a trademark of Eonic Systems Inc.

Virtuoso Molto **™** is a trademark of Eonic Systems Inc.

RTXC is a trademark of A.T. Barrett & Associates.

TRANS-RTXC is a trademark of Eonic Systems Inc.

RTXC/MP is a trademark of Eonic Systems Inc.

RTXCmon is a trademark of Eonic Systems Inc.

The history of Virtuoso

Welcome to the world of **Virtuoso**. We think that you have purchased one of the most versatile and unique systems available for the implementation of a real-time system, be it on a single or on a multi-processor target system. Before we jump into the details, we would like to spend a few moments to explain the philosophy behind **Virtuoso**.

In 1989, Intelligent Systems International (which later became Eonic Systems), was founded. At that time, the INMOS transputer was the only processor available with in-built support for parallel processing, however, it lacked the ability to support tasks with multiple levels of priority. In order to apply this technology to hard real-time applications, ISI wrote a multi-tasking kernel for the transputer. Derived from RTXC, ISI added support for multiple processors, and launched a product called TRANS-RTXC.

In addition, ISI started to port to a variety of other processors. This was possible because TRANS-RTXC was redesigned much more with portability in mind, and re-named as RTXC/MP. It was available for targets ranging from 8-bit microcontrollers to 32-bit multi-processor networks.

A major addition to the supported target processors was the Texas Instruments TMS320C30 and C40. This brought RTXC/MP into the DSP world, and to a new level of performance. However, the requirements of DSP applications needed a radical new approach in the implementation of the kernel. On the one hand, DSP applications running on hundreds of processors require a powerful and easily understood paradigm for distributed processing; while on the other hand the need to process interrupts from many sources requires an efficient, low-level approach. Often these apparently conflicting requirements are present in the same system.

The result of these considerations was **Virtuoso**. The concepts behind **Virtuoso** are some of the most advanced, and they combine to give very powerful and efficient support for real-time, DSP and parallel system design.

The ability to use any of the kernel services to access a kernel object located on another processor in the system, no matter where it is located, frees the programmer from considering the details of interprocessor communication. Because the semantics of the kernel services were designed for distributed operation, it is the only system that guarantees that the deterministic behavior of the application is unchanged when the target network is changed. This paradigm is called the **Virtual Single Processor**, as it allows a multi-processor target to be programmed exactly as if it were a single processor.

In the same context **Virtuoso** also provides multiple levels of support, allowing a trade-off to made of ease of programming for performance. Two levels

are dedicated to handling interrupts, one level consists of light context tasks (called the nanokernel processes) and the highest level is the preemptive priority driven C task level. This level has a rich set of semantics, is independent of the network topology and hence is fully scalable.

Despite providing these rich features, **Virtuoso** is still one of the smallest and fastest real-time operating systems available.

The latest addition to the family is **Virtuoso Synchro**, an application generator for synchronous dataflow applications. It supports the specification, simulation, emulation and implementation of mono- and multi-rate DSP applications on multi-processor targets, using a graphical user interface. The generated code runs with minimum of input-to-output delays and memory requirements, and can be used where even the smallest overhead from the kernel would be unacceptable

Virtuoso is complemented by a range of support tools and libraries that make **Virtuoso** a complete programming environment, designed to meet the needs of the developer in a wide range of applications. What **Virtuoso** delivers today is the potential to combine the incremental processing from a single processor to over a 1000 multi-processor network while meeting hard real-time constraints.

We look forward to receiving your comments, opinions and suggestions which might help us in the evolution of **Virtuoso**. As **Virtuoso** comes with 12 months support and upgrades, do not hesitate to contact us. It could save you a lot of time and it could start a long lasting relationship.

Milestones

- 1989: ISI founded
- 1990: Release of TRANS-RTXC
- 1992: **Virtual Single Processor** concept introduced
- 1992: RTXC/MP ported to the TMS320C30 and C40
- 1992: Release of 2nd generation kernel, **Virtuoso**
- 1993: Nanokernel programming level introduced
- 1994: Port to ADSP 21020 and 21060
- 1995: Release of Virtuoso Synchro
- 1995: ISI changed name to Eonic Systems, Inc.

Manual Format

This manual set is divided into three distinct parts.

PART 1 Virtuoso concepts

This part discusses the general philosophy behind **Virtuoso**. It gives information on how **Virtuoso** operates, the concepts behind its design and how the developer needs to use it. A short tutorial is included in order to prepare the user who is not familiar with real time programming.

PART 2 Virtuoso Reference Manual

This part contains the reference part of the manual with a detailed discussion of the way the **Virtuoso** kernel works, how to use the kernel services and how to use the **Virtuoso** development tools.

PART 3 Virtuoso Binding Manual

Part 3 of this manual contains the specific information about installing and using a given **Virtuoso** implementation for a given target processor with a given C compiler. As this information may vary for different combinations of processors and compilers, the contents of this part depends on the particular combination you have licensed.

Other manuals include :

Virtuoso Technical Notes.

Virtuoso Modulo 0 - VI.

License agreement

EONIC SYSTEMS agrees to grant upon payment of fee, to the undersigned **CUSTOMER** and CUSTOMER agrees to accept a non-transferrable and non-exclusive license to use the Software, hereinafter referred to as the Licensed Product, as listed in the license registration form and subject license registration form is attached hereto and made a part of this Agreement. In case of doubt, the items as mentioned on the invoice upon delivery of the Software, shall be taken as the Licensed Product.

OWNERSHIP AND CONDITIONS :

1. OWNERSHIP :

Customer acknowledges that Eonic Systems retains all rights, title, and interest in and to the Licensed Product and all related materials are and shall at all times remain the sole and exclusive property of Eonic Systems. The Licensed Product, the original and any copies thereof, in whole or in part, and all copyright, patent, trade secret and other intellectual and proprietary rights therein, are owned by and remain the valuable property of Eonic Systems. Customer further acknowledges that the Licensed Product embodies substantial creative efforts and confidential information, ideas, and expressions. Neither the Licensed Product nor this Agreement may be assigned, sublicensed, or otherwise transferred by Customer without prior written consent from Eonic Systems.

2. FEES :

For and in consideration of the rights and privileges granted herein, Customer shall pay to Eonic Systems a license fee, due and payable upon execution of this Agreement, in the amount specified on the invoice.

3. DEFINITIONS :

3.1 SOURCE CODE is any representation of the Licensed Product that is suitable for input to, or is produced as output from an assembler, compiler, interpreter, source translator, or disassembler, either directly or indirectly on any medium, regardless of type, including, but not limited to listings printed on paper, and any magnetic or optical medium.

3.2 EXECUTABLE CODE is any representation of the Licensed Product which can be directly executed by the instruction set of a computer or indirectly by an interpreter in a computer. The storage or transmission medium is not relevant to this definition and includes, but is not limited to, magnetic, optical, Read-Only-Memory of all sorts, and Random Access Memory.

3.3 OBJECT CODE is any form of the Licensed Product not included in the definitions of SOURCE CODE or EXECUTABLE CODE above including, but not limited to, object code files and object code libraries on any medium.

3.4 SITE is any single designated place of business where the Licensed Product will be used by Customer in the development of Customer's application. The SITE is limited to a single building or department or group of license users but Eonic Systems may, at its sole discretion, determine what shall constitute the SITE.

4. CUSTOMER'S PRIVILEGES :

Regarding the Licensed Product, the Customer may :

4.1 Use any representation of the Licensed Product on one development station at the Customer's SITE.

4.2 Copy the Licensed Product for backup or archival purposes and to support Customer's legitimate use of the Licensed Product.

4.3 Merge or otherwise combine the Licensed Product, in part with other works in such a fashion as to create another work agreeing that any portion of the Licensed Product so merged remains subject to the terms of this Agreement. Whenever the source code of the Licensed Product is changed during the work, CUSTOMER shall consult Eonic Systems to verify if the changes are within the boundaries of the License Agreement.

4.4 Distribute on any medium the EXECUTABLE CODE derived from the Licensed Product so long as the Licensed Product is an integral and indistinguishable part of the EXECUTABLE CODE and the applicable runtime license fee has been paid to Eonic Systems.

4.5 Extend this Agreement to include more than one SITE by paying the appropriate license fee for the Licensed Product for each additional SITE.

4.6 Extend this Agreement to include more than one development station by paying an additional license fee for the Licensed Product for each additional development station.

5. CUSTOMER OBLIGATIONS :

Regarding the Licensed Product, the Customer shall :

5.1 Include and shall not alter the Copyright or any other proprietary notices on any form of the Licensed Product. The existence of any such copyright

notice shall not be construed as an admission or presumption of publication of the Licensed Product.

5.2 Maintain appropriate records of the number and location of all copies that it may make of the Licensed Product, and shall make these records available to Eonic Systems upon reasonable request thereof.

5.3 Upon termination of this license, render written certification that all copies of the Licensed Product and any related materials, in any form, excluding EXECUTABLE CODE have been destroyed.

5.4 Take appropriate action by agreement or otherwise, with its employees, contractors, subcontractors, agents, or any other person or organization under Customer's control and having access to the Licensed Product, to satisfy Customer's obligations under this Agreement with respect to the use, copying, protection, and security of the Licensed Product.

5.5. Pay to Eonic Systems a runtime license fee for every processor or computer system executing any instance of the licensed product, be it as OBJECT CODE and indistinguishable from the EXECUTABLE CODE as far as the payment of said runtime license fees is not covered by any other agreement between Eonic Systems and CUSTOMER.

6. CUSTOMER PROHIBITIONS :

Regarding the Licensed Product, the Customer shall not :

6.1 Permit any person or persons under its control to compromise the exclusiveness of the Licensed Product and the rights of Eonic Systems under the law of this Agreement.

6.2 Provide or otherwise make available to another party, any SOURCE CODE or OBJECT CODE or documentation which forms part of the Licensed Product, whether modified or unmodified or merged with one or more other works.

6.3 Use the benefits of the Licensed Product to engage in the development of a product or products having the equivalent functional specification or serving the same purpose as the Licensed Product so as to be in direct competition with the Licensed Product.

7. LIMITED WARRANTY :

NO WARRANTY OF THE LICENSED PRODUCT IS PROVIDED EXCEPT AS STIPULATED HEREIN.

7.1 Eonic Systems provides the Licensed Product “As Is” without any warranty, expressed or implied, including but not limited to, any warranty of merchantability or fitness for a particular purpose.

7.2 Eonic Systems does not warrant that the functions contained in the Licensed Product will meet Customer’s requirements, or that the operation of the Licensed Product will be uninterrupted or error free.

7.3 Eonic Systems does warrant the media upon which the Licensed Product is distributed to Customer to be free of defects in material and workmanship under normal use for a period of ninety (90) days from the date of shipment of the Licensed Product to Customer. Eonic Systems will replace such defective media upon its return to Eonic Systems.

7.4 Eonic Systems’ liability hereunder for damages, regardless of the form of action, shall not exceed the amount paid by Customer for the Licensed Product. Eonic Systems will not be liable for any lost profits, or for any claims or demands against Customer. Eonic Systems shall not be liable for any damages caused by delay in delivery, installation or furnishing of the Licensed Product under this Agreement. In no event will Eonic Systems be liable for any kind of incidental or consequential, indirect, or special damages of any kind.

8. GENERAL :

8.1 This Agreement is valid from the moment Customer has placed an order and Eonic Systems has duly executed it. This Agreement will remain in effect until Eonic Systems receives written notice of termination by Customer. Eonic Systems may terminate this Agreement, effective upon written notice thereof to Customer, if Customer neglects to perform or observe any of the terms set forth in this Agreement. This Agreement shall automatically terminate upon any act of bankruptcy by or against Customer, or upon dissolution of Customer.

8.2 If any of the provisions, or portions thereof, of this Agreement are invalid under any applicable statute or rule of law, they are, to that extent, deemed to be omitted.

8.3 This Agreement shall be governed by the laws of the State of Belgium and the relevant laws of the European Community and Customer expressly

submits to jurisdiction therein by process served by mail on Eonic Systems at the address below.

8.4 If Customer issues a purchase order, memorandum, or other written document covering the Licensed Product provided by Eonic Systems, it is specifically understood and agreed that such document is for Customer's internal purposes only and any and all terms and conditions contained therein shall be of no force or effect.

8.4 This Agreement supersedes any and all prior representations, conditions, warranties, understandings, proposals, or agreements between Customer and Eonic Systems, oral or written, relating to the subject matter hereof and constitutes the whole, full, and complete Agreement between Customer and Eonic Systems.

8.5. The Licensed Product includes a 12 months period of support and maintenance provided Customer submits his questions in written form and Customer duly returns the completed form to Eonic Systems that accompanies the Licensed Product.

RETURN A SIGNED COPY TO EONIC SYSTEMS TO VALIDATE THE 12 MONTHS MAINTENANCE

IN WITNESS WHEREOF

the parties hereto have executed this Agreement by their duly authorized representatives :

_____

CUSTOMER (print) Authorized Signature

_____

_____

_____

Address (print)

EONIC SYSTEMS, Nieuwlandlaan 9, B-3200 Aarschot, Belgium.

_____

Date

RETURN A SIGNED COPY TO EONIC SYSTEMS TO VALIDATE THE 12 MONTHS MAINTENANCE

EXHIBIT A : LICENSED PRODUCTS :

	Product	Serial Number	Qty
1.			
2.			

LICENSED SITE : (print or type)

Company : _____

Department : _____

Address : _____

State/Prov. : _____

ZIP/Postal Code: _____

Country : _____

Telephone : _____

Fax : _____

e-mail : _____

Technical Contact: _____

FOR EONIC SYSTEMS USE ONLY :

Order/Date/Invoice: _____

Date Shipped: _____

Reseller : _____

Date Reg. Received: _____

Virtuoso ™

The Virtual Single Processor Programming System

Covers :

Virtuoso Classico ™

Virtuoso Micro ™

Version 3.11

Part 1. The concepts

Creation : February 5, 1990

Last Modification : September 6th, 1996

1. Installation

1.1. Installing the software

The Virtuoso Package is available in two types of licenses. The fist one only contains the binary files while the second one is delivered with the source code. The source code is delivered on a separate floppy. For the rest of this manual we will simply refer to Virtuoso. To install the Virtuoso package on a PC hosted system, follow this procedure :

1. Insert the floppy in the drive
2. Type “install”

Follow the instructions on the screen.

This will create the virtuoso directories on the requested drive and copy the files onto the hard disk. If you want to install onto a different structures, edit the install.bat file but be aware that the supplied makefiles assumes a subdirectory structure as the one on the floppies. The installation on UNIX hosted systems is similar, but uses a tar file.

Please read the readme file first, before you proceed any further.

Next, you will need to set up the paths. You need a path to the C compiler (e.g. \tic440), the libraries (\lib) and to the directory of executable programs (\bin). When using the PC graphics library, you also need to set up a path to the Borland graphics driver (e.g. \bc\bgi.).

In the \examples directory, you can find small test programs. It is advised to copy one of these into your own directory if you start programming.

IMPORTANT NOTE :

As your board might have a different memory lay-out and interprocessor connections than those used to build the examples, please verify the memory layout and interprocessor connections (if any) so that they reflect your own board and remake the examples.

Each directory is delivered with a makefile (Borland make.exe compatible). “make' will generate a new binary. “make install” will copy the libraries to the \lib directory. “make clean” will remove all files that can be regenerated.

1.2. Kernel libraries provided

The following libraries are provided :

The Virtuoso kernel :

1. `VIRTOS.LIB` : no support for the task level debugger.
2. `VIRTOSD.LIB` : with support for the task level debugger.

Host access :

1. `CONIO.LIB` : simple console I/O, mainly used for terminal I/O
2. `STDIO.LIB` : C style I/O
3. `BGI.LIB` : Borland BGI graphics calls

To recompile the libraries, go to the `SOURCE\Virtuoso` directory and type `MAKE`. This will display instructions on how to proceed.

Note : Other libraries are provided as well. These are target dependent. See the relevant sections or the readme files.

1.3. Confidence test

```
1. Change to the EXAMPLES directory:

cd \VIRTUOSO\EXAMPLES\D1P
```

Note that \d1p is called \demo1p on older releases.

2. The sysdef definition file for the different system objects is already given. From these Sysgen will construct the .C files and .H files needed for compilation. Just invoke your text editor and view the definition file.

3. The different object definitions can now be viewed. For a better understanding it is advised to have a quick look at the relevant section in Part2 of the manual.

4. After viewing the system definition file, you can invoke the Sysgen system generation utility on the file. This generates automatically all include files, There is a node.c and a node.h file for each processor in the system. You can try as follows :

```
make
```

This will call the make utility that operates on the makefile. This will parse the system definition file, compile, link and configure the test program.

5. Run the final demonstration program, using one of the supplied *.bat program. For example :

```
run test
```

This starts up the server program on the host and boots the demonstration program onto the root processor. For example :

```
host_x -rlsz test.nli
```

Follow the instructions provided on the screen, while having a look at the relevant sections of the Reference Manual.

To start developing a new application, it is recommended to start from an example program, and to copy it to a new directory, where you will develop it.

6. The supplied test program runs as a program on a processor connected to the host. Exit from the demo with CTRL-C or terminate the server to return to DOS. While \d1p is not compiled with the debugger options, most other examples are. The debugger is started by hitting the ESCAPE key.

A good way to familiarize yourself with Virtuoso is to play a bit with the example programs. See what happens when you invoke the debugger while the benchmark loop is running, or change the source to invoke the debugger task from within another task. Using the debugger, inspect the different elements of the system while you look up the meaning of the information provided in Part 2 of the manual. You might find that the user interface is simple but remember that this way any terminal can do the job, enabling Virtuoso to be used in exactly the same way when developing software for different target processors using different host systems.

To really have a look at how a multitasking kernel works, we advise you to select the L (List the last scheduling events) at different moments when the benchmark is running. You will certainly remark the microsecond accuracy with which the Virtuoso kernel is able to schedule the application tasks. Another point to see is the protocol involved when using the server. However, in this example, it perfectly demonstrates the interrupting nature of the preemptive scheduler.

We hope you will enjoy using this product, remember our 12 months free support service and we will greatly appreciate your comments or suggestions for improving this product based on your experience in the field.

1.4. Virtuoso compilation symbols

When recompiling the Virtuoso kernel, you'll have to enable or disable the compile time switches. These are as follows :

TLDEBUG :

If defined includes support for the task level debugger.

TLMONIT

This switch does include the use of the tracing monitor.

The default delivered virtosd.lib always includes both switches.

1.5. The license agreement

1.5.1. Site developers license and runtimes

All Virtuoso products contain the same license agreement. This license agreement is basically a site developers license that gives you the right to install the software on one developer's stations at the same site. The binary version does not contains runtime royalties except those for developing. The version with source code also contains runtime royalties for 100 target processors. Above this amount, contact Eonic Systems or your distributor.

1.5.2. Support and maintenance

Any Virtuoso package is also delivered with 12 months support and maintenance. To make you eligible for these 12 months support and upgrades, you must sign and return the license agreement to Eonic Systems or to your distributor. So don't wait any longer and mail this license form today.

From now on this means you can submit any problem you would encounter by fax or by mail. If the problem is really holding you up, don't hesitate to call.

In addition, during the 12 months following the delivery, we come out with any upgrade (software or manual), we will ship you a new version (shipping cost not included). Bugfixes are always fixed when reported.

1.6. Cross development capability

As the microkernel level is close to 100 % identical for all target processors, we have seen that a number of our customers have continued to use our evaluation kit on PC for cross development even if the target system is a rack with several tens of processors. Therefore we decided to include this Borland version (binary only) with any license delivered.

1.7. The final reference

While every effort was made to have this manual reflect the Virtuoso package, the final reference is the source, especially as some target processors or boards might impose small changes. So if you are not sure about a library function, first take a look at the include files (*.h) and the examples.

2. A short introduction

2.1. The one page manual

When using Virtuoso, the programmer will develop his program along the following steps. We outline here the steps for the use of Virtuoso Classico as this entails all levels supported by Virtuoso.

1. Define the essential microkernel objects used as building blocks for the application. These are the tasks, the semaphores, the queues, the mailboxes, the resources and the timers. If the target is a multiprocessor system, the user will need to define the network topology as well. This is achieved by preparing a description file (called “sysdef”) in text format. Sysgen (normally invoked when calling the make utility), then reads the sysdef file and generate one *.c and one *.h include file per processor node. This work is facilitated by the use of a makefile and grouping the tasks in a library.
2. Write and debug the tasks, as normal independent programs that cooperate using the microkernel services.
3. Develop lower level ISRs and drivers (can be nanokernel processes);
4. Compile and link.
5. Load the target system and run the application.
6. Debug and fine-tune it using the debugger and tracing monitor.

In order to exploit the real-time features, the Virtuoso microkernel is linked with the user tasks and runs as a single executable image on each processor. This approach results in faster and smaller code, the latter being particularly important for real-time applications.

Virtuoso is not only used for single processor applications. For those applications requiring more than one processor, the tasks can communicate with other connected processors, using the communication drivers included with the MP implementations. The VSP implementations of Virtuoso also provide support to execute fully transparently remote microkernel services from within different processors, permitting to consider the whole network of processors as a virtual single processor system. The system definition file that is parsed by Sysgen contains all the hardware dependent information and effectively shields the application from the underlying hardware.

For debugging and fine tuning, Virtuoso is delivered with a task level debugger and tracing monitor. The use of these tools will help you in debugging the system as well as optimizing its performance. If as a result tasks or other objects are moved to other processors, the only work to do is to change the system definition file, regenerate the system tables and recompile the application. Developing applications with Virtuoso can be as easy as developing

any other application under straight C.

IMPORTANT NOTE

Nevertheless, please read the manual before you start writing your programs.

2.2. Underlying assumptions when programming

When developing a multitasking application on a single processor, the programmer is free to program his tasks anyway he wants as long as he follows the semantics of the microkernel services. When the target is a multiprocessor system and he wants to benefit from the transparent migration of tasks and other Virtuoso objects from one processor to another, he has to keep in mind that this imposes one important rule : not to use global pointers or global data unless he explicitly wants to exploit the performance of local access using pointers. The reason is that otherwise the program code is no longer transparent to the location in the processor network because pointers are local objects by definition. By the same token, memory allocation is a local operation as well.

Note that when using local pointers, the program might require additional synchronization or resource locking to assure the validity of the data. The latter requirements are also valid for systems with common memory. The resource management services of Virtuoso can be used to implement this protection.

Let’s illustrate this with a small example. Two tasks communicate using a mailbox. When both tasks are on the same processor, it is possible not to send the whole data from the sender to the receiver task, but only a pointer to it. Provided the user takes the necessary precautions to avoid that the data is overwritten by the sender before the receiver has effectively used it, this can provide for very good performance. However if one of the tasks is now moved to another processor, passing the pointer has become a meaningless operation. Hence this program is not scalable. Using global variables leads to similar problems. For above reason the semantics of Virtuoso reflect the non-locality of pointers. If wished, the user can exploit it for maximum performance but he will be warned because his program code will show it. In any case, using local variables always lead to safer and more modular programs and is good programming practice anyway.

Similarly embedding task functions in a single large file leads to problems when rebuilding the application. The safest practice is to compile each task function separately and add them all to a single task library. Upon linking, the linker will then only add the relevant task functions to the executable images, avoiding the generation of unnecessary large executables.

3. Virtuoso : an overview

3.1. Requirements for a programming system

Processor technology is changing very rapidly. Processors are becoming faster, microcontrollers are becoming more complex and richer in features, but developers are left with the impossible task to keep up. In addition fundamental I/O bandwidth limitations force the designers to go parallel to reach the required level of performance. The conclusion is clear : the only way to shorten the development cycle is to use tools that relieve the developer from the technology change. The ideal development tool must not only provide for faster application development by giving the programmer a head start, but should also be future proof. The requirements can be split in three areas :

1. A consistent high level API across all target processors
2. The utilities to debug and maintain the code
3. Target processor specific support for best performance.

3.2. The high level view : a portable set of services

3.2.1. A multi-tasking real-time microkernel as the essential module

In many applications sensors supply raw data, preprocessing algorithms filter and examine the data, control algorithms process the data and deduce from it control commands, while additional functions deal with user interaction, data logging or safety concerns. In most cases tight timing requirements need to be fulfilled especially if the system has to deal with events that can happen at any time, regardless of the current function being executed at that moment. A common and proven solution is to map the different functions onto separate tasks, assign priorities to these tasks and to use a standardized way of defining the interaction between the tasks. The core of this solution is the availability of a real-time microkernel that manages the timely execution of the tasks and their interactions in a way that frees the applications programmer from the burden of doing it himself. It must also be noted that for multiprocessor targets, multitasking is a must to achieve high performance because without it, it is not possible to overlap computation and calculation. As such the microkernel must correctly provide for priority driven preemptive scheduling of the tasks, permit to transfer data between the tasks in a synchronous and asynchronous way, coordinate tasks, deal with timed events, allocate memory and protect common resources. In addition, the microkernel should be small, secure and flexible while providing a very fast response. Practice has shown that in general the same set of microkernel services is sufficient.

3.2.2. Classes of microkernel services

The Virtuoso programming system is built around a second generation real-time microkernel. It provides the same API by way of a virtual single processor model independently of the number or type of interconnected processors that are actually being used from single 8-bit microcontrollers to multi 32-bit processor systems. This approach also means that the programmer can continue to use his single processor experience and start using multiple processors with a minimum of effort.

The Virtuoso programmer's model is based on the concept of microkernel objects. In each class of objects, specific operations are allowed. The main objects are the tasks as these are the originators of all microkernel services. Each task has a (dynamic) priority and is implemented as a C function. Tasks coordinate using three types of objects : semaphores, mailboxes and FIFO queues. Semaphores are signalled by a task following a certain event that has happened, while other tasks can wait on a semaphore to be signalled. To pass data from one task to another, the sending task can emit the data using a FIFO queue or use the more flexible mailboxes. While the first type provides for buffered communication, mailboxes always provide a synchronized service and permit the transfer of variable size data. Filtering can be performed on the desired sending or receiving task, the type of message and the size. Further services available with Virtuoso are the protection of resources and the allocation of memory. The microkernel also uses timers to permit tasks to call a microkernel service with a time-out. Depending on the processor type, some microkernel calls also provide access directly to communication hardware and high precision timers. The latter is used to directly measure the CPU workload. Finally, the C programmer disposes of a standard I/O, graphics and runtime library of which some of the functions are executed as remote procedure calls by a server program on a host machine.

3.2.3. The object as the unit of distribution

In a traditional single processor real-time kernel, objects are identified most often by a pointer to an area in memory. This methodology cannot operate across the boundaries of a processor as a pointer is by definition a local object. Virtuoso solves this problem by a system-wide naming scheme that relates the object to a unique identifier. This identifier is composed of a node identifier part and an object identifier part. This enables the microkernel to distinguish between requested services that can be provided locally and those services that require the cooperation of a remote processor. As a result, any object can be moved anywhere in the network of processors without any changes to the application source code. A possible mapping of objects into a real topology is illustrated in figure 1. Note that each object, including semaphores, queues and mailboxes could reside as the only object

on a node. The key to this transparency of topology is the use of a system definition file (see below). In this context we emphasize that with Virtuoso the node identifier is nothing more than an attribute of the object.

The transparent distributed operation would not work if the semantics of the microkernel services and their implementation were not fully distributed. This imposes a certain programming methodology. E.g. global variables or pointers are only allowed if the programmer very well knows that only local references to it are used and the objects referencing to it will not be mapped over more than one target processor. When common memory is used, the objects must be protected using a resource lock

.

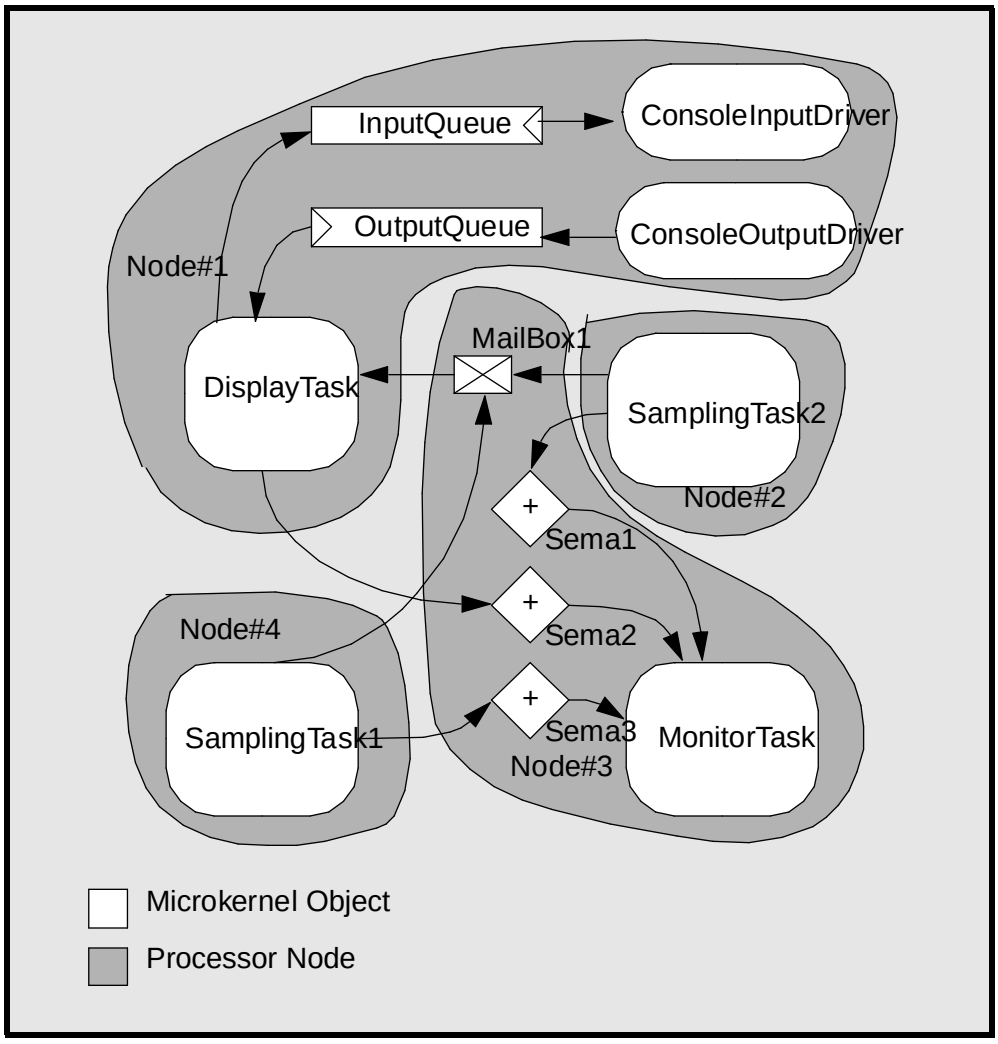


FIGURE 1 A possible mapping of objects onto a network

3.3. A multi-level approach for speed and flexibility

As any designer knows, a single tool or method cannot cover all of the different aspects of an application. In particular DSPs are increasingly used for signal processing and embedded control at the same time. This poses quite a challenge to the programming tool as it must handle timing constraints expressed in microseconds. Traditionally this meant programming in assembler at the interrupt level. One of the drawbacks of this approach is a lack of modularity and hence the sheer impossibility to build complex multitasking applications. Real-time multitasking kernels on the other hand provide for modularity but impose an unacceptable overhead for handling fast interrupts at the task level.

On parallel DSPs the situation is even more demanding than on single processor DSPs. The reason is that interprocessor communication is generating interrupts that have to be processed at the system level. Hence a minimum interrupt latency as well as interrupt disabling time is a must. As an example consider the TMS320C40 DSP.This processor requires to handle already up to 14 interrupts sources without any of them being related to external hardware.

The Virtuoso programming system solves this Gordian Knot by providing an open multilevel system built around a very fast nanokernel managing a number of processes . The user can program his critical code at the level he needs to achieve the desired performance while keeping the benefits of the other levels. Internally, the kernel manages the processor context as a resource, only swapping and restoring the minimum of registers that is needed. The different levels are described below. ISR stands for Interrupt Service Routine.

LEVEL 1 : ISR0 level

This level normally only accepts interrupts from the hardware. Interrupts need only be disabled during ISR0 (e.g. less than 1 microsecond on a C40). The developer can handle the interrupt completely at this level if required or pass it on to one of the higher levels. The latter is the recommended method as it disables global interrupts for a much shorter time. This approach allows to handle interrupts (in burstmode) at over 1 Million interrupts per sec on a C40. The programmer himself is responsible for saving and restoring the context on the stack of the interrupted task.

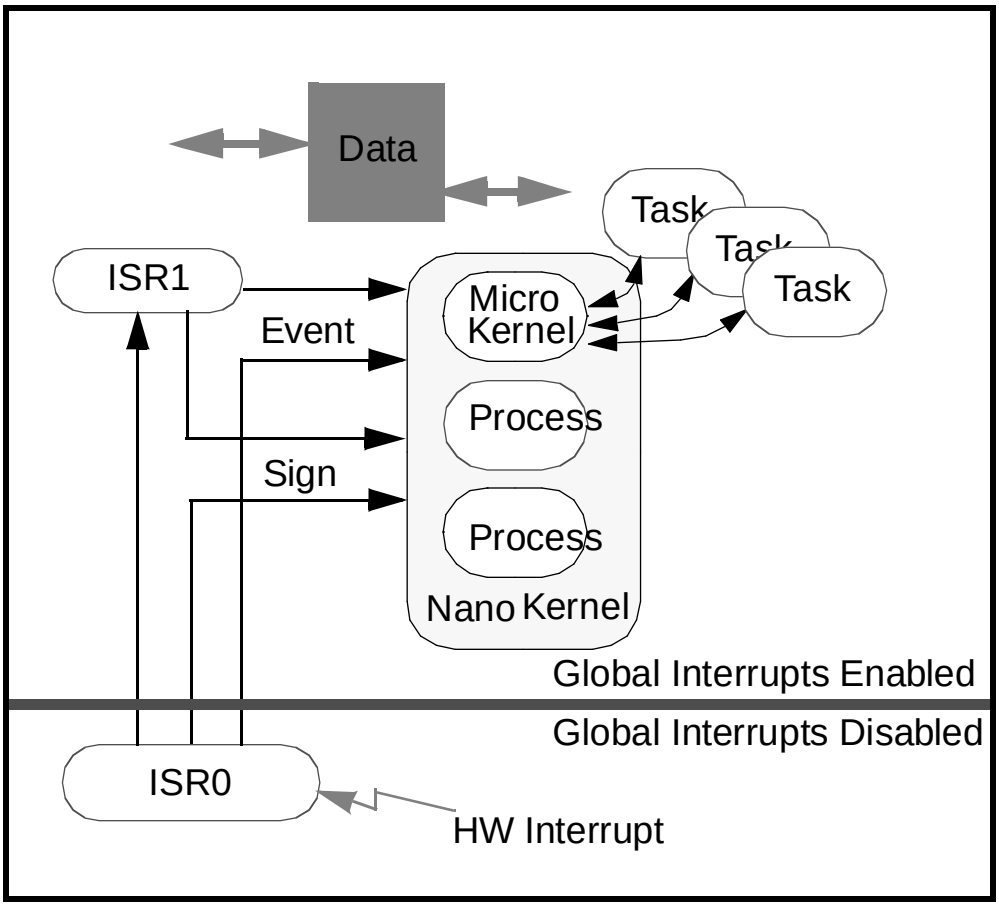


FIGURE 2 Multi level support mechanism in Virtuoso

LEVEL 2 : ISR1 level

The ISR1 level is invoked from ISR0. It is used for handling the interrupt with global interrupts enabled. An ISR1 routine permits to raise an Event for a waiting task. An ISR1 routine must itself save and restore the context but permits interrupts to be nested. An ISR1 routine can be replaced by a nanokernel process that is easier to program as the nanokernel takes care of the context switch. When the processor supports multi-level interrupts, the ISR1 level can be viewed as having itself multiple levels of priority. The use of priority however should be limited only to determine what interrupts are masked out when a given interrupt occurs.

LEVEL 3 : The nanokernel level (Processes)

This is a major breakthrough for DSP processors. The nanokernel level is composed of tasks with a reduced context, called processes. These deliver a

task switch in less than 1 microsecond on a C40. Several types of primitives are available for synchronization and communication. Each process starts up and finishes as a assembly routine, can call C functions and leaves the interrupts enabled. Normally one will only write low level device drivers or a very time critical code at this level. One of the processes is the microkernel itself that manages the (preemptive) scheduling of the tasks (see below).

The following example tested on a 40 MHz C40 illustrates the resulting unprecedented performance. Two processes successively signal and wait on each other using a intermediate counting semaphore (Signal - Task switch - Wait - Signal - Task switch - Wait). Round-trip time measured : 5775 nano-seconds or less than one microsecond per operation.

Processes have the unique benefit of combining the ease of use of a task with the speed of an ISR. In a multi-processor system they play an essential role. Without the functionality of the processes, the designer has the option or to program at the ISR level and hence often disabling interrupts because of critical sections, or to program at the C task level but resulting in much increased interprocessor latencies. This is due to the fact that in an multiprocessor system, interprocessor communication has to be regarded as high priority interrupts because if not acted upon immediately, it can delay the processor that is the source of the interrupt.

LEVEL 4 : The microkernel level (tasks)

This is the standard C level with over 70 microkernel services. This level is fully preemptive and priority driven and each task has a complete processor context. It provides a high level framework for building the application as explained in the previous paragraph. Programming at this level provides for full topology transparency and offers the highest degree of flexibility. The overhead at this level comes not as much from the use of C and the heavier register context but mainly from the heavier semantically context offered by the microkernel services.

Good programming practice dictates a concern for portability and scalability of the application. Hence, one should program as much as possible at the microkernel level. Programming at the lower levels can be justified for two reasons :

1. Faster response times;
2. Better performance.

Indeed the lowest levels will always preempt the higher levels and because of the minimum context have a lower overhead. However these levels are processor dependent (dictated by the use of assembly) and should only be used when needed as portability and scalability are lost.

Most real-time kernels only provide a single ISR level and the C task level as this is sufficient for supporting applications using standard microprocessors and microcontrollers. This is also the case for ports of Virtuoso to this class of processors.

3.4. An execution trace illustrated

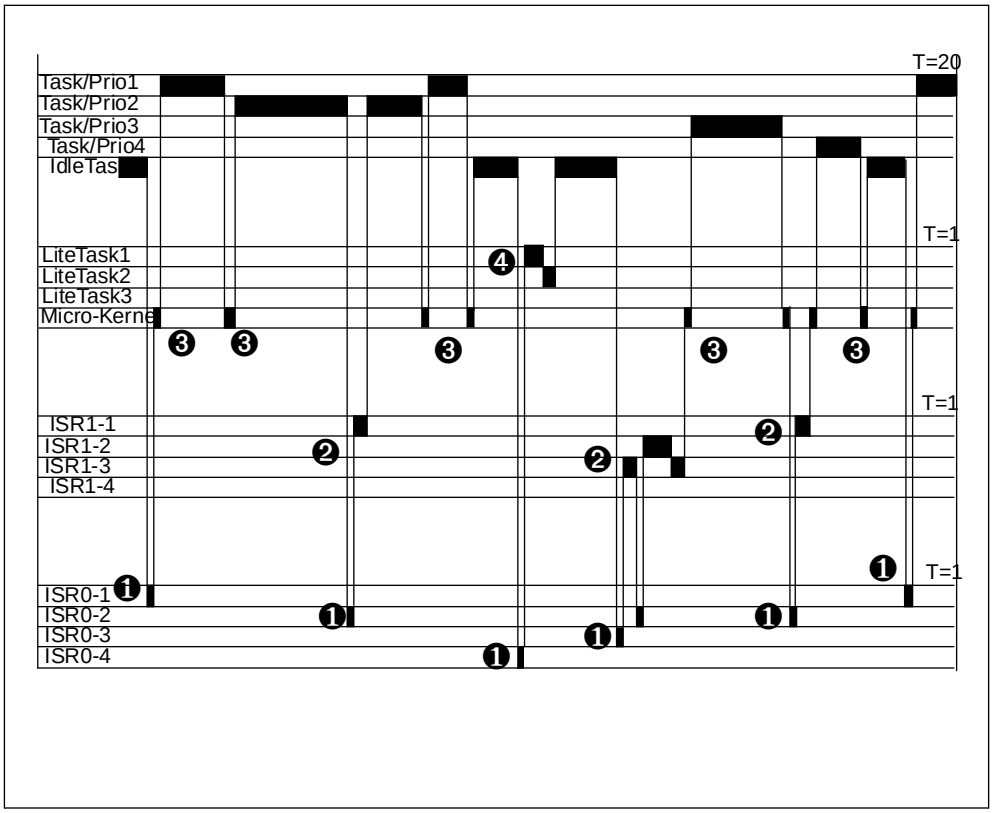


FIGURE 3 An execution trace of a hypothetical Virtuoso program segment

In Figure 2, we illustrate a hypothetical example that illustrates the interaction between the different levels. The order of magnitude of the timestep is indicated in microseconds. As can be seen any lower level has an effective higher priority than any higher level and can preempt an activity executing at any higher level.

The processor accepts an interrupt (1). This can be an external interrupt, a comport interrupt or a timer interrupt. The interrupt is passed on to a higher level for further processing. This disables interrupts for less than one microsecond on a C40, or about 1.5 microseconds on a 96K.

An ISR enters ISR1 level (2). This can only be to further process an interrupt accepted by the ISR0 level. On processors with support for multi-level interrupts an ISR executing at level 1 an be preempted by an interrupt of a higher priority.

The microkernel is invoked (3). This can happen as a result of signal coming from an ISR0, a microkernel service request from a task, a task entering a wait state or an event raised by an ISR1 or a nanokernel process. The microkernel is a nanokernel process that waits on any of these events to happen.

A nanokernel process is executed (4). In this example an ISR0 could have triggered a receiver driver that passed on the packet to a transmitter driver to forward the packet to another processor.

3.5. Processor specific support

If a target processor has specific hardware features that enable the programmer to take advantage of, most likely Virtuoso provides a set of services that exploit these features. E.g. high precision timers and drivers for communication ports. See iface.h and the binding manual in part 3 for details.

4. Functional support from Virtuoso

4.1. Introduction

Virtuoso is probably the first programming tool that provides hard real-time capabilities to programmers of parallel systems without abandoning the experience one has gained from programming multitasking applications on single processor systems. As such Virtuoso comes in two versions. The first one is dedicated to single processor targets, while the second version provides for a transparent distributed operation. In conjunction with the associated programming tools, it is also a tool that provides for easy programming of parallel processing systems in general. For these reasons we call Virtuoso the virtual single processor programming system. While the runtime library that comes with a compiler only provides for low level services, Virtuoso provides for a complete programming framework. On the other hand Virtuoso is much smaller and faster than a full operating system, because it only concentrates on the services needed for a real application and leaves the development environment to the host station. Virtuoso can be regarded as a real-time operating system where all objects (tasks, etc.) are defined at the initialization of the system. In the future, Eonic Systems will release versions that support dynamic creation and deletion of system objects, as well as fault tolerant versions.

During the design, major efforts also went into making sure that Virtuoso is a future proof programming tool. With today's very fast technological changes this is a necessity, because technology can evolve faster than the time it takes to design an application. Therefore, choices were made that enable the user to develop his applications mostly independently of the technology he uses. This was achieved by opting for the use of portable ANSI C as a programming language and by adopting a virtual single processor model even if multiple processors are involved. The fact that we have been able to port Virtuoso to new processors in less than two weeks proves the point.

4.2. Parallel processing : the next logical step

Some people regard parallel processing as a new technology. The question is whether this is really so. In our opinion, this is more a natural result of the evolution of technology. The point is that because of the growing level of integration of VLSI devices, we have now mass volume production of complete systems on a chip. In fact, a typical high end processor is now a completely self contained computer with CPU, memory, specialized coprocessors and even network facilities. As a result, the production cost of computers-on-a-chip has become so low that we can start to use them as components. Most

of the systems that use processors in numbers aim at providing more processing power for a lower cost, simply because providing the same processing power in a single very fast processing unit has reached a technological barrier. This barrier is mostly an I/O barrier. For multiprocessor systems, this can be the common bus. But even on single processor systems the speed of the processor can be so high that no memory exists so that the processor can operate with zero wait states. Today we can also apply computers as components in application areas traditionally dominated by dedicated logic. The reason being that in most applications the single chip processors are more than fast enough even if the functionality is provided by means of a program. As a result, it is now perfectly possible to replace traditional hardware logic or circuitry by a reprogrammeble processor. While this does not provide ultimate performance versus hardware, this is more than often offset by the fact that one gains flexibility.

This shows that while every design tries to optimize the performance/cost function, in practice one must distinguish the different aspects of the two factors. Performance and cost mean different things depending on the real application at hand and depending on the particular phase of the product life cycle. As such, a safety critical system has very different design criteria than a parallel supercomputer, although the same components might be used. In essence, to really estimate the performance/cost function one should integrate over the whole life cycle of the application. The latter is broader than simply the product life cycle because current technology is evolving so fast that even during development it might make sense to change the technology in use. The reason being that designing the application might take longer than it takes the silicon industry to generate a new generation of processors.

Therefore, in general the statement is that the goal must be to maximize the use of the resources (getting best “performance” out of the used material) for a minimum life cycle cost (development, maintenance and upgrade costs) of the application. The emphasis is on the application and not on a particular system. A system today might use processing components that can change over the life cycle of the application.

In the light of above, when designing an application today we are faced with two major challenges : Firstly, the very rapid change of technology. Secondly, while processors are becoming components, programming them to work together is still a major intellectual challenge, to be compared with the design of any complex system where different subunits interact.

What Virtuoso delivers is a tool that helps to meet these challenges in terms of programming such a system. That is, it was designed to be able to solve hard real-time problems. This kind of problem is the most difficult and more general case when compared with other systems where the real-time requirements are less stringent. In order to keep up with the technology,

portability was a major issue. The use of C is consistent with this objective as today every new processor is released with a decent C compiler (even DSPs). Finally, the problem of parallel and/or distributed programming has been solved by the adoption of a virtual single processor model. Based on a message passing mechanism, this is a very general approach that can be ported to any environment (common memory, local memory and LANs).The result is that Virtuoso provides for portable applications from simple 8bit microcontrollers to systems with a large number of high end 32bit processors. It is also universal (independent of the processor or of the communication medium) and scalable (if more power is needed, one adds processors and redistributes the workload).

4.3. What is (hard) real-time ?

Real-time systems are inherently more difficult to design because they must not only deliver correct answers, but they must also provide the answers at the right and at the predictable moment in time. Failing to meet these hard real-time deadlines, can lead to fairly innocent effects as well as to catastrophes (example : airplane). We also know soft real-time systems. These systems must deliver the answer most of the time within a statistically defined timing interval. Missing some of these intervals is acceptable in most cases. A typical example is a terminal. One expects a reaction of the system within 2 seconds. Faster is better but occasional longer reaction times are tolerated, even a complete reboot if this is not needed too frequently (e.g. your own PC). Virtuoso was designed to be able to meet hard real-time requirements, but this also entails the capability to meet soft real-time requirements! How did technology provide solutions to meet hard real-time requirements?

A simple and crude approach is to use enough resources without imposing limits to solve the problem. Theoretically, this is like using a Supercomputer to add 2+2 in all cases where that operation is an important aspect of the application. Clearly this is an overkill and a waste of resources. It is also very likely that it won't solve the problem as some applications simply require a different type of solution (for example a distributed system). Or worse when the reaction time is important, often fast processors are very bad at the interrupt response level. The key is to develop smarter algorithms to solve the problem. In the past this has been done with programming constructs like super loops. In a super loop, all possible events are polled in a large loop, each branch of the loop then handling the event. While this approach can solve some soft real-time applications, it is not very flexible as a single change to the system can require a complete review of the program as the whole program's timing behavior is affected. For hard real-time problems, this means that the processor load had to be kept low so that in most cases an acceptable performance can be obtained.

Things changed when people started to realize that the best thing to do is to model the real world problem more directly. The problem at hand can very often be described by determining the different functions and agents of the system. The better algorithms then map each function into a so-called task and provide the capability to give the processor to the task that needs it most (so called preemptive scheduling, as it enables that a given task is preempted in order to switch to another task). This implies the use of a criterium, in this case the assignment of a priority to each task.

A task is to be considered as an independent unit of execution that interacts with other tasks or with the outside world using a predefined mechanism. Conceptually, tasks must be looked upon as executing simultaneously or overlapping in time. This means that on a single processor, a task switching algorithm has to be implemented because the processor can only execute one thing at a time and the concurrent behavior of the tasks must be simulated. For a long time processors were rather slow so that task switching was a heavy operation. Current microprocessors are much better at it and some, like the transputer, even have direct support in hardware for doing it. Nevertheless, real-time kernels were brought to the market to provide this kind of solution in a generic way.

In the solutions above, priorities are used by the task scheduler because dealing directly with time bound constraints is very difficult. Some better algorithms deal directly with time by using a deadline scheduler. This kind of scheduler assigns the processor to the task with the nearest deadline. This algorithm can also provide better performance under higher processor loads. The Extended and Dynamic Support Package of Virtuoso will use this kind of scheduler.

4.4. The high demands of Digital Signal Processing

Digital Signal Processing often impose demands that cannot be fulfilled by traditional processors. Therefore Digital Signal Processors (DSPs) have architectures that enable faster execution of algorithms combined with fast handling of interrupts. As a result most DSPs have built in circuits that execute multiplication and addition in a single clock cycle, have multiple buses for parallel data access and have DMA and/or communication hardware on the chip. The combination of all these features imposes quite a challenge to the software environment, especially in the context of the use of a C compiler that was not always thought out with these demands in mind. Most real-time kernels operate at the C level only and have an unacceptable overhead for handling the high demands on DSPs. The Virtuoso system has solved this problem by introducing an open four level support that tries to manage the context as a resource, trading in performance for flexibility but only when the application needs it at a certain point.

4.5. A first conclusion

The real benefit from the use of Virtuoso is that it optimizes the use of the resources (in this case mainly the processor) in a way that is fairly independent of the processor technology. A question remains : how does one design an application with a real-time kernel ?

The answer is that real-time kernels have evolved because practice has shown what services a real-time kernel should provide. This means that one divides up the application into tasks that interact by different methods. Most interactions can be classified as “signals” (= event flags) and “communication”. As each task is given a priority, the scheduler then executes the task that is runable with the highest priority first. The orderly execution of the program is achieved through the interaction mechanisms. A major benefit is that the different functions are isolated in tasks, while the interaction is also well defined. The problems have more to do with erroneous programming of the interactions so that they can eventually block the system (called deadlock). Deadlock is not a property of the use of a kernel (at least it shouldn’t be), but it is a programmer’s error.

In addition a real-time kernel must provide the means handling time based events and the protection of common resources. Virtuoso has the right kernel services for this. Nevertheless, while Virtuoso is based on a priority based scheduling algorithm, we plan to introduce some refinements. The first one is the use of priority inheritance. This mechanism is useful when several tasks use the same resource. The basic algorithm will then temporarily assign a higher priority to lower priority tasks when they are blocking a higher priority task from running because they have locked on a resource. In order to implement support for earliest deadline scheduling, we have opted for a simple scheme that works in conjunction with the priority based scheduling. This is achieved through additional kernel services that permit to bind the execution of a task to a given time limit.

4.6. Parallel programming : the natural way

Some people still think parallel programs are hard to write. This attitude is understandable if one realizes that older computer technology has more or less dictated the sequential programming style. Sequential programming languages reflect the one-instruction-at-a-time functional behavior of the older sequential processors. Even simple multitasking was a costly function. The net result is that most people have been forced for years to squeeze their problem into the constraints of a sequential programming language. Most problems however, especially in the world of control applications, are inherently parallel. For example, a system will sample external signals, process them and then output something to an external device. In this generic

example, it is easy to distinguish three functions. So what is more natural than to isolate these functions as three tasks ? The interaction between these tasks is also fairly easy to describe if one defines these interactions as local actions of each of these tasks. How much more difficult would this have been using a sequential programming style ? Nevertheless, each task's internal function is probably best described using a sequential notation. As such, most multitasking and real-time kernels have adopted this scheme. Virtuoso is no exception to that except that Virtuoso also permits the application to be distributed over several processors, so that eventually all tasks can really execute in parallel.

4.7. About objects and services

4.7.1. The Virtuoso microkernel objects and the related services

As described above, Virtuoso provides a tool to organize the behavior of a real-time program that is composed of several tasks. The philosophy behind this is one of tasks that coordinate and synchronize through the common use of so-called microkernel objects. Tasks themselves are also microkernel objects but they play a dominant role. Together with the kernel they are the originators of all the actions that use the microkernel objects to achieve the desired result. Each type of microkernel object can be looked upon as being part of a Class on which different operations are permitted. Each operation must follow the specified semantics. Figure 1 shows the relationship (independent of where the objects are located in the system) between a number of tasks and the way they use objects to synchronize and communicate. In a single processor system, all these objects are located on the same node, but in a parallel processing system, Virtuoso permits these objects to be located on any processor in the system, except when they are tied to the use of specific external hardware. This was possible through the use of unique names for the objects. On the next pages we describe these classes in a general way.

4.7.2. Class Task

4.7.2.a. The task as a unit of execution

In Virtuoso, a task is a program module which exists to perform a defined function or a set of functions. A task is independent of other tasks but may establish relationships with other tasks. These relationships may exist in the form of data structures, input/output, or other constructs. A task executes when the Virtuoso task scheduler determines that the resources required by the task are available. Once it begins running, the task has control of all of the needed system's resources. But as there are other tasks in the system, a running task cannot be allowed to control all of the resources all of the time.

Thus, Virtuoso uses the concept of multitasking.

Multitasking appears to give the processor the apparent ability to be performing multiple operations concurrently. Obviously, a processor cannot be doing two or more things at once as it is a sequential machine. However, with the functions of the system segregated into different tasks, the effect of concurrency can be achieved. In multitasking, each task once given operating control either runs to completion, or to a point where it must wait for an event to occur, for a needed resource to become available, or until it is interrupted. Efficient use of the processor can be obtained by using the time a task might wait for an event to occur to run another task.

This switching from one task to another forms the basis of multitasking. The result is the appearance of several tasks being executed simultaneously.

4.7.2.b. Priority and scheduling

When several tasks can be competing for the resource of execution time, the problem is to determine how to grant it so that each gets access to the system in time to perform its function. The solution most often used, is to assign a priority to each task indicative of its relative importance to other tasks in the system. Virtuoso uses a fixed priority scheme in which up to a user defined maximum number of tasks may be defined. Tasks which have a need to respond rapidly to events are assigned high priorities. Those that perform non time critical functions are assigned lower priorities. Without a kernel, most processors only know one priority level. The exception is the transputer that uses two priority levels. In itself this is not sufficient to solve all hard real-time applications, but it has proven to be helpful when designing Virtuoso for the transputer. Virtuoso provides an efficient software based way of assigning multiple priorities to tasks.

It is the priority of each task that determines when it is to run in the hierarchy of tasks. When a task may run depends on what is happening to the tasks of higher priority. Tasks are granted execution time in a strict descending order of priority. While executing, a task may be interrupted by an event which causes a task of higher priority to be runable. The lower priority task is placed into a temporary state of suspension and execution control is granted to the higher priority task. Eventually, control is returned to the interrupted task and it is resumed at the point of its interruption. Thus, when any task is given execution control, no higher priority task can be in a runable state. This is an important point to remember. When all application tasks are in an unrunable state, control is granted to the null task.

The null task is a do-nothing task which allows the system to run in an idle mode while waiting for an event that will resume or start a higher priority task. The null task is always the lowest priority task in the system and is

always runable (the “main” part of your application program). It is a required part of the system and may be customized by the user if special needs exist (e.g. to halt the processor when idling). The only condition is that it must never be blocked from running. In Virtuoso, the null task is also used to measure the workload of the local CPU.

4.7.2.c. Task execution management

Virtuoso provides a number of services that directly control the execution state of a task. These services can operate asynchronously of the current execution thread of the task. A task normally starts its life by a KS_Start() call. This call can be invoked during start-up of the system (if defined so in the system generation file) or at runtime from within another task. During its life, a task can be suspended (i.e. blocked from running) and resumed afterwards. When suspended, the task has no chance to become executable, so this service must be used with caution. With the KS_SetEntry() service, one can change the actual function of a task at runtime. After a subsequent KS_Start() call, the task will then have a different function. Virtuoso also permits to group tasks and has services that operate on a whole group within a single call. Note especially the microkernel services KS_SetPrio() and KS_Yield(). These enable to change the order of execution (in order of priority) at runtime. When tasks happen to have the same priority, they are scheduled in a round-robin fashion, i.e. they run until they are descheduled or until they yield the CPU voluntarily. The user should base his program on any order of execution when assigning equal priority to a number of tasks.

In addition, when a task issues a microkernel service that is not immediately available, Virtuoso will put the task in a wait state. During the wait state, another task can be rescheduled to continue its execution. The wait state and suspend state can be coexisting. It is very important to know that when several tasks are in a wait state while requesting the same service in conjunction with the same microkernel object (e.g. a resource), they are inserted in the wait list in order of their priority they had at the moment the wait state was entered.

The figure below illustrates the possible Task states and the transition possibilities.

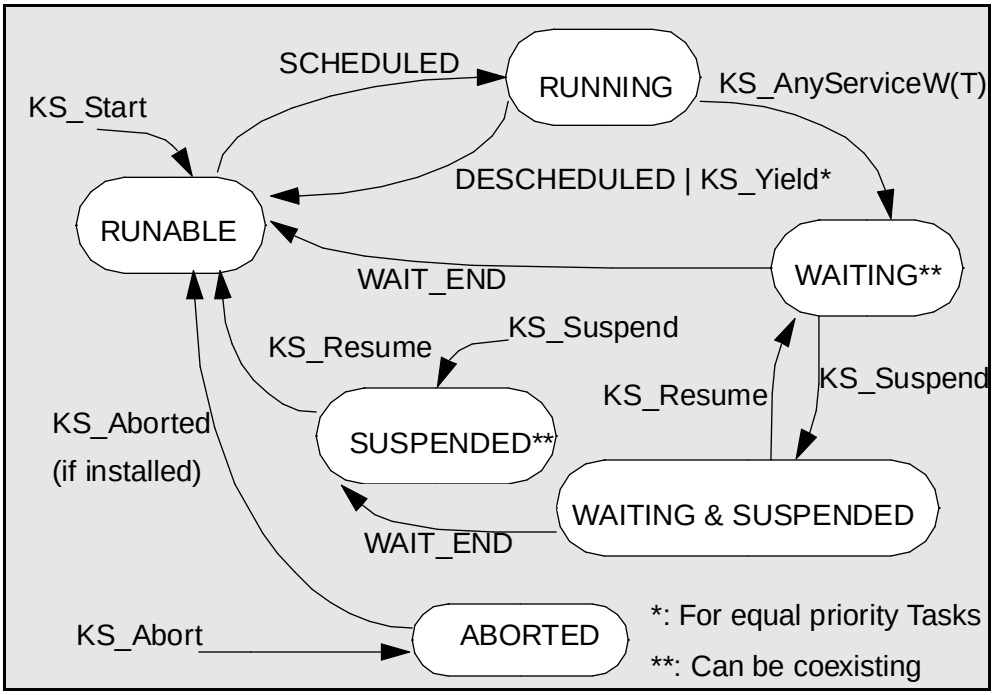


FIGURE 4 Task state transition diagram

In normal conditions, tasks should never terminate unless the system's operation as a whole is terminated because the application requires it. Therefore, aborting a task needs to be done with the right precautions. When a task is aborted, all traces of its previous execution should be cleared from the system. As such this is a service only to be used with care as it not only takes much time, but the application must be written so that this operation will not generate any unexpected side-effects (like tasks waiting on messages from the aborted task). While the Virtuoso microkernel can clear the task, the application might require specific actions. This is handled with the KS_Aborted service call. When used, the microkernel will save the specified task aborting entry point in a separate area. When the task is aborted, Virtuoso transfers control to this entry point and executes it with the current priority of the task. In the Abort function the user should deallocate all acquired system resources such as memory and timers. Application specific measures can be programmed here to assure the continuity of the application.

4.7.3. Class Timer

This class of calls permits an application task to use a timer as part of its function by allocating a Timer object. From then on, the timer can be started to generate a timed event at a specified moment (one shot) or interval

(cyclic). This event can then signal the associated semaphore. Timers are mainly used to regulate the execution of tasks with respect to a required timely behavior. While the timer objects are relatively simple in nature, using them correctly can be tricky. The reason is that the designer not only needs to know when to start a certain task, he must also take account of the worst case execution delay of his task. A real system involves meeting deadlines and missing a deadline simply means that the system has failed.

Virtuoso efficiently manages multiple timers using an ordered linked list of pending timer events. A timer for an event is inserted into the linked list in accordance with its duration. A differential technique is used so that the timer with the shortest time to expiration is at the head of the list. Timed events may even be simultaneous. Microkernel services for scheduling and cancelling timed events are an integral part of the microkernel.

Most microkernel calls that involve the cooperation of another task or device can be invoked to wait until synchronization is established. As this can cause some tasks to be blocked indefinitely, it is possible to limit the waiting time to a time interval provided as a parameter of the microkernel service that caused the task to wait. This time interval is called a time-out and makes use of the microkernel timers. When the time-out expires, the service returns with an error code.

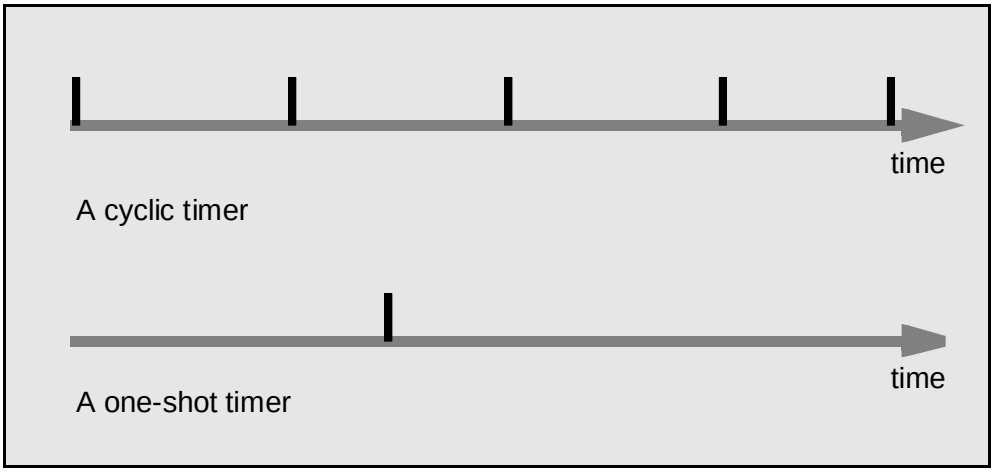


FIGURE 5 Types of timers

4.7.4. Class Memory

In any system, memory is a resource for which tasks are competing. Memory management is an area where various techniques can be applied. Many techniques are very fast and utilize elegant models for allocation and deallo-

cation. Unfortunately, most have a common flaw, one which is very serious in a deterministic environment: fragmentation. Fragmentation refers to the disorganization which occurs when memory is allocated from and released to a common memory pool. At some moment it is possible that a request for a certain block size fails because not enough contiguous memory exists even though the total amount of free memory exceeds the requested block size. There are re-organization and garbage collection techniques abounding but they fail one major test of a real time system. The failure is in the time required to repair the fragmentation. The amount of housekeeping time cannot be controlled or even predicted. Consequently, the real time processing of an event needing the memory is indeterminate if normal memory allocation techniques are used.

Virtuoso implements a form of dynamic memory management that partitions the memory in static blocks. At system definition time, the available RAM memory is divided into one or more memory maps where each map is logically made up of a number of fixed size blocks. The size of the blocks and number of blocks per map is user defined. When a task requires memory for local storage by the KS_Alloc() microkernel service, Virtuoso will allocate memory (a single block) from the specific partition memory pool. A pointer to the base address of the block is returned to the requesting task. If no memory is available in the specified partition, a NULL value (0) is returned. When the memory is no longer needed by a task, the block can be released back to the memory pool via the KS_Dealloc() microkernel service.

The partitioned memory technique, however, does not in itself prevent the loss of an event due to unavailable memory. It is the responsibility of the system designer to configure the system to have enough allocatable memory in the appropriate partitions for the worst case conditions. Note that dynamic memory allocation and deallocation is still possible by the use of resource protection.

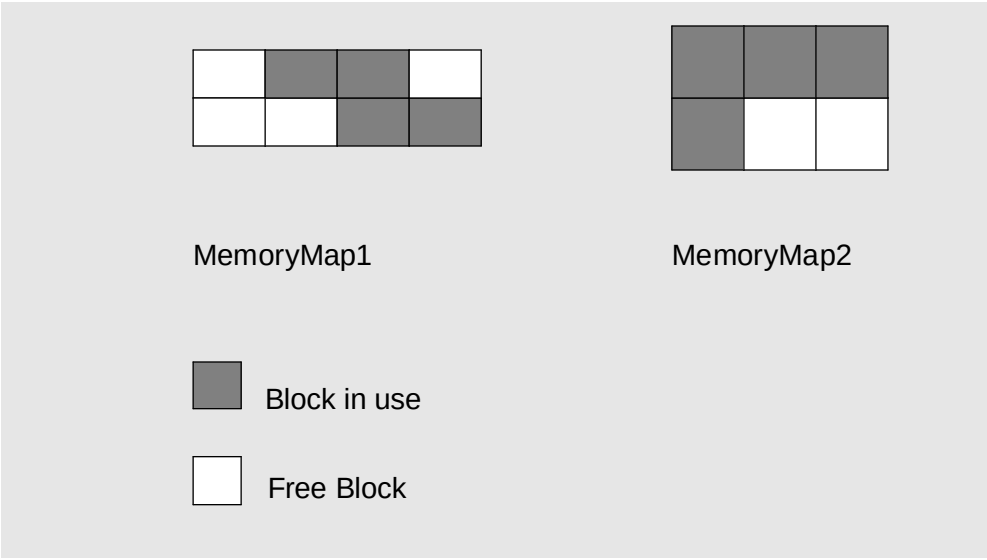


FIGURE 6 Virtuoso allocates memory in user-defined fixed block sizes.

4.7.5. Class Resource

The resource protection calls are needed to assure that access to resources is done in an atomic way. Unless the processor provides real physical protection, the locking and unlocking of a resource is in fact a convention that all tasks using a resource must follow. A task can get the exclusive use of a resource by using a KS_LockW() service call on it. After its use, the resource must be released with an KS_Unlock() call. With this mechanism all critical objects can be protected.

4.7.6. Class Semaphore

Semaphores are used to synchronize/handshake between two tasks and/or events. A signalling task will signal a semaphore while there will be another task waiting on that semaphore to be signalled. One can wait on a semaphore with a time-out or return from the wait if no semaphore is signaled. This can be useful to make sure that the task doesn't get blocked. Manual resetting of the semaphore is also possible. With v.3.0 counting semaphores are used. They permit the concurrent signalling of a semaphore while a (priority ordered) waiting list is kept of all tasks waiting on the semaphore.

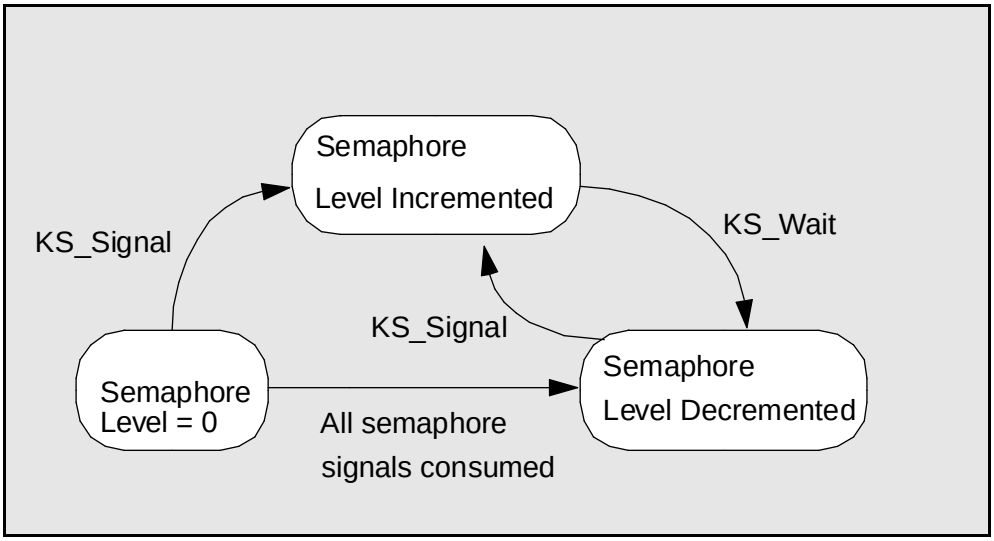


FIGURE 7 Semaphore level transition diagram

4.7.7. Class Message

Messages are used between a sender and a receiver task. This is done using a mailbox. The mailbox is used as an intermediate agent that accepts message headers. Message headers only contain the necessary information about the actual message. This permits to match send and receive of a sender-receiver pair. In the single processor RTXC, in reality a pointer is passed from sender to receiver task. With Virtuoso an actual copy of the data is made. This data is not part of the message header but is referenced by it. In practice this involves more than one step. In a first step the message is transmitted, while in a second step the data referenced by it, is transmitted.

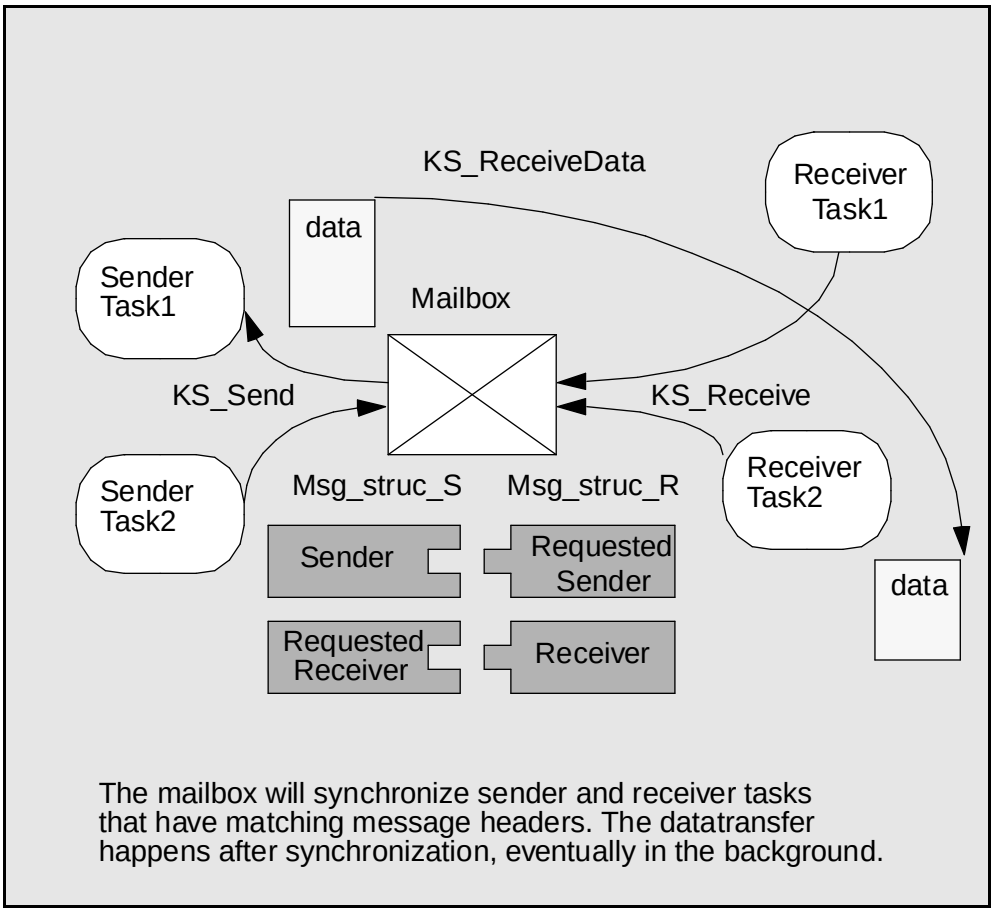


FIGURE 8 The mailbox permits a selective transport between two tasks.

This is to ensure that the receiver task is ready to accept the data, while avoiding that routing buffers are needlessly being filled up. With the KS_ReceiveW() call, the two operations are done automatically by the microkernel, but the user must know beforehand where he wants the data to go. He has the possibility to decide where to put the data by receiving the message header separately. Depending on its contents, he can then decide on where to put the data by issuing the ReceiveData() call.

Messages work with arbitrary sizes and permit a selective transport between sender and receiver, including the specification of the priority of the message.

In practice it was not possibly to use the single processor semantics of most real-time kernels within Virtuoso. The reason is that one can't pass pointers from processor to processor. While the microkernel could test for locality, we are forced to pass a copy in all cases because otherwise the programmer looses the portability of his code (he would need to test to see if the mes-

sage came from a local or remote task and handle the message accordingly). This would complicate his code too much while he would make a copy in most cases anyway. So if he only wants to pass a pointer (because he knows beforehand that he is on the same processor), he will need to pass that pointer explicitly. For this reason the header contains a 32bit Info field.

4.7.7.a. Once-only synchronization : the KS_MoveData() service

The message mechanism is a very secure and flexible mechanism because it synchronizes between sender and receiver before the receiver determines the actual datatransfer. In repetitive situations, the synchronization and its associated overhead can be eliminated using the KS_MoveData() service. This service requires that all elements of the datatransfer have been fixed once beforehand. As an example consider an image generating task that copies the resulting image always to the same address in video memory. The service can be invoked from anywhere in the system and will copy the data for a given size from a source pointer on a given node to a destination pointer on a node. When both pointers reside on the same processor, the operation is performed using a straight memcopy().

FIGURE 9 The KS_Move Data() acts like a distributed memory copy operation

4.7.8. Class Queue

Queues are also used to transfer data (or whatever the data represents) from any task to any other task but here the data is actually transferred in a buffered and time ordered way. The advantage is that no further synchronization is required between the enqueuing and the dequeuing task, permitting the enqueuer to continue (unless the queue was full). Another advantage of this mechanism is that a queue also acts as a “port”. For example to access the console from any node in the system, one simply enqueues or dequeues the queue that is associated with the console. If used in conjunction with a resource (to ensure the atomicity of the protocol), this permits easy imple-

mentation of a distributed service, such as the graphical server that comes with Virtuoso.

While the single processor version impose no limit on the size of the elements, in the distributed version the size of the entries have been limited to 24 bytes for performance reasons. Buffered communication for larger sizes can easily be handled by combining queues to pass the parameters and to use the KS_MoveData() to move the actual data.

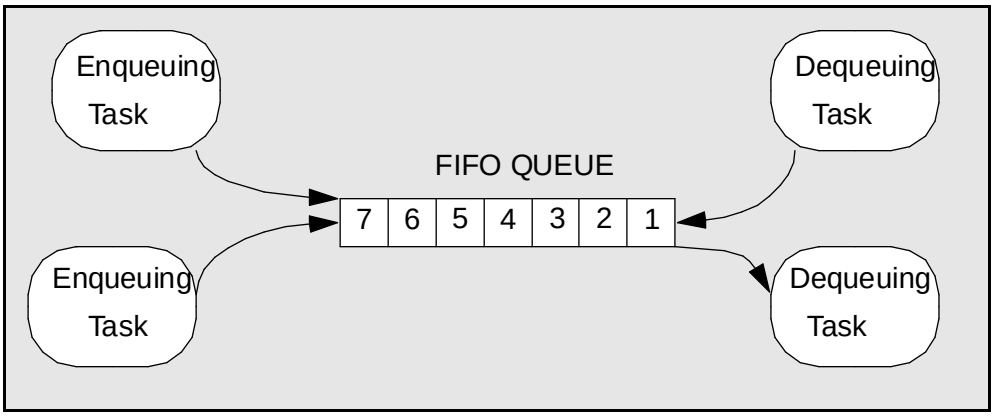


FIGURE 10 A FIFO queue acts as an ordered buffer between tasks.

This summarizes the normal microkernel objects that are used at the task level to implement an application. Virtuoso also provides a set of additional functions that can be used in conjunction. These are categorized in two classes :

4.7.9. Class Special

These include less trivial operations. The KS_Nop() call is only there for benchmark reasons, while the KS_User() permits the application programmer to run a function at the same priority of the microkernel. The latter one must be used with caution as it disables preemption during the execution of that critical function.

4.7.10. Class Processor Specific

Today's processors often have a number of additional features or peripheral device support built in on the chip. Because most of these are different from processor to processor, we have listed them here. These include : the capability to measure time with a high precision and hence the capability to measure precisely the CPU workload. Other microkernel services provide

support to use the communication ports of the chips in an easy way. Finally, Virtuoso also provides low level support for enabling and disabling interrupts so that a Task can wait on an Event that is generated by the Interrupt Service Routine. As most of these features are close to the hardware, see the next paragraph for more details.

4.8. Low level support with Virtuoso

The unit of execution of Virtuoso is the task. The coordination of Virtuoso tasks by the various objects permit a high level design of the application independent of the actual target processor the application is running on. From the viewpoint of the programmer, the task is a fairly large unit of execution. This grainsize must be large enough so that the overhead of requesting microkernel services and the resulting context switches is acceptable. A standard Virtuoso task is written in C and task swapping implies that a full context is being saved (partly by the C compiler upon the function call and partly by the microkernel to save the rest of the registers).

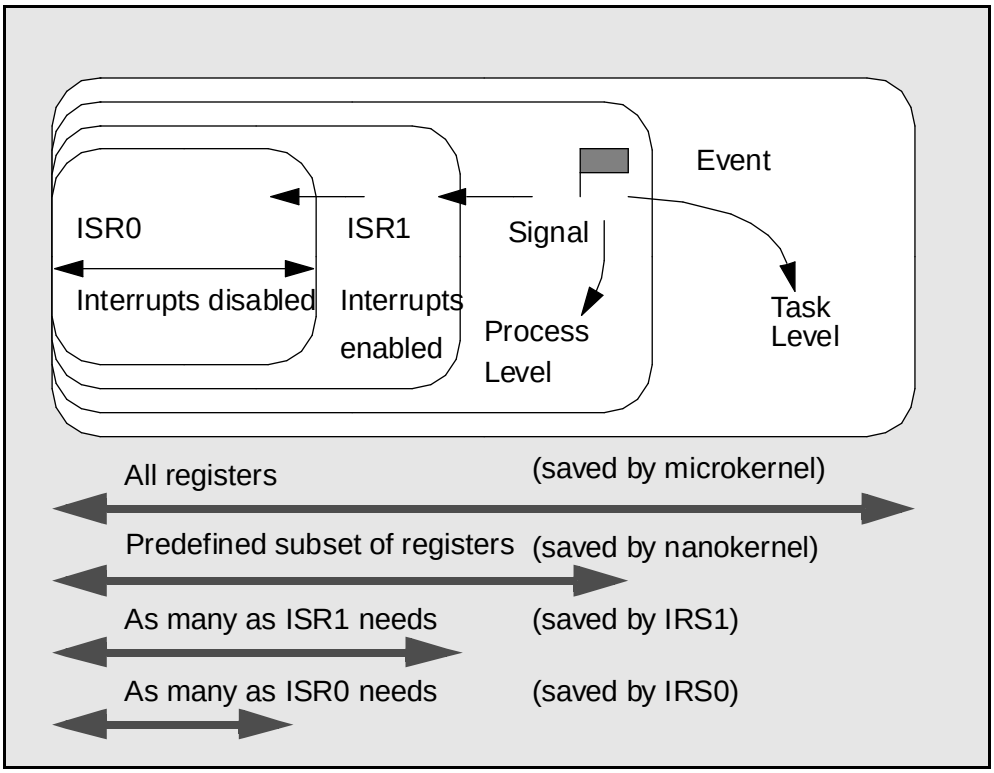


FIGURE 11 The four processor contexts considered by the Virtuoso system

In practice, especially on DSPs and high end RISC processors, one also needs units of execution with a smaller context and hence less context switching time. These can only be obtained by writing assembly code (unfortunately, we don't control what context the compiler uses).

The Virtuoso kernel manages each part of the "context" as if it were part of the CPU resource allocated to a unit of execution. As such, only those parts that are required are saved and restored by the kernel.

In total, Virtuoso distinguishes four levels :

1. The ISR0 level; (ISR with global interrupts disabled)
2. The ISR1 level; (ISR with global interrupts enabled)
3. The Virtuoso nanokernel process level;
4. The Virtuoso microkernel C task level.

4.8.1. The ISR levels

At the ISR level, the user controls himself how much context he uses and he is responsible himself for saving the context. In practice this can be 1 or 2 registers. ISR's talk directly with the interrupt hardware of the processor. The user should in most cases perform at most two actions at the ISR level :

1. Accept the interrupt (e.g. at the ISR0 level)
2. Handle the interrupt, if needed (e.g. at the ISR1 level)

On most processor types (e.g. microcontrollers) these two levels are not distinguished. When the ISR has completed a cycle (e.g. filled up a buffer area), he must signal the microkernel by raising an event. The task that should further process the data, must wait on it by issuing a KS_EventW(ISR) microkernel service. When the event is raised, the task becomes runable again. Note that interrupts are disabled during the execution time of the ISR if this method is used. Hence, ISR's should be kept short.

In order to accommodate more demanding applications (typically to be found in the DSP area), interrupt levels were split. In these implementations, the ISR0 only accepts the interrupt and then leaves further handling of the interrupt to a level were interrupts are enabled again. This is a very short operation, hence interrupts are only disabled for a few clock cycles. In the next step of processing the ISR1 level can be used. The ISR1 interrupt handler will then typically fill up a buffer and transfer control to a waiting task (often the driver associated with the hardware that generated the interrupt). Note than when the ISR is very short, it is better to execute everything at the ISR0 level, because the transition to the ISR1 level requires a few processor cycles. Also consider that some processors have up to 32 priority levels (with interrupt capability) for handling interrupts. This basically means that one

disposes of ISR0 up to ISR32 to handle interrupts.

Instead of using an ISR1, the interrupt can be handled using a nanokernel process. This has the low register requirements of an ISR but the programmer has additional services available that permit him to build up the interrupt handling using other processes as a true multi-tasking program. This is very useful for fast interrupt driven drivers. For example the interprocessor communication drivers of Virtuoso are written as nanokernel processes.

At the end, but not necessary so, the ISR1 or the process can signal an event on which a normal task is waiting to further process the interrupt at the C level.

An important point to note is that Virtuoso does not impose any use of a certain level to address the application requirements. As the diagram below illustrates, the user can choose any combination of levels to achieve the required performance. He should try to program as much as possible at the microkernel level as this level is fully portable and scalable. Lower levels require more assembly, provide less flexibility but can provide the time critical performance needed for very fast reaction to interrupts.

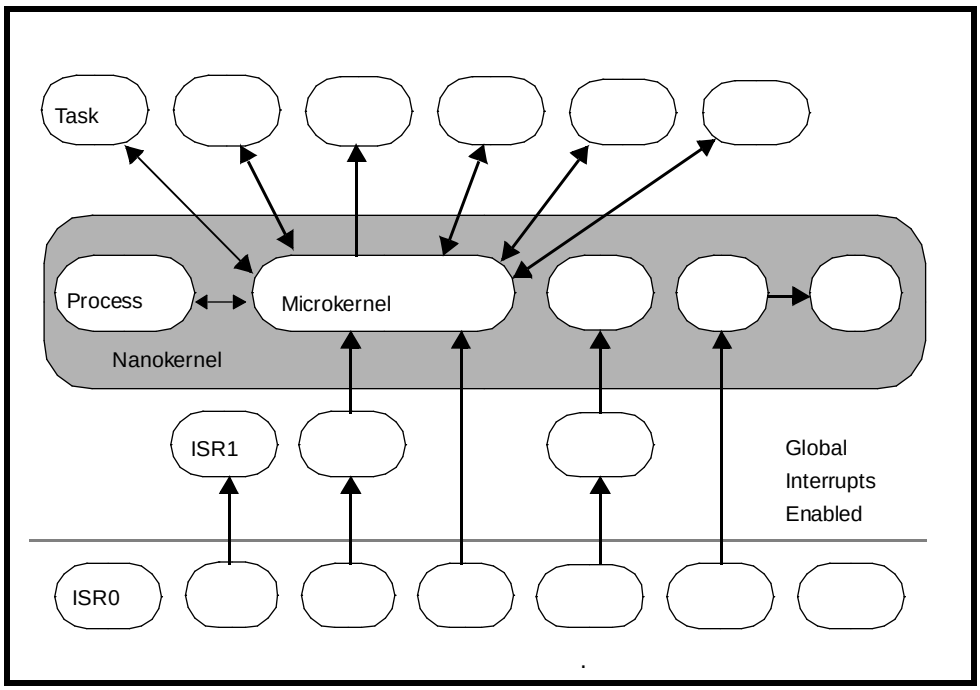


FIGURE 12 Interrupt servicing :an open multi-level approach.

4.9. Levels supported by the Virtuoso products.

In order to provide an overview, following table summarizes what default levels are supported in each Virtuoso product.

	Nano	Micro	Classico	
main()	yes	yes	yes	
microkernel	no	yes	yes	
nanokernel	yes	no	yes	
ISR0	yes	yes	yes	
ISR1	yes	no	yes	Multiple priorities when supported by hardware.

4.10. Support for parallel processing

Virtuoso comes in three implementations. The first one (/SP) has only support for single processor system. The second one (/MP) adds multiprocessor capabilities by providing low level drivers that permit to communicate between directly connected processors. The third implementation (/VSP) enables the user to access the defined objects and their services in a way totally independent of the location of objects in the network. That is, the kernel will automatically assure that the kernel service is executed even if the cooperating objects are located on different nodes. As a result source code of applications developed on the single processor version can be reused without any modification.

4.11. Target Environment

The standard support package of Virtuoso is designed to mostly operate in an embedded or semi-embedded processor environment. Over the life of the system, the application will be mostly static. The processor may be a single board microcomputer, a personal computer, a minicomputer or a large multiprocessor system. No assumptions are made about the configuration of the target system. It is the responsibility of the user to define the target environment and to insure that all necessary devices have program support. User interfaces can be built with the I/O and graphics library that has to be used with a server program on a host computer (e.g. a PC or UNIX workstation).

4.12. Virtuoso auxiliary development tools

Before programming the application, the first thing done after a functional analysis, is to translate the results of the analysis into the terms understood by the system generation utility of Virtuoso. The result is the sysdef text file in which the structure of the application (tasks, semaphores, topology, etc.)

is specified using a simple syntax. This file acts as a specification tool and as a maintenance tool. It is part of the system because in order to change an attribute of an object (e.g. the processor location or the stack size of a task), one changes the file and invokes the system generation utility Sysgen on it to regenerate all the necessary include files. The only thing left to do is to write the actual code of the task's function using the microkernel services.

The debugger is a task level debugger. When invoked it suspends all the tasks in the system (on all processors) and permits the user to jump to any processor in the system to examine the current state of all the defined objects.

The tracing monitor is integrated with the debugger and permits to inspect a system trace, in which the last 256 scheduling events were recorded.

Finally, Virtuoso intends to be more than a simple bare bones real time kernel. Functions are provided that give access to system resources, normally found on a host system. We strongly believe in this approach as it relieves the target environment from the burden of the development utilities and hence provides for a optimal use of the system resources.

The graphical server is an extension of the host server program that permits to perform some elementary graphics on the host screen from any task in the network. In general a standard I/O server (console I/O and simple file I/O with the host file system) is always supplied.

Note that there also exists a complementary product that contains a general purpose as well as specialized library of mathematical and signal processing functions.

4.13. Single processor operation

In practice, the microkernel is like a part of the runtime library of the compiler with the difference that we are really talking about tasks and not just functions. The user must consider the microkernel as an activity with a higher priority than any of his application tasks. The microkernel is therefore written to act upon events as fast as possible, minimizing the time that is spent in the microkernel. As such whenever anything happens in the system that awakes the microkernel, the task status can change. The result is that at this moment the microkernel will determine whether the current task is still runable or whether any other task with a higher priority has become runable. If so, the microkernel will switch the context to this task. If no other processors are involved in a microkernel service, the operation is as follows. When calling a microkernel service, the application task actually calls a microkernel interface function. In this interface, the parameters of the required service are copied to a parameter area and the microkernel is awakened by an inter-

rupt. The microkernel will then examine the call and execute the required actions. Eventually, a task swap can result. The microkernel can also be awakened by other sources, such as a timed event, an interrupt generated by external hardware or by data arriving on the communication ports. These events can even be simultaneous. In the latter case the interrupts will result in a command packet being put on the microkernel command queue. The microkernel handles all the commands one by one until the command queue is empty. It eventually swaps the context and returns the CPU to the highest priority task that is runable.

4.14. Virtual Single Processor operation

Virtuoso provides the user with the same API (Application Programmers Interface) whether used on a single or on a multiple processor system. For the user the differences are minor, as the implementation of Virtuoso has made it possible to provide a virtual single processor.

The main thing to remember is that once a single processor Virtuoso program has been (properly) developed, the programmer will be able to keep his source program mostly intact, when processors are added or when tasks or other microkernel objects are reallocated to remotely placed processors. Some reasons for using Virtuoso can be :

1. The need for more processing power or;
2. The need to place some processing power physically close to a monitored or controlled unit.

Thanks to communication links like those on the transputer or on the TMS320C40, this is done relatively easy. With Virtuoso, one can improve the real-time characteristics while keeping the original single processor source code intact ! Scalable processing power with scalable real-time software, that's what Virtuoso delivers.

The virtual processor model was obtained by using a system wide unique naming scheme. All microkernel related objects are defined system wide, including the tasks and their priorities, semaphores, queues, mailboxes and resources. Hence, for the programmer, it is as if he is using a processor cluster as a single real-time processing unit. The tricky thing is that the Virtuoso microkernel does more than correctly scheduling the tasks as is done in the single processor version.

Its operation is as follows :

First, for each microkernel service, the microkernel verifies whether the requested resource or service is available locally or on some other processor. If it is locally available, it is handled the same way as a single processor

kernel would do. If it is not locally available, the microkernel service request is transformed into a remote command packet. This command packet is composed of a header and a body. The header contains the type of microkernel service and its parameters, while the body contains the actual data to be transferred, if any. As such, some microkernel commands contain no body at all, as they simply transfer a microkernel service from one processor to another (example : the KS_Start() call).

To achieve this kind of topology transparency, the microkernel has been extended with an embedded router that will locate the processor where the service or resource is located. The router will then select the communication port to reach the desired processor. The router itself manages a pool of message buffers. These buffers are dynamically allocated. In addition, the microkernel messages indicate the priority of the requesting task. As such, the outgoing buffers are rearranged so that older messages with a lower priority will never block the more recent ones with a higher priority.

This can in practice be guaranteed because the granularity (and hence the delay involved) of a communication is about the same as the invocation of a microkernel service. Because communication is considered as a microkernel activity, it is very unlikely that the routing will hold up any task. In practice, it is so that Virtuoso can almost be considered as a system that implements some kind of virtual common memory. This was achieved by implementing a router that handles all data transport, including remote memory operations. This way, memory copying and message routing are considered as the same type of operation. The result is speed as these operations shortcut the layered architecture. The memory operations are the only active operations by the router that involve part of microkernel services. All other messages are passed on to the communication port drivers (e.g. links) or to the different kernel-agents.

This mechanism of ordering all requests in order of priority is used at all times and at all levels in the microkernel and guarantees that all higher priority related actions will precede lower priority related actions, even if they happen later in time. A single exception is that once a microkernel service is being served in the microkernel, it will continue to run until the microkernel action has finished. Note that meanwhile interrupts will still be accepted in the background.

The different microkernel activities are event driven, avoiding any form of active waiting or polling so that a suspended microkernel action will never block the system.

4.15. Heterogeneous processor systems

The virtual single processor model is so powerful that Virtuoso even runs on

systems with mixed types of processors. The key point is that the underlaying mechanism used by the microkernel is message based. Therefore, to transfer a command packet or data from one processor to another, the only requirement is the presence of a (fast enough) communication port. This can be common memory (like in VME systems), a serial link or a dual port RAM interface. A typical example could be a system with several processors depending on the function they have to fulfill. For example, control I/O functions typically require less powerful processors while compute intensive calculations can best be run on high speed processors like a DSP. Using Virtuoso gives a uniform interface to the programmer, while he can modify the target hardware without affecting much the source of his program.

5. Simple Examples

This section gives a simple example of programming with the Virtuoso microkernel. It illustrates the steps needed to define, build and run an application with a single task: the famous “hello, world” program. Then a more complex example with two tasks communicating via a queue is explained.

5.1. Hello, world

This example illustrates a simple program that uses the `stdio` library to print a string on the screen. Rather than start from scratch, it is easier to copy various files from one of the example directories and modify them. The files you need are normally as follows:

`sysdef`	The system definition file.
`makefile`	Build definition for the make utility. The use of make is recommended, even in the simplest applications, because of the dependencies that arrise from the system definition file.
`main1.c`	Standard main entry-point code, which is provided in source form for flexibility, although for normal applications it does not need to be changed.

The `sysdef` file is then edited to the following form:

```
NODE NODE1 C40

DRIVER NODE1 ‘HostLinkDma (2, 2, PRIO_DMA)’
DRIVER NODE1 ‘Timer0_Driver (tickunit)’

/*   taskname   node  prio entry    stack   groups */
/* ------------------------------------------------------------ */

TASK STDIODRV NODE1 3    stdiodrv  256    [EXE]
TASK HELLO    NODE1 10   hello     400    [EXE]

/*    queue node  depth width */
/* ---------------------------------------- */
QUEUE STDIQ NODE1 64    4
QUEUE STDOQ NODE1 64    4
```

```
/*       resource  node */
/* ------------------------ */
RESOURCE HOSTRES   NODE1
RESOURCE STDIORES  NODE1
```

The task STDIODRV, the queues STDIQ and STDOQ and the resource HOSTRES and STDIORES are used in the run-time system to coordinate accesses to the host. The only application task defined here is HELLO.

The body of the application task is as follows:

```
#include <_stdio.h>

void hello()
{
          printf("hello, world\n");
}
```

Note that the stdio include file has a leading underscore, to differentiate it from the include files that are often provided with compilers, so you can be sure that the Virtuoso version of the file is used. Then a function called hello() is defined. The name of the function is the same as the entrypoint specified in the system definition file. The use of the stdio function printf() is absolutely standard.

Next, modify the makefile. The code for the application tasks is compiled into a library, controlled by the variable

```
TASKS = hello.obj
```

Assuming that the above file is called hello.c, default rules in the makefile will compile and link the code automatically when make is invoked. To rebuild everything, make will start by pre-processing the sysdef file, and running sysgen. This will generate a C source file, node1.c and two headers, node1.h and allnodes.h. The node1.c file contains configuration code in C for one node, and the header files are to be included into the users code in order to access kernel objects. None of these files should be edited by hand.

Once the application is built, it may be run using the net loader and host server, as follows:

```
boardhost -rls hello
```

The name of the server is target-specific, as is the file, in this case hello.nli, which specifies the mechanism of loading the target hardware. However, when the program runs, the expected greeting should appear on the screen, whatever system is used.

As it is designed for real-time applications, which normally run continuously, there is no facility to terminate a Virtuoso program, so hit CTRL-C or CTRL-BREAK, and press X to get back to the operating system prompt.

5.2. Use of a Queue

The above example does not show how tasks interact using the microkernel objects. In this example a sender task sends a message to a receiving task via a queue. To make things more interesting, the two tasks are placed on different processors. The `sysdef` file is as follows:

```
NODE NODE1 C40
NODE NODE2 C40

NETLINK NODE1 'NetLinkDma (4,PRIO_DMA)' , NODE3 'NetLinkDma
         (0,PRIO_DMA)'

DRIVER  NODE1 'HostLinkDma (2, 2, PRIO_DMA)'
DRIVER  NODE1 'Timer0_Driver (tickunit)'
DRIVER  NODE2 'Timer0_Driver (tickunit)'

/*   taskname   node  prio entry    stack   groups */
/* ------------------------------------------------------- */

TASK STDIODRV NODE1 3    stdiodrv  256    [EXE]

TASK SENDER   NODE1 10   sender   400    [EXE]
TASK RECEIVER NODE2 10   receiver 400    [EXE]

/*    queue node  depth width */
/* ---------------------------------------- */
QUEUE STDIQ NODE1 64    4
QUEUE STDOQ NODE1 64    4

QUEUE DEMOQ NODE1 10    4

/*       resource  node */
/* ----------------------- */
RESOURCE HOSTRES   NODE1
RESOURCE STDIORES  NODE1
```

Here, the second node is defined at the begining of the file, and the netlink driver is declared to set-up the connection between the two processors. The two tasks, `SENDER` and `RECEIVER` are declared in the same way as the hello task of the previous example, and the queue through which they are to communicate, `DEMOQ`, is also declared.

The sender task is implemented as follows:

```
#include <iface.h>
#include <_stdio.h>
#include "allnodes.h"

void sender()
{
    int data = 42;

    printf("Sending %d to queue\n", data);
    KS_EnqueueW(DEMOQ, &data, 4);
}
```

And the receiver is implemented as follows:

```
#include <iface.h>
#include <_stdio.h>
#include "allnodes.h"

void receiver()
{
    int data;

    KS_DequeueW(DEMOQ, &data, 4);
    printf("Received %d from queue\n", data);
}
```

Note that the queue is identified in the calls to the kernel services by use of a C symbol with the same name as was used in the `sysdef` file. This is declared in the header file `allnodes.h`. These microkernel object identifiers can be accessed from code running on any node, due to the Virtual Single Processor implementation. In this example, it does not matter on which processor the queue is placed - both tasks can access it as if it were on the same processor.

In the /SP or /MP versions of Virtuoso, the two tasks and the queue would all have to be placed on the same processor.

The two tasks should be stored in separate files, so that when the code is generated for the two nodes, each node is only linked with the functions it needs, wth no dead code.

When the program is run, the following output should be seen:

```
Sending 42 to queue
Received 42 from queue
```

6. Applications

Virtuoso makes it much easier to use multiple processors to solve a single problem. The result is the feasibility of a new range of applications that were not possible before or not cost efficient.

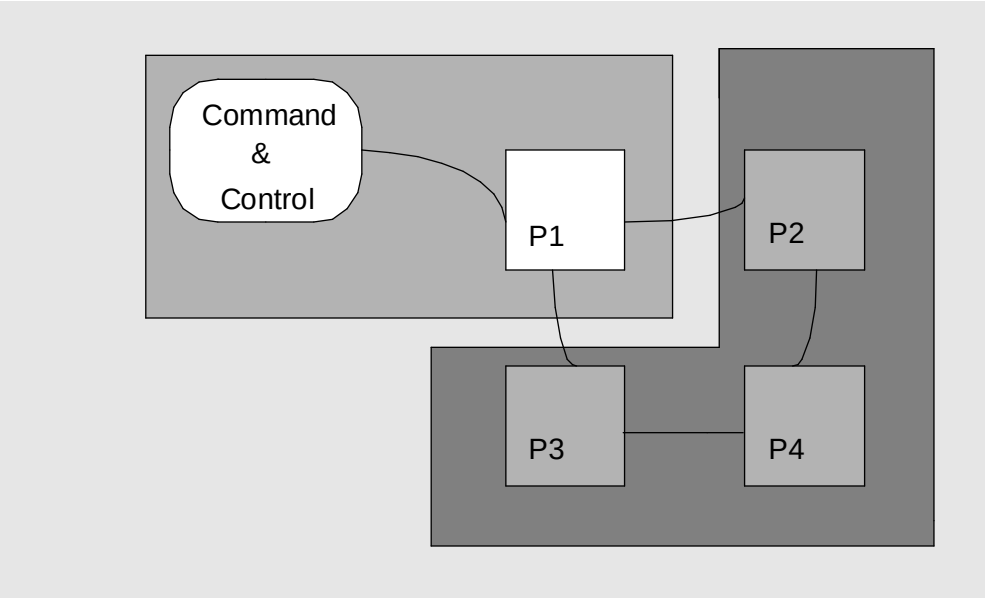


FIGURE 13 With Virtuoso, processors can easily be added or moved.

6.1. Scalable embedded systems

Traditional embedded applications are based on a single processors or microcontrollers. When the application requires more processing power the designer has two different options :

1. Using a more powerful processor
2. Adding processors

The same problem arises when he needs to physically redistribute the processing power for technical reasons. In both cases, with traditional processors, this will require a hardware redesign and probably a lot of programming work because the underlying hardware has changed.

The third option is to use from the start on hardware that has interprocessor capabilities, be it built in on the chip or with communication hardware at the board level (transputers, C40, VME,...) The result is that when using Virtuoso

from the start on, adding additional processing power is a matter of inserting a board, while the application only needs to be reconfigured. Sysgen then generates the new include files. The only thing left to do is to recompile and link the whole program.

6.2. Complex control systems

Because Virtuoso provides the user with a virtual single processor model it is now much easier to build distributed control systems, because it is no longer important where a particular microkernel object is located in the system. The programmer will see no logical difference between processing nodes located in the same enclosure and processing nodes which are located on remote sites and interconnected using optical fibers over several km.

6.3. Simulation in the control loop

This is a promising area. Until now, most complex systems that need to be controlled (electricity power plants, chemical plants) rely on an experienced operators. But even with years of experience they can make mistakes, especially in exceptional conditions. Often it has been observed afterwards that the operator error could have been prevented if the operator would have got complete know-how of the behavior of the system in these exceptional condition (e.g. a reactor failure). The same applies for a change in the desired production level of a given product (e.g. cracking installation). The solution is to use simulators that work in parallel with the control system. This is possible today but it requires a supercomputer to do it. With a parallel machine this can be done in a cost efficient way, while Virtuoso provides the necessary real-time characteristics.

6.4. Fault tolerant systems

Because Virtuoso provides complete transparency as to the location of objects in the system, it is fairly straightforward to write application dependent programs that have a degree of fault tolerance. The solution is simply to provide passive (or active) backup copies of the active tasks on other processors. As a result we have then hardware and software redundancy. On each processor, monitoring tasks check the operational status of the processor. When a failure is detected, the tasks on the erroneous processor are considered as terminated while the backup tasks are made executable. As the location of queues, semaphores, mailboxes and resources is not visible to the application task, it is fairly straightforward to write them in such a way that the system continues only with a minor delay (the time needed to detect the failure and to change the tasks' execution state).

6.5. Communication systems

Another use of Virtuoso is to exploit the powerful routing capability, without being concerned too much about the real-time facilities. The point is that Virtuoso is a message based system. Hence it is quite forward to construct a system where each node is running the Virtuoso protocol even if different processor types are in use and even if different types of transmission media are being used. On each node, one has to port the microkernel (fairly easy because of the use of ANSI C) while to accommodate the different transmission media, one only has to implement a communication port driver. The current version of Virtuoso could in principle accommodate 64K nodes with each 64K tasks. In practice generating this kind of system will involve additional work to change the system generation utility to work more efficiently and to write specific loaders. For smaller systems, using homogenous types of processors and communication media, the current solution is more than adequate.

Virtuoso ™

The Virtual Single Processor Programming System

User Manual

Covers :

Virtuoso Classico ™

Virtuoso Micro ™

Version 3.11

PART 2: Reference Manual

Creation date : February 5, 1990

Modification date : September 6, 1996

7. Virtuoso microkernel types & data structures

7.1. Microkernel types

In order to understand how Virtuoso works and how to build real-time application systems around it, it is useful although not necessary to understand how its various control and data structures work. This section describes these structures and their relationships. The descriptions will include:

Object Type	Related C datatypes
Tasks	K_TASK, K_TGROUP, K_PRIO
Semaphores	K_SEMA
Mailboxes	K_MBOX
Queues	K_QUEUE
Memory Maps	K_MAP
Resources	K_RES
Timers	K_TICKS

The second field in the list above is the predefined corresponding data type. The Virtuoso microkernel services which deal with these classes of control and data structures will be presented in a subsequent section. See also the k_struct.h and k_types.h file for the full details.

7.2. Tasks

In a real-time application, the functions of the system are assigned to various tasks. Virtuoso supports as many tasks as the user wants in a single system.* The nature of each task is, of course, application dependent and left to the imagination of the system designer. However, there are attributes

*. The number of tasks is in practice only dependent on the word size of the processor used.

that all tasks share. These include:

1. a task identifier
2. a priority level
3. a task group set
4. a task state
5. an entry point
6. a task abort handler
7. a task stack pointer and stack size
8. a task context

7.2.1. Task Identifier & Priority

Each task is identified by a numerical identifier which is a number combined with a second field referring to the processor it is located on. The task identifier serves as a reference during microkernel operations. Virtuoso uses a fixed task numbering scheme, where the numbers are generated at compile time, because it is low in overhead and quite adequate for most applications.

Besides an identifier, each task also has a priority initially defined at system generation time. On each processor the microkernel schedules the local executable tasks in order of priority. While the priority is fixed at start time, it can be altered by using the KS_SetPrio() microkernel service or dynamically by the microkernel.

7.2.2. Task group set

Each task can be part of a number of task groups.The task group set is a 32 bit word of which each bit represents a task group. The existence of task groups and the related microkernel services permit task operations in a single atomic action.

7.2.3. Task State

Whenever the Virtuoso microkernel is looking for a task to execute, it examines the execution state variable to see if the task is runable or not. The execution state is contained in a single word and a value of 0 (zero) indicates that the task is runable. Whenever the task state is different from zero, this indicates that the task is suspended, waiting or aborted.

A task normally becomes runable after it has been started. Once runable, the task will become active if it has the highest priority of all runable tasks. A task becomes not runable if it is suspended or if it issues a microkernel

service that cannot be serviced immediately (hence waiting). When suspended, a task can only become runable again by a KS_Resume() service call. Normally, a task only terminates when it reaches the end of its execution thread. Tasks can be aborted independently of their current point of execution by a KS_Abort() service call. Once aborted, the task can only be restarted by a KS_Start() service call.

7.2.4. Task Entry Point

The entry point is the address where the task is to begin execution. It corresponds with the address of the C function that implements that task. The entry point can be changed at runtime to provide a kind of dynamic tasking.

7.2.5. Task Abort Handler

The task abort handler is an alternative entry point that can be installed at runtime. This permits to execute asynchronously (but only once) application dependent actions when the task has been aborted.

7.2.6. Task Stack

Each task must have a stack for its local workspace. The size of each task's stack is dependent on many things such as the maximum depth of nested function calls and the maximum amount of working space needed for temporary variables.

7.2.7. Task Context

The context of a task consists of the information needed to resume execution of the task after it has been descheduled. On most processors, this information consists of the values held in a subset of the processor's registers at the moment the task was descheduled.

7.3. Semaphores

There are several forms that a semaphore may take. Virtuoso uses counting semaphores. The semaphore starts with a count of zero at the start of the program. Whenever the semaphore is signalled using the KS_Signal() call, the count is incremented by one and the signalling task eventually continues execution. Whenever a task waits on a semaphore to be signalled (using the KS_Wait() call), two possible situations can happen. When the semaphore count is greater than zero, the count is decremented by one and the waiting task continues its execution. Otherwise, the task is put into the semaphore

waiting list in order of its priority. When the semaphore is then signalled, the task with the highest priority is removed from the waiting list. This algorithm requires that the semantic meaning given to the event associated by the semaphore is totally independent of the order in which tasks signal the semaphore or wait on it. Virtuoso permits to signal a list of semaphores in a single operation while a task can wait for any semaphore in a list of semaphores to be signalled. If more than one is signalled, the first signalled on the list is taken.

Semaphores are typically used to synchronize tasks to the occurrence of a given event. One task may need to wait for the other to reach a certain point before it can continue. Input/output operations are an example of this type of synchronization. For instance, when an input operation is desired, the task waits on the input to complete (the event) by associating the event with a particular semaphore and suspending further processing until the the semaphore is signalled. When the input operation is completed, the device driver signals the semaphore associated with the event to indicate that the data is available. This signalling causes the waiting task to resume, presumably to process the input data.

7.4. Mailboxes

Mailboxes are the means by which data can be transmitted synchronously from a sender to a receiver task. The actual message acts as a request for a data transfer between a sender and a receiving task.

The data referenced by the message may contain whatever is required by the receiver task and in whatever format. A mailbox acts as a ‘meeting place’ for tasks wishing to exchange a message or the data referenced by it. It maintains two waiting lists : one for senders, and one for receivers. When a new send or receive request arrives, the mailbox searches one of the lists for a corresponding request of the other type. If a match is found, it is removed from the waiting list, and the data transfer is started. When this has finished both tasks are allowed to resume execution. If no match can be found, the caller is suspended and put on a waiting list, or the operation fails.

The message is implemented using a datastructure that contains following elements :

1. The size of the referenced data;
2. the pointer to the data at sender’s side
3. the pointer to the data at receiver’s side
4. the sending task
5. the receiving task
6. the info field (optional).

The sending task will fill in following elements when issuing a KS_Send() call:

1. the data size;
2. the pointer to the data (possibly undefined if message size is zero)
3. the requested receiver task (the predefined ANYTASK, if any task will do)
4. the info field (optional).

The receiver task will fill in the following elements and issue a KS_Receive() call:

1. the requested sender (the predefined ANYTASK, if any task will do);
2. the data size;
3. the starting address to which the data must be copied;
4. the info field (optional).

The mailbox will then try to match the sender and receiver messages and copy the relevant sender message fields into the message structure of the receiver if a match is found and vice versa. The message data will automatically be copied starting at the specified address. The copy operation will be limited by the smallest given size should the sizes not match. The receiver can also inhibit this automatic data copying by filling in a NULL pointer as the starting address. This way the receiver can inspect the message and determine which action to take. The copying of the data is then started by invoking the KS_ReceiveData() service after filling in the starting address and the size. Note that the info field can be used to transmit a one word message. If no data copying is necessary and the receiver has used a NULL pointer, he still has to reschedule the sender by issuing the KS_ReceiveData() service call with a zero size filled in. This call then acts as an ACK for the sender.

The C syntax struct of the message header is as follows :

```
typedef struct
{
INT32 size;       /* size of message (bytes) */
void *tx_data;    /* pointer to data at sender side */
void *rx_data;    /* pointer to data at receiver side */
K_TASK tx_task;   /* sending task */
K_TASK rx_task;   /* receiving task */
INT32 info;       /* information field, free for user */
} K_MSG;
```

Note that it is also possible to copy data directly using the KS_MoveData() service. However this service requires that correct pointers are provided using other techniques before. Because this technique permits to perform

this synchronization operation only once, it provides for much better performance.

7.5. Queues

Virtuoso supports first-in-first-out (FIFO) queues having single or multiple bytes per entry. The queues support both a single producer and a single consumer as well as multiple producer/consumer services. Queues are addressed with an identifier that is unique to each queue. Queue identifiers are assigned in the system generation procedure. Virtuoso queues are different from messages because queue entries act as buffers for the actual data. Another difference is that the queue entries represent the chronological order of processing. Also the priorities of the sender or receiver are not considered. Queues are usually used to handle data such as character streams input/output or other data buffering.

A queue is defined by two elements during the system generation process. These are :

1. the queue size, giving the number of entries;
2. the queue entry size, giving the size of each entry.

In the current implementation, the entry size is limited to 24 bytes when using the multiprocessor version of Virtuoso. There is no limitation when the queue is local.

7.6. Resources

In any system which uses the concepts of multitasking, there is the inevitable problem of two or more tasks competing for a single (physical) resource. A resource might be an external device, a block of memory or a data structure. These resources, as well as others like them, usually require that no other task gains access to them during a critical period when a task is operating on that resource. To do so might involve the corruption of a data structure or garbling of the output. The case is more serious for write access to a resource than for read access but the implications of either case cannot be determined. The solution is to protect the physical resource while it is being used by one task so that other tasks cannot gain either read or write access.

While the solution is obvious, the problem is how to grant protection in any possible configuration of Virtuoso. Thus the construct called a logical resource, or just simply resource. In Virtuoso, a resource can be anything: memory, a device, a non-reentrant code section, or whatever. A typical resource is the server interface to a host. All that is necessary for any such resource needing protection is to be identified and given a name. It is the responsibility of the programmer to provide such identification. The

programmer must insure that any use of the resource be preceded by a Virtuoso microkernel service that locks out other users. Correspondingly, when the task has finished using the resource, the Virtuoso function to unlock the resource must follow. With these two simple functions, KS_Lock() and KS_Unlock(), Virtuoso can protect a resource and at the same time arbitrate which task is to get subsequent control when the critical period has passed. Note that it is still the programmers responsibility that the resource is correctly used. As such the logical resource is merely a flag. If bypassed, you will get cases in which your physical resource is no longer protected !

When a task locks an idle (not being used) resource, Virtuoso sets the current owner to the requesting task. Another task trying to lock the same resource will then find the resource already in use. Virtuoso places the second task into a resource wait list, and the task is inserted into the wait list of the resource in order of its priority. When the resource is unlocked by its current owner, Virtuoso will look for the next waiter in the list. If there is one, it will become owner of the resource.

In practice, it is possible that situations arise in which resources are locked by lower priority tasks while a higher priority task that wants to lock on it is kept in the waiting list. Tasks of intermediate priority can then get hold of the CPU and preempt the lower priority task. The result is that the higher priority task can be kept for quite a long time from running even while the resource it is waiting for is not used. The solution is then to raise temporarily the priority of the lower priority task. This algorithm is called the priority inheritance algorithm. A variant of this algorithm is planned to be available in a later version of Virtuoso.In the current version, the programmer can achieve the same result by raising the priority of the using task prior to the resource lock. He must not forget to reset the priority afterwards.

7.7. Timers

In the descriptions of time based functions to follow, it will be important to understand the conventions used by Virtuoso. All timers in Virtuoso are either 16-bit or 32-bit values depending on the processor size. The two time units used internally by Virtuoso are ticks (mostly maintained by a low resolution clock driver) and the value of the high resolution timer hardware, if any present. A tick gives the amount of time between clock generated system interrupts, or equivalently, the period between clock interrupt service requests.

The tick value is expressed as a number of high precision clock cycles in the mainx.c file.

7.8. Memory maps

The free RAM in a Virtuoso system may be divided into uniquely identifiable maps. A map is addressed via an identifier which is assigned to it during the system generation procedure. Each map may be subdivided into any number of blocks of the same size. Thus, a request for memory from a specific map results in the return of the address of one of the blocks in the map. It is of no consequence which block is referenced since all blocks within a map are of equal size. A map consists of a set of unused blocks, i.e., free memory, and a set of used blocks.

A map is the structure which Virtuoso uses to manage the memory partition.

8. Virtuoso microkernel services

8.1. Short overview

This Section will describe the complete set of the Virtuoso microkernel services. The microkernel services are divided into nine classes listed below :

1. Task control
2. Semaphores
3. Messages & mailboxes
4. Queues
5. Timer management
6. Resources
7. Memory management
8. Processor specific
9. Special

Most of the services exist as three variant types, depending on what action must be taken when the service cannot be provided immediately. The first type of the service returns with an error code when the service is not available. The second type will wait until the service is available. As this can be forever, the third type of the service permits to limit the waiting period to a certain time-interval, expressed in timer ticks.

The suffixes used to distinguish the three variant types are as follows :

1. No suffix : no waiting (returns an error when not serviced);
2. -W : wait till service available;
3. -WT : wait till service available or timeout expires.

Two other suffixes are used as well :

1. -M : to indicate that the service operates on a list
2. -G : to indicate that the service operates on a set of taskgroups.

8.2. Important note

While Virtuoso is ported as much as possible with the same API to all processors, it is possible that particular processors have a slightly different API, especially if the service is processor specific. Always verify the correct syntax in the iface.h include file if you encounter errors when compiling. The source is always the ultimate arbiter.

8.3. Task control microkernel services

- start a task from its entry point
```
void KS_Start(K_TASK task);
```

- start a set of taskgroups
```
void KS_StartG(K_TGROUP taskgroup);
```

- suspend a task
```
void KS_Suspend(K_TASK task);
```

- suspend a set of taskgroups
```
void KS_SuspendG(K_TGROUP taskgroup);
```

- resume a task
```
void KS_Resume(K_TASK task);
```

- resume a set of taskgroups
```
void KS_ResumeG(K_TGROUP taskgroup);
```

- abort a task
```
void KS_Abort(K_TASK task);
```

- abort a set of taskgroups
```
void KS_AbortG(K_TGROUP taskgroup);
```

- install an abort handler function
```
void KS_Aborted(void (*function)(void));
```

- delay a task
```
void KS_Sleep(K_TICKS ticks);
```

- yield processor to another task
```
void KS_Yield(void)
```

- set timeslicing period
```
void KS_SetSlice(K_TICKS ticks,
                 K_PRIO prio);
```


- change a task's priority
```
void KS_SetPrio(K_TASK task,
                int priority);
```

- change a task's entry point
```
void KS_SetEntry(K_TASK task,
                 void (*function)(void));
```

- add a task to a set of taskgroups
```
void KS_JoinG(K_TGROUP taskgroup)
```

- remove a task from a set of taskgroups
```
void KS_LeaveG(K_TGROUP taskgroup)
```

- get current task identifier
```
K_TASK KS_TaskId;
```

- get current taskgroup mask
```
K_TGROUP KS_GroupId(void);
```

- get current task's priority
```
K_PRIO KS_TaskPrio;
```

- get current task's node identifier
```
K_NODE KS_NodeId;
```

8.4. Semaphore microkernel services

A complete set of microkernel services for using counting semaphores is provided by Virtuoso.

- signal semaphore
```
void KS_Signal(K_SEMA sema);
```

- signal multiple semaphores
```
void KS_SignalM(K_SEMA *semalist);
```

- test if a semaphore was signalled
```
int KS_Test(K_SEMA sema);
```

- wait for a semaphore to be signalled

```
int KS_TestW(K_SEMA sema);
int KS_Wait(K_SEMA sema);
```

- wait for a semaphore with timeout

```
int KS_TestWT(K_SEMA sema,
              K_TICKS ticks);
int KS_WaitT(K_SEMA sema,
             K_TICKS ticks);
```

- wait for one of many semaphores

```
K_SEMA KS_TestMW(K_SEMA *semalist)
K_SEMA KS_WaitM(K_SEMA *semalist)
```

- test for one of many semaphores and wait with timeout

```
K_SEMA KS_TestMWT(K_SEMA *semalist,
                  K_TICKS ticks);
K_SEMA KS_WaitMT(K_SEMA *semalist,
                 K_TICKS ticks);
```

- return the current semaphore count

```
int KS_InqSema(K_SEMA sema);
```

- reset a semaphore count to zero

```
void KS_ResetSema(K_SEMA sema);
```

- reset multiple semaphores

```
void KS_ResetSemaM(K_SEMA *semalist);
```

8.5. Mailbox microkernel services

- insert message into a mailbox

```
int KS_Send(K_MBOX mailbox,
            K_PRIO prio,
            K_MSG msgstruc);
```

- insert message into mailbox and wait till done

```
int KS_SendW(K_MBOX mailbox,
             K_PRIO prio,
             K_MSG *msgstruc);
```


- insert message into mailbox and wait with timeout

```
int KS_SendWT(K_MBOX mailbox,
              K_PRIO prio,
              K_MSG *msgstruc,
              K_TICKS ticks);
```

- receive message from mailbox if any

```
int KS_Receive(K_MBOX mailbox,
               K_MSG *msgstruc);
```

- receive message from mailbox and wait till done

```
int KS_ReceiveW(K_MBOX mailbox,
                K_MSG *msgstruc);
```

- receive message and wait with timeout

```
int KS_ReceiveWT(K_MBOX mailbox,
                 K_MSG *msgstruc,
                 K_TICKS ticks);
```

- retrieve message data

```
void KS_ReceiveData(K_MSG *msgstruc);
```

- copy data for size from a source address on a source node to a destination address on a destination node

```
void KS_MoveData(int node,
                 int size,
                 void *source,
                 void *destination,
                 int direction);
```

8.6. Queue microkernel services

- put an entry into a queue

```
int KS_Enqueue(K_QUEUE queue,
               void *source,
               int size);
```

- put an entry into a queue and wait till done

```
int KS_EnqueueW(K_QUEUE queue,
                void *source,
                int size);
```

- put an entry into a queue and wait with timeout

```
int KS_EnqueueWT(K_QUEUE queue,
                 void *source,
                 int size,
                 K_TICKS ticks);
```

- get an entry from a queue

```
int KS_Dequeue(K_QUEUE queue,
               void *destination,
               int size);
```

- get an entry from a queue and wait till done

```
int KS_DequeueW(K_QUEUE queue,
                void *destination,
                int size);
```

- get an entry from a queue and wait with timeout

```
int KS_DequeueWT(K_QUEUE queue,
                 void *destination,
                 int size,
                 K_TICKS ticks);
```

- return the current number of queue entries

```
int KS_InqQueue(K_QUEUE queue);
```

- remove all current entries and clear the list of the tasks waiting to dequeue

```
voidKS_PurgeQueue(K_QUEUE queue);
```

8.7. Timer management microkernel services

These services are only available on processors where a timer exists, or a periodic interrupt can be supplied to the processor. If these functions are not available, then the timeout variants ow all waiting kernel functions do also not exist.

- allocate timer

```
K_TIMER *KS_AllocTimer(void);
```

- deallocate timer

```
void KS_DeallocTimer(K_TIMER *timer);
```

- Start timer with specified delay or cyclic period and signal semaphore at each period

```
void KS_StartTimer(K_TIMER *timer,
                   K_TICKS delay,
                   K_TICKS cyclic_period,
                   K_SEMA sema);
```

- reset and restart timer with new specified periods

```
void KS_RestartTimer(K_TIMER *timer,
                     K_TICKS delay,
                     K_TICKS cyclic_period);
```

- stop the timer

```
void KS_StopTimer(K_Timer *timer);
```

- compute elapsed time

```
K_TICKS KS_Elapse(K_TICKS *stamp);
```

- return the low resolution time

```
K_TICKS KS_LowTimer(void);
```

8.8. Resource management microkernel services

- request resource and lock

```
int KS_Lock(K_RES resource);
```

- request resource and lock and wait till granted

```
int KS_LockW(K_RES resource);
```

- request resource and lock and wait with timeout

```
int KS_LockWT(K_RES resource,
              K_TICKS ticks);
```

- release resource

```
void KS_Unlock(K_RES resource);
```

8.9. Memory management microkernel services

- allocate block from a given map

```
void *KS_Alloc(K_MAP map,
               void **block);
```

- allocate block from a given map

```
void *KS_AllocW(K_MAP map,
                void **block);
```

- allocate block from a given map and wait until available

```
void *KS_AllocWT(K_MAP map,
                 void **block,
                 K_TICKS ticks);
```

- return block to a map given set

```
void KS_Dealloc(K_MAP map,
                void **block);
```

- inquire on blocks in use in Map

```
int KS_InqMap(K_MAP map);
```

8.10. Special microkernel services

This is a class of directives which are included for special purposes.

- “do nothing” microkernel service (used for benchmarks)

```
void KS_Nop(void);
```

- function executed with preemption disabled

```
int KS_User(int (*function)(void *0),
            void *arg);
```

8.11. Drivers and processor specific services

This class of microkernel services is only available for some processors and their use is to enable user tasks to access the processor specific hardware such as communication links and interrupt pins. See iface.h for the processor specific function.

- start a read operation of a block of size bytes from a given link and return the associated event number

```
int KS_Linkin(int link,
              int size,
              void *destination);
```

- starts a write operation of a block of size bytes to a given link and return the associated event number

```
int KS_Linkout(int link,
               int size,
               void *source);
```

- read a block of size bytes from a given link and wait until done

```
void KS_LinkinW(int link,
                int size,
                void *destination);
```

- read a block of size bytes from a given link and wait until done or time-out expires, only for T800 and T9000

```
int KS_LinkinWT(int linkin,
                int size,
                void *destination,
                K_TICKS ticks);
```

- writes a block of size bytes to a given link and wait till done

```
void KS_LinkoutW(int link,
                 int size,
                 void *source);
```

- write a block of size bytes to a given link and wait until done or time-out expires, only on T800 and T9000

```
int KS_LinkoutWT(int linkout,
                 int size,
                 void *source,
                 K_TICKS ticks);
```

- enable an ISR

```
void EnableISR(int IRQ,
               void (*ISR)(void));
```

- disable interrupt service routine
  ```
  int DisableISR(int IRQ);
  ```

- suspend the calling task until an event occurrs
  ```
  void KS_EventW(int IRQ);
  ```

- read the current high precision clock
  ```
  int KS_HighTimer(void);
  ```

- return the current CPU workload
  ```
  int KS_Workload(void);
  ```

- set the measuring interval for the workload monitor
  ```
  void KS_SetWlper(K_TICKS period);
  ```


9. Nanokernel types and datastructures

9.1. Nanokernel processes and channels

The nanokernel unit of execution can be considered as a light task, that is a task with a light context as compared with the microkernel tasks. To avoid any confusion and in analogy with the internal transputer architecture, the following terminology is introduced :

1. 'Processes' for designating the nanokernel (light) tasks;
2. 'Channels' for designating the interprocess communication objects.

The nanokernel has been designed for minimal overhead when switching between processes. This has been achieved by:

1. A small number of registers that have to be saved over a context switch;
2. A minimum semantic overhead
3. Small size, often fitting in internal RAM.

The small number of registers means that process code must be written in assembly language. It is still possible to call C functions from within a process, if certain rules are observed.

The minimum semantic overhead results from the following design options :

1. No preemption, but interruptible by interrupt service routines;
2. Strict FIFO scheduling;
3. Only one waiting process allowed when synchronizing with another process;
4. No time-outs;
5. No distributed operation.

These restrictions are not important if the microkernel level is present and if nanokernel processes are used in an appropriate way. Overhead is reduced by a factor of 10 when compared to similar microkernel operations. In Virtuoso Nano, solely based on the nanokernel, some of these restrictions were lifted. Refer to the product manual. The rest of this chapter is specific for the nanokernel as used within Virtuoso Classico.

9.2. Nanokernel channels

Nanokernel processes can synchronize using three types of channels:

1. A counting semaphore channel;
2. A linked list channel;
3. A stack channel.

More details are available in part 3 for each processor type.

10. Nanokernel services

The nanokernel processes have a much simpler scheduling mechanism and set of services than the microkernel tasks. Nanokernel processes are never preempted by another nanokernel process (and hence are by definition critical sections). Nanokernel processes only deschedule voluntarily upon issuing a kernel service. They execute in pure FIFO order when executable. Note however that they can be interrupted by an ISR level routine but will themselves preempt any microkernel task when becoming executable. Hence, consider the nanokernel level as a set of high priority processes while the microkernel tasks have a low priority. Note that the microkernel itself is a nanokernel process.

Most nanokernel services are assembly routines. As parameters are passed using registers, no general syntax can be provided as it is processor dependent. As they start up and terminate in assembly, a good know-how of the target processor is required. In addition as registers are used to pass parameters, be very careful when programing at this level !

Refer to part 3 of the manual for the details. In Virtuoso Nano, a more complete nanokernel is used. Refer to its manual for details.

10.1. Process management

Note :

The actual syntax might be different depending on the target processor as they depend on the registers and instructions available. The descriptions below are therefore generic while the binding manual provides processor specific descriptions.

```
init_process (void *workspace,
      void entry(void),
      int param1,
      int param2,...);
      /* Sets up a nanokernel process */
      /* C function called from microkernel or main() level */
start_process (void *workspace);
      /* Starts up a nanokernel process */
      /* C function called from microkernel or main() level */
nanok_yield
      /* Yield CPU to another nanokernel process*/
```

10.2. ISR management

```
end_isr0
     /* Terminates a level 0 ISR */
end_isr1
     /* Terminates a level 1 ISR */
set_isr1
     /* Switch to ISR level 1 */
```

10.3. Semaphore based services

```
prhi_sig
     /* Signal and increment semaphore */
prhi_wait
     /* Wait on semaphore to be signalled */
```

10.4. Stack based services

```
prhi_psh
     /* Push data onto a stack */
prhi_popw
     /* Pop data from stack, wait if stack empty */
prhi_pop
     /* Same as above but no waiting */
```

10.5. Linked list based services

```
prhi_put
     /* Insert at head of linked list*/
prhi_getw
     /* Get element from linked list, wait if list is empty */
prhi_get
     /* Same as above but no waiting */
```


11. Alphabetical List of Virtuoso microkernel services

In the pages to follow, each Virtuoso microkernel service will be shown in alphabetical order. Each Virtuoso microkernel service will be given in a standard format:

- SUMMARY . . . . . Brief summary of the service.
- CLASS. . . . . One of the Virtuoso microkernel service classes of which it is a member.
- SYNOPSIS . . . . . The formal C declaration including argument typing.
- DESCRIPTION . . . A description of what the Virtuoso microkernel service does when invoked and how a desired behavior can be obtained.
- RETURN VALUE . . The return values of the Virtuoso microkernel service.
- EXAMPLE . . . . . One or more typical Virtuoso microkernel service uses. The examples assume the syntax of C, but error handling is ignored, except for the service under discussion.

  On some common processors characters are stored one per word or sizeof does not give the number of 8 bit bytes in a type. On these processors the examples may need modification where a size parameter or structure is used.
- SEE ALSO. . . . . List of related Virtuoso microkernel services that could be examined in conjunction with the current Virtuoso microkernel service.
- SPECIAL NOTES . . Specific notes and technical comments.

11.1. KS_Abort

• SUMMARY . . . . . Abort a task.

• CLASS . . . . . Task

• SYNOPSIS . . . . .

```
void KS_Abort(K_TASK task);
```

• DESCRIPTION . . . The KS_Abort microkernel service is used to abort a task's execution. If the task has no abort handler, it will terminate execution immediately; otherwise the abort handler function is executed, using the identity and priority level of the aborted task.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
K_TASK  WORKTASK;

KS_Abort(WORKTASK);    /* abort task WORKTASK    */
KS_Abort(KS_TaskId);   /* abort the current task */
```

• SEE ALSO. . . . . KS_AbortG
KS_Aborted

• SPECIAL NOTES . . Be very careful when using this service as the microkernel does not clean up the current state.

It is illegal to abort a task while it is blocked in a waiting kernel service. The task is not removed from the waiting list of the object. This means that KS_Abort should only be used on a task that is executing or ready to execute. A safe way to ensure this is to restrict the use of KS_Abort to aborting the current task.

The parameter to this task must specify a local task. Supplying a remote task will have unpredictable side effects.

11.2. KS_AbortG

• SUMMARY . . . . . Abort a task group.

• CLASS. . . . . Task

• SYNOPSIS . . . . .

```
void KS_AbortG(K_TGROUP taskgroup);
```

• DESCRIPTION . . . The KS_Abort microkernel service is used to abort a task group or a set of task groups in a single call. It is an atomic equivalent to calling KS_Abort for every task in the set defined by the argument. See KS_Abort for details.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
K_TGROUP   WORKERS;
K_TGROUP   SLAVES;

/*
 * abort all tasks in the WORKERS or SLAVES groups
 */
KS_AbortG(WORKERS | SLAVES);
```

• SEE ALSO. . . . . KS_Abort
KS_Aborted

• SPECIAL NOTES . . See the notes under KS_Abort.

11.3. KS_Aborted

• SUMMARY . . . . . Install abort handler function.

• CLASS . . . . . Task

• SYNOPSIS . . . . .

```
void KS_Aborted(void (*function)(void));
```

• DESCRIPTION . . . The KS_Aborted microkernel service installs an abort handler function for the calling task. When the calling task is aborted by a KS_Abort or KS_AbortG call, the abort handler function is executed using the task identity and priority level of the task that installed it. KS_Aborted can be called any number of times; each subsequent call overwrites the previously installed handler. KS_Aborted (NULL) removes any installed handler. The microkernel removes an abort handler after it has been invoked, or when a task terminates normally.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
K_RES     DISPLAY;

void DisplayAbort(void)
{
    /*
     * free the DISPLAY resource
     * safe if another task owns the display
     */
    KS_Unlock(DISPLAY);
}

void Display (void)
{
    KS_Aborted(DisplayAbort);
    KS_Lock(DISPLAY);
    ... do display actions
    KS_Unlock (DISPLAY);
}
```

• SEE ALSO. . . . . KS_Abort
KS_AbortG

11.4. KS_Alloc

• SUMMARY . . . . . Allocate a block of memory.

• CLASS. . . . . Memory

• SYNOPSIS . . . . .

```
int KS_Alloc(K_MAP   map,
             void  **block);
```

• DESCRIPTION . . . The KS_Alloc microkernel service is used to allocate a block of memory from a predefined memory map, without waiting.

• RETURN VALUE . . RC_OK if a block is available, RC_FAIL otherwise.

• EXAMPLE . . . . . .

```
typedef void *   MyBlock;
MyBlock          p;
K_MAP            MAP1K;
int              RetCode;

RetCode = KS_Alloc(MAP1K, &p);
if (RetCode != RC_OK) {
    printf("Out of memory\n");
}
```

• SEE ALSO. . . . . KS_AllocW
KS_AllocWT
KS_Dealloc
KS_InqMap

11.5. KS_AllocW

• SUMMARY . . . . . Allocate a block of memory with wait.

• CLASS . . . . . Memory management

• SYNOPSIS . . . . .

```
int KS_AllocW(K_MAP   map,
              void  **block);
```

• DESCRIPTION . . . The KS_AllocW microkernel service is used to allocate a block of memory from a predefined memory map. If the map is empty, the calling task is put on a priority-ordered waiting list and is descheduled until a block becomes available.

• RETURN VALUE . . RC_OK.

• EXAMPLE . . . . . .

```
typedef void *   MyBlock;
MyBlock          p;
K_MAP            MAP1K;
int              RetCode;

RetCode = KS_AllocW(MAP1K, &p);
if (RetCode != RC_OK) {
    printf("Cannot allocate memory\n");
}
```

• SEE ALSO. . . . . KS_Alloc
KS_AllocWT
KS_Dealloc
KS_DeallocW
KS_DeallocWT

11.6. KS_AllocWT

• SUMMARY . . . . . Allocate a block of memory with timed out wait.

• CLASS. . . . . Memory management

• SYNOPSIS . . . . .

```
int KS_AllocWT(K_MAP    map,
               void   **block,
               K_TICKS  ticks);
```

• DESCRIPTION . . . The KS_AllocWT microkernel service is used to allocate a block of memory from a predefined memory map. If the map is empty, the calling task is descheduled and put on a priority-ordered waiting list. If no block is available within the specified time out, the allocation fails but the task is allowed to proceed.

• RETURN VALUE . . RC_OK or RC_TIME if the call timed out.

• EXAMPLE . . . . . .

```
typedef void *   MyBlock;
MyBlock          p;
K_MAP            MAP1K;
int              RetCode;

RetCode = KS_AllocWT(MAP1K, &p, 100);
if (RetCode == RC_TIME) {
    printf("No memory available after 100 ticks\n");
}
```

• SEE ALSO. . . . . KS_Alloc
KS_AllocW
KS_Dealloc

11.7. KS_AllocTimer

• SUMMARY . . . . . Allocate a timer and returns its address.

• CLASS . . . . . Timer

• SYNOPSIS . . . . .

```
K_TIMER *KS_AllocTimer(void);
```

• DESCRIPTION . . . The KS_AllocateTimer allocates a timer object from the timer pool and returns its address or NULL if no timer is available.

• RETURN VALUE . . A pointer to a K_TIMER, or NULL.

• EXAMPLE . . . . . .

```
K_TIMER  *MyTimer;

if ((MyTimer = AllocTimer()) == NULL) {
    printf("Fatal error : no more timers\n");
}
```

• SEE ALSO. . . . . KS_DeallocTimer
KS_StartTimer
KS_RestartTimer
KS_StopTimer

• SPECIAL NOTES . . The only legal use of a K_TIMER is as an argument to the microkernel services listed above. Programs should not access the structure fields directly.

11.8. KS_Dealloc

• SUMMARY . . . . . Deallocate a block of memory.

• CLASS. . . . . Memory management

• SYNOPSIS . . . . .

```
void KS_Dealloc(K_MAP   map,
                void  **block);
```

• DESCRIPTION . . . KS_Dealloc returns a memory block to the free pool of the specified map. A task switch will occur if an higher priority task was waiting for a memory block of the same partition.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
typedef void *   MyBlock;
MyBlock          p;
K_MAP            MAP1K;
int              RetCode;

RetCode = KS_AllocW (MAP1K, &p);
if (RetCode != RC_OK) {
    printf("Cannot allocate memory\n");
}
/*
 * code that uses memory block
 */
KS_Dealloc(MAP1K, &p);
```

• SEE ALSO. . . . . KS_Alloc
KS_AllocW

• SPECIAL NOTES . . The microkernel does not check the validity of the pointer argument. The user should ensure that blocks are deallocated to the same memory partition they were allocated from, and only once.

Using an invalid pointer will have unpredictable side effects.

11.9. KS_DeallocTimer

• SUMMARY . . . . . Deallocate a timer.

• CLASS . . . . . Timer

• SYNOPSIS . . . . .

```
void KS_DeallocTimer(K_TIMER *timer);
```

• DESCRIPTION . . . The KS_DeallocTimer microkernel service returns a timer object to the timer pool.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_TIMER *my_timer;

my_timer = KS_AllocTimer();        /* get timer    */
KS_StartTimer(my_timer,10,0,NULL);
KS_DeallocTimer(my_timer);         /* return timer */
```

• SEE ALSO. . . . . KS_AllocTimer

11.10. KS_Dequeue

• SUMMARY . . . . . Get an entry from a FIFO queue.

• CLASS. . . . . Queue

• SYNOPSIS . . . . .

```
int KS_Dequeue(K_QUEUE  queue,
               void    *data,
               int      size);
```

• DESCRIPTION . . . The KS_Dequeue microkernel service is used to read a data element from a FIFO queue. If the queue is not empty, the first (oldest) entry is removed from the queue and copied to the address provided by the caller. If the queue is empty, KS_Dequeue returns with an error code.

• RETURN VALUE . . RC_OK if operation succeeds, RC_FAIL otherwise.

• EXAMPLE . . . . . .

```
char      *data;
K_QUEUE    DATAQ;

if (KS_Dequeue(DATAQ, data, 1) == RC_OK) {
    if (*data == '#')
        startdemo ();
}
```

• SEE ALSO. . . . . KS_DequeueW
KS_DequeueWT
KS_Enqueue
KS_EnqueueW
KS_EnqueueWT

• SPECIAL NOTES . . The size parameter is the number of 8 bit bytes to be read. It should be equal to the width of the queue (which implies dequeueing only one entry at a time) or unpredictable side effects may occur.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.11. KS_DequeueW

• SUMMARY . . . . . Get an entry from a FIFO queue with wait.

• CLASS . . . . . Queue

• SYNOPSIS . . . . .

```
int KS_DequeueW(K_QUEUE  queue,
                void    *data;
                int      size)
```

• DESCRIPTION . . . KS_DequeueW is used to get an entry from a FIFO queue. If the queue is EMPTY, the calling task is put into a waiting list in order of its priority. If the queue is NOT EMPTY, the oldest entry in the queue is removed and returned to the calling task.

• RETURN VALUE . . RC_OK.

• EXAMPLE . . . . . .

```
int      command;
K_QUEUE  COMMANDS;
int      RetCode;

RetCode = KS_DequeueW(COMMANDS, &command, sizeof (int));
if (RetCode != RC_OK) {
    printf("problem reading from queue\n");
}
```

• SEE ALSO. . . . . KS_Dequeue
KS_DequeueWT
KS_Enqueue
KS_EnqueueW
KS_EnqueueWT

• SPECIAL NOTES . . The size parameter is the number of 8 bit bytes to be read. It should be equal to the width of the queue (which implies dequeueing only one entry at a time) or unpredictable side effects may occur.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.12. KS_DequeueWT

• SUMMARY . . . . . Get an Entry from a FIFO queue with timed out wait.

• CLASS. . . . . Queue

• SYNOPSIS . . . . .

```
int KS_DequeueWT(K_QUEUE  queue,
                 void    *data;
                 int      size,
                 K_TICKS  ticks)
```

• DESCRIPTION . . . KS_DequeueWT is used to get an entry from a FIFO queue. If the queue is EMPTY, the calling task is put into a waiting list in order of its priority until the QUEUE NOT EMPTY condition or until the timeout expires. If the queue is NOT EMPTY, the oldest entry in the queue is removed and returned to the calling task.

• RETURN VALUE . . RC_OK, or RC_TIME if the service timed out.

• EXAMPLE . . . . . .

```
int        command;
int        RetCode;
K_QUEUE    COMMANDS;
K_SEMA     OPERATOR_SLEEPS;

/*
 * read a command
 */
Retcode = KS_DequeueWT(COMMANDS, &command, sizeof (int), 100);
if (RetCode == RC_TIME) {
    /*
     * command was not supplied in time
     */
    KS_Signal(OPERATOR_SLEEPS);
}
```

• SEE ALSO. . . . . KS_Dequeue
KS_DequeueW
KS_Enqueue
KS_EnqueueW
KS_EnqueueWT

• SPECIAL NOTES . . The size parameter is the number of 8 bit bytes to be read. It should be equal to the width of the queue (which implies dequeueing only one entry at a time) or unpredictable side effects may occur.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.13. KS_DisableISR

• SUMMARY . . . . . Disable interrupt service routine.

• CLASS. . . . . Driver service.

• SYNOPSIS . . . . .

```
void KS_DisableISR(int IRQ);
```

• DESCRIPTION . . . The KS_DisableISR microkernel service disables the specified interrupt routine, and removes the corresponding ISR from the interrupt handler table. As a result, all subsequent interrupts from that source are ignored.

• RETURN VALUE . . None.

• EXAMPLE . . . . . .

```
KS_DisableISR(4)
printf("Interrupt 4 now disabled");
```

• SEE ALSO. . . . . KS_EnableISR
KS_EventW

• SPECIAL NOTES . . This service does not actually enter the microkernel and therefore cannot cause a task switch.

11.14. KS_Elapse

• SUMMARY . . . . . Compute elapsed time.

• CLASS . . . . . Timer

• SYNOPSIS . . . . .

```
K_TICKS KS_Elapse(K_TICKS *reftime);
```

• DESCRIPTION . . . The KS_Elapse microkernel service returns the elapsed time between two or more events. To get the elapsed time, one issues two calls. The first one is required to set the beginning time and the returned value should be discarded. The second and each subsequent call returns the number of clock ticks between the previous time marker and the current system low timer.

• RETURN VALUE . . Elapsed time in system ticks.

• EXAMPLE . . . . . .

```
K_TICKS    timestamp, diff1, diff2;
K_SEMA     SWITCH;

(void) KS_Elapse(&timestamp);  /* determine reference time */
KS_TestW(SWITCH);              /* wait for event           */
diff1 = KS_Elapse(&timestamp); /* time since first call    */
KS_TestW(SWITCH);              /* wait for event           */
diff2 = KS_Elapse(&timestamp); /* time since second call   */
```

• SEE ALSO. . . . . KS_LowTimer

KS_HighTimer

11.15. KS_EnableISR

• SUMMARY . . . . . Install an interrupt service routine.

• CLASS. . . . . Processor specific

• SYNOPSIS . . . . .

```
int KS_EnableISR(int   IRQ,
                 void  (*ISR0)(void));
```

• DESCRIPTION . . . The KS_EnableISR microkernel service provides a safe way to install an interrupt service routine. The operation may fail if the interrupt is already enabled. Any previously installed handler should be removed first (using KS_disableISR). An ISR can operate entirely in the background or it can pass the interrupt on to a waiting task by generating an event.

• RETURN VALUE . . RC_OK or RC_FAIL.

• EXAMPLE . . . . . .

```
extern void   ADC_ISR(void);

if (KS_EnableISR(4, ADC_ISR) != RC_OK) {
    printf("Unable to install the ADC ISR\n");
}
```

• SEE ALSO. . . . . KS_DisableISR

KS_EventW

Part 3 of this manual.

• Special Notes . . . . This service is processor specific and the prototype may vary between processors.

This service does not actually enter the microkernel and therefore cannot cause a task switch.

11.16. KS_Enqueue

• SUMMARY . . . . . Insert entry into a FIFO queue.

• CLASS . . . . . Queue

• SYNOPSIS . . . . .

```
int KS_Enqueue(K_QUEUE  queue,
               void    *data,
               int      size);
```

• DESCRIPTION . . . The KS_Enqueue microkernel service is used to put an entry in a FIFO queue. If the queue is not full the data is inserted at the end of the queue, and the call returns. If the queue is full, KS_Dequeue returns with an error code.

• RETURN VALUE . . RC_OK if operation succeeds, RC_FAIL otherwise.

• EXAMPLE . . . . . .

```
typedef struct{
    float X, Y;
} POINT;
POINT   next_point;
K_QUEUE POSITION;

/*
 * put X,Y coordinates in the POSITION queue,
 * if queue is full we don't care
 */

(void) KS_Enqueue(POSITION, &next_point, sizeof(POINT));
```

• SEE ALSO. . . . . KS_EnqueueW
KS_EnqueueWT
KS_Dequeue
KS_DequeueW
KS_DequeueWT

• SPECIAL NOTES . . The size parameter is the number of 8 bit bytes to be queued. It should be equal to the width of the queue (which implies enqueueing only one entry at

a time) or unpredictable side effects may occur.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.17. KS_EnqueueW

• SUMMARY . . . . . Insert entry into a FIFO queue with wait.

• CLASS . . . . . Queue

• SYNOPSIS . . . . .

```
int KS_EnqueueW(K_QUEUE  queue,
                void    *data,
                int      size);
```

• DESCRIPTION . . . KS_EnqueueW inserts an entry into a FIFO queue. If the queue is FULL, the calling task is put into a waiting list in order of its priority. When the queue becomes NOT FULL, the entry is inserted into the queue.

• RETURN VALUE . . RC_OK if operation succeeds, RC_FAIL otherwise.

• EXAMPLE . . . . . .

```
typedef struct{
    float X, Y;
} POINT;
POINT   next_point;
K_QUEUE POSITION;
int     RetCode;

/*
 * put X,Y coordinates in the POSITION queue,
 * if queue is full we wait for space
 */

RetCode = KS_EnqueueW(POSITION, &next_point, sizeof(POINT));
if (RetCode != RC_OK) {
    printf("failed to queue co-ordinates\n");
}
```

• SEE ALSO. . . . . KS_Enqueue
KS_EnqueueWT
KS_Dequeue
KS_DequeueW
KS_DequeueWT

• SPECIAL NOTES . . The size parameter is the number of 8 bit bytes to be queued. It should be

equal to the width of the queue (which implies enqueueing only one entry at a time) or unpredictable side effects may occur.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.18. KS_EnqueueWT

• SUMMARY . . . . . Insert an entry into a FIFO queue with timed out wait.

• CLASS . . . . . Queue

• SYNOPSIS . . . . .

```
void KS_EnqueueWT(K_QUEUE  queue,
                  void    *data;
                  int      size,
                  K_TICKS  ticks);
```

• DESCRIPTION . . . KS_Enqueue inserts an entry into a FIFO queue. If the queue is FULL, the calling task is put into a waiting list in order of its priority until the QUEUE NOT FULL condition or until the timeout expires. When the queue is NOT FULL, the entry is inserted into the queue.

• RETURN VALUE . . RC_OK, or RC_TIME if timed out.

• EXAMPLE . . . . . .

```
typedef struct{
    float X, Y;
} POINT;
POINT   next_point;
K_QUEUE POSITION;
int     RetCode;

/*
 * put X,Y coordinates in the POSITION queue,
 * if queue is full we wait for a maximum of
 * 100 ticks.
 */

RetCode = KS_EnqueueWT(POSITION,
                       &next_point,
                       sizeof(POINT),
                       100);
if (RetCode == RC_TIME) {
    printf("timed out queueing co-ordinates\n");
}
```


• SEE ALSO . . . . . KS_Enqueue
KS_EnqueueW
KS_Dequeue
KS_DequeueW
KS_DequeueWT

• SPECIAL NOTES . . The size parameter is the number of 8 bit bytes to be queued. It should be equal to the width of the queue (which implies enqueueing only one entry at a time) or unpredictable side effects may occur.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.19. KS_EventW

• SUMMARY . . . . . Wait for an event to be signalled.

• CLASS . . . . . Processor specific

• SYNOPSIS . . . . .

```
void KS_EventW(int IRQ);
```

• DESCRIPTION . . . This call will put the calling task in an EVENT WAIT State. When the event is raised, or was raised before the service call, the call will return and the event is cleared.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
while (1) {
    KS_EventW(7);       /* Wait for an event of type 7 */
    /*
     * do something with the event
     */
}
```

• SEE ALSO. . . . . KS_EnableISR
KS_DisableISR

• SPECIAL NOTES . . This service is processor specific and the prototype may vary according to processor type.

11.20. KS_GroupId

• SUMMARY . . . . . Returns the taskgroup set of the task.

• CLASS. . . . . Task management

• SYNOPSIS . . . . .

```
K_TGROUP KS_GroupId(void);
```

• DESCRIPTION . . . This microkernel service reads the taskgroup identifier of the calling task. Each group is indicated by a bit set in the word.

• RETURN VALUE . . The taskgroup identifier.

• EXAMPLE . . . . . .

```
K_TGROUP    SLAVES
if (KS_GroupId() & SLAVES) {
    /*
     * do some work since I am a SLAVE
     */
    work();
}
```

• SEE ALSO. . . . . KS_JoinG
KS_LeaveG

• SPECIAL NOTES . . This service is currently implemented as a macro.

11.21. KS_HighTimer

• SUMMARY . . . . . Read the processor's high precision timer.

• CLASS . . . . . Processor specific

• SYNOPSIS . . . . .

```
int KS_HighTimer(void);
```

• DESCRIPTION . . . This service reads the processor's high precision timer.

• RETURN VALUE . . The current high precision clock value.

• EXAMPLE . . . . . .

```
int    TimeNow;

TimeNow = KS_HighTimer();
```

• SEE ALSO. . . . . KS_Elapse
KS_LowTimer

• SPECIAL NOTES . . The precision and return type are processor dependent. On some processors reading the high precision timer without using this service may cause unpredictable side effects.

This service does not actually enter the microkernel and therefore cannot cause a task switch.

11.22. KS_InqMap

• SUMMARY . . . . . Returns the number of free blocks in a map.

• CLASS. . . . . Memory management

• SYNOPSIS . . . . .

```
int KS_InqMap(K_MAP map);
```

• DESCRIPTION . . . The KS_InqMap microkernel service returns the number of free blocks in the memory map.

• RETURN VALUE . . Number of free blocks.

• EXAMPLE . . . . . .

```
typedef void *   MyBlock;
MyBlock         p;
K_MAP            MAP3;
int              FreeBlocks;

FreeBlocks = KS_InqMap(MAP3);
if (FreeBlocks > 10)
    KS_AllocW(MAP3, &p);
```

• SEE ALSO. . . . . KS_Alloc
KS_AllocW
KS_AllocWT
KS_Dealloc

11.23. KS_InqQueue

• SUMMARY . . . . . Read the current queue depth.

• CLASS . . . . . Queue

• SYNOPSIS . . . . .

```
int KS_InqQueue(K_QUEUE queue);
```

• DESCRIPTION . . . The KS_InqQueue microkernel service allows the calling task to read the current number of entries in a queue.

• RETURN VALUE . . Current number of entries in the queue.

• EXAMPLE . . . . . .

```
K_QUEUE   CHARQ;
K_SEMA    XOFF;
int       depth;

depth = KS_InqQueue(CHARQ);
if (depth > 20)
    KS_Signal(XOFF);
```

• SEE ALSO. . . . . KS_PurgeQueue

11.24. KS_InqSema

• SUMMARY . . . . . Read the current semaphore count.

• CLASS. . . . . Semaphore

• SYNOPSIS . . . . .

```
int KS_InqSema(K_SEMA sema);
```

• DESCRIPTION . . . The KS_InqSema microkernel service allows the calling task to read the current count of the specified semaphore. It gives the difference between the number of times a semaphore was signalled and the number of times a task was waiting on that semaphore.

• RETURN VALUE . . Current semaphore count.

• EXAMPLE . . . . . .

```
K_SEMA      TestSema;
int         count;

count = KS_InqSema(TestSema);
if (count > 200)
    printf("Consumer tasks can't follow events\n");
```

• SEE ALSO. . . . . KS_ResetSema
KS_ResetSemaM

11.25. KS_JoinG

• SUMMARY . . . . . Add the calling task to the specified task groups.

• CLASS . . . . . Task

• SYNOPSIS . . . . .

```
void KS_JoinG(K_TGROUP);
```

• DESCRIPTION . . . This microkernel service sets the task group bits in the task group identifier.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
K_TGROUP     ALARM_GRP;
K_TGROUP     ABORT_GRP;

KS_JoinG(ALARM_GRP | ABORT_GRP);
```

• SEE ALSO. . . . . KS_JoinG
KS_LeaveG

• SPECIAL NOTES . . This service is currently implemented as a macro, and cannot cause a task switch.

11.26. KS_LeaveG

- SUMMARY . . . . . Remove the calling task from the specified task groups.
- CLASS. . . . . Task
- SYNOPSIS . . . . .

```
(void) KS_LeaveG(K_TGROUP);
```

- DESCRIPTION . . . This microkernel service clears the specified taskgroup bits in the task group identifier.
- RETURN VALUE . . NONE.
- EXAMPLE . . . . . .

```
KS_LeaveG(ABORT_GRP);
```

- SEE ALSO. . . . . KS_JoinG
  KS_LeaveG
  KS_GroupId
- SEE ALSO. . . . . This service is currently implemented as a macro, and cannot cause a task switch.

11.27. KS_Linkin

• SUMMARY . . . . . Start to read a datablock from a link and continue.

• CLASS . . . . . Processor specific

• SYNOPSIS . . . . .

```
int KS_Linkin(int   link,
              int   size,
              void *datablock);
```

• DESCRIPTION . . . This call reads data from a processor link. The call returns immediately with no descheduling of the calling task. It is possible to wait on the termination of this event at a later stage of the program.

• RETURN VALUE . . Event number.

• EXAMPLE . . . . . .

```
int     LinkEvent1, LinkEvent2;
int     Block1[20];
int     Block2[20];

LinkEvent1 = KS_Linkin (1, 20, Block1);
LinkEvent2 = KS_Linkin (2, 20, Block2);
/*
 * ... other code, not using Block
 */

/*
 * wait for data to be read
 */
KS_EventW(LinkEvent1);
KS_EventW(LinkEvent2);
```

• SEE ALSO. . . . . KS_LinkinW
KS_LinkinWT
KS_Linkout
KS_linkoutW
KS_LinkoutWT
iface.h

• SPECIAL NOTES . . Attempting to read from a link without using this service may cause unpre-

dictable side effects on some processors.

The size parameter is the number of 8 bit bytes to be read.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.28. KS_LinkinW

• SUMMARY . . . . . Read a datablock from a link with wait.

• CLASS . . . . . Processor specific

• SYNOPSIS . . . . .

```
void KS_LinkinW(int   link,
                int   size,
                void *datablock);
```

• DESCRIPTION . . . This call reads data from a processor link. The call returns when the datablock has been read in. Meanwhile the calling task is put in a LinkWait State.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
char        buf[128];

/*
 * read 15 bytes from link 3
 * store it in buf
 */
KS_LinkinW(3,15,buf);
```

• SEE ALSO. . . . . KS_Linkin
KS_LinkinWT
KS_Linkout
KS_linkoutW
KS_LinkoutWT
iface.h

• SPECIAL NOTES . . Attempting to read from a link without using this service may cause unpredictable side effects on some processors.

The size parameter is the number of 8 bit bytes to be read.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.29. KS_LinkinWT

• SUMMARY . . . . . Read a datablock from a link with timed out wait.

• CLASS. . . . . Processor specific

• SYNOPSIS . . . . .

```
int KS_LinkinWT(int      link,
                int      size,
                void    *datablock,
                K_TICKS  ticks);
```

• DESCRIPTION . . . This call reads data from a processor link. The call returns when the datablock has been read in or when the timeout has expired. Meanwhile the calling task is put in a LinkWait State.

• RETURN VALUE . . RC_OK if reading is finished before the timeout expires, RC_TIME otherwise.

• EXAMPLE . . . . . .

```
char        buf[128];
int         RetCode;

/*
 * read 15 bytes from link 3
 * store it in buf
 * don't wait more than 1000 ticks
 */
RetCode = KS_LinkinWT(3, 15, buf, 1000);
if (RetCode != RC_OK) {
    printf("timed out reading data\n");
}
```

• SEE ALSO. . . . . KS_Linkin
KS_LinkinW
KS_Linkout
KS_linkoutW
KS_LInkoutWT
iface.h

• SPECIAL NOTES . . This service is only implemented on transputers. Reading from a link without

using this service will have unpredictable side effects.

The size parameter is the number of 8 bit bytes to be read.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.30. KS_Linkout

• SUMMARY . . . . . Start to write a datablock to a link and continue.

• CLASS. . . . . Processor specific

• SYNOPSIS . . . . .

```
int KS_Linkout(int   link,
               int   size,
               void *datablock);
```

• DESCRIPTION . . . This call writes data to a processor link. It is possible to wait on the termination of this event at a later stage in the program.

• RETURN VALUE . . Event number.

• EXAMPLE . . . . . .

```
int       LinkEvent1, LinkEvent2 ;
int       Block1[200];
int       Block2[200];

/*
 * start output operations
 */
LinkEvent1 = KS_Linkout (1, 200, Block1);
LinkEvent2 = KS_Linkout (2, 200, Block2);
/*
 * ... other code, not using blocks
 */

/*
 * wait until link operation finishes
 */
KS_EventW(LinkEvent1);
KS_EventW(LinkEvent2);

/*
 * data in blocks can be overwritten
 */
```

• SEE ALSO. . . . . KS_LinkoutW
KS_LinkoutWT
KS_LinkinW
KS_LinkinWT
iface.h

• SPECIAL NOTES . . Attempting to write to a link without using this service may cause unpredictable side effects on some processors.

The size parameter is the number of 8 bit bytes to be written.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.31. KS_LinkoutW

• SUMMARY . . . . . Write a datablock to a link with wait.

• CLASS. . . . . Processor specific

• SYNOPSIS . . . . .

```
void KS_LinkoutW(int   link,
                 int   size,
                 void *datablock);
```

• DESCRIPTION . . . This call writes data to a processor link. The call returns when the writing of the datablock has finished. Meanwhile the calling task is put into the LINK-WAIT state.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
static char chanmessage[] = "This is a test message !\n";
int         msgsize;

msgsize = strlen(chanmessage);
KS_LinkoutW(3, msgsize, chanmessage);
```

• SEE ALSO. . . . . KS_Linkout
KS_LinkoutWT
KS_LinkinW
KS_linkinWT
iface.h

• SPECIAL NOTES . . Attempting to read from a link without using this service may cause unpredictable side effects on some processors.

The size parameter is the number of 8 bit bytes to be written.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.32. KS_LinkoutWT

• SUMMARY . . . . . Write a datablock to a link with timed out wait.

• CLASS . . . . . Processor specific

• SYNOPSIS . . . . .

```
int KS_LinkoutW(int     link,
                int     size,
                void   *datablock,
                K_TICKS ticks);
```

• DESCRIPTION . . . This call writes data to a processor link. The call returns when the writing of the datablock has finished or when the timeout has expired. Meanwhile the calling task is put into the LINKWAIT state.

• RETURN VALUE . . RC_OK if writing is finished before the timeout expires, RC_TIME otherwise.

• EXAMPLE . . . . . .

```
static char chanmessage[] = "This is a test message !\n";
int         msgsize;
int         RetCode;

msgsize = strlen(chanmessage);
RetCode = KS_LinkoutWT(3, msgsize, chanmessage, 100);
if (RetCode != RC_OK) {
    printf("write of message timed out\n");
}
```

• SEE ALSO. . . . . KS_Linkout
KS_LinkoutWT
KS_LinkinW
KS_LinkinWT
iface.h

• SPECIAL NOTES . . This service is only implemented on transputers. Reading from a link without using this service will have unpredictable side effects.

The size parameter is the number of 8 bit bytes to be written.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.33. KS_Lock

• SUMMARY . . . . . Lock a resource.

• CLASS. . . . . Resource

• SYNOPSIS . . . . .

```
int KS_Lock(K_RES resource);
```

• DESCRIPTION . . . The KS_Lock microkernel service provides a generalized way of protecting a logical resource. If the resource is busy at the time of request, the call returns with an error code. If the resource is available, the resource is marked BUSY to prevent others from using it. The logical resource can be anything such as a shared database, non-reentrant code, an I/O server, etc. Nested lock requests by the current owner are supported. KS_Unlock requests by non-owners are ignored.

• RETURN VALUE . . RC_OK if successful, RC_FAIL if otherwise.

• EXAMPLE . . . . . .

```
K_RES    GRAPHRES;
K_SEMA   BUZZER;

/*
 * if display is in use, use buzzer to give warning
 */
if (KS_Lock(GRAPHRES) != RC_OK) {
    KS_Signal(BUZZER);
} else {
    display_warning();
    KS_Unlock(GRAPHRES);
}
```

• SEE ALSO. . . . . KS_LockW
KS_LockWT
KS_Unlock
KS_UnlockW
KS_UnlockW

11.34. KS_LockW

• SUMMARY . . . . . Lock a resource with wait.

• CLASS . . . . . Resource

• SYNOPSIS . . . . .

```
int KS_LockW(K_RES resource);
```

• DESCRIPTION . . . The KS_LockW microkernel service provides a generalized way of protecting a logical resource. If the resource is in use at the time of request, the calling task is inserted into the waiting list in order of priority. If the resource is available, the resource is marked BUSY to prevent others from using it. The logical resource can be anything such as a shared database, non-reentrant code, an I/O server, etc. Nested lock requests by the current owner are supported. KS_Unlock requests by non-owners are ignored.

• RETURN VALUE . . RC_OK.

• EXAMPLE . . . . . .

```
K_RES       GRAPHRES;

(void) KS_LockW(GRAPHRES);
moveto(100,100);
lineto(200,100);
KS_Unlock(GRAPHRES);
```

• SEE ALSO. . . . . KS_LockW
KS_LockWT
KS_Unlock
KS_UnlockW
KS_UnlockW

11.35. KS_LockWT

• SUMMARY . . . . . Lock a resource with timed out wait.

• CLASS. . . . . Resource

• SYNOPSIS . . . . .

```
int KS_LockWT(K_RES   resource,
              K_TICKS ticks);
```

• DESCRIPTION . . . The KS_LockWT microkernel service provides a way of protecting a logical resource. If the resource is already owned by another task at the time of request, the calling task is inserted in the waiting list in order of priority. The calling task is removed from the waiting list at the moment the resource becomes available or if the time-out expires. If the resource is available, the resource is marked BUSY to prevent others from using it. The logical resource can be anything such as a shared database, non-reentrant code, an I/O server, etc. Nested lock requests by the current owner are supported. However, KS_Unlock requests by non-owners are ignored.

• RETURN VALUE . . RC_OK if successful, RC_TIME if not successful.

• EXAMPLE . . . . . .

```
K_RES    GRAPHRES;

if (KS_LockWT(GRAPHRES, 100) == RC_OK) {
    moveto(100,100);
    lineto(200,100);
    KS_Unlock(GRAPHRES)
} else {
    printf("cannot lock graphical display\n");
}
```

• SEE ALSO. . . . . KS_LockW
KS_LockWT
KS_Unlock
KS_UnlockW
KS_UnlockW

11.36. KS_LowTimer

• SUMMARY . . . . . Read the microkernel system timer.

• CLASS . . . . . Timer

• SYNOPSIS . . . . .

```
K_TICKS KS_LowTimer(void);
```

• DESCRIPTION . . . This call returns the current value in ticks of the microkernel system clock as defined during system generation.

• RETURN VALUE . . The current system clock value.

• EXAMPLE . . . . . .

```
K_TICKS     TimeNow;

TimeNow = KS_LowTimer();
```

• SEE ALSO. . . . . KS_Elapse
KS_HighTimer

• SPECIAL NOTES . . The precision is processor dependent.

11.37. KS_MoveData

• SUMMARY . . . . . Copy data.

• CLASS. . . . . Processor Specific

• SYNOPSIS . . . . .

```
void KS_MoveData(int   node,
                 int   size,
                 void *source,
                 void *destination,
                 int   direction);
```

• DESCRIPTION . . . Copy size bytes of data from a source address on a source node to a destination address on a destination node as indicated by the direction (RECV or SEND) and wait until the operation is finished. When the direction is RECV (receive), the node argument is the source node and the destination node is the one on which the receiving task is residing. When the direction is SEND (transmit), the node argument is the destination node and the source node is the one on which the sending task is residing. With SEND, the service returns when it is safe to overwrite the original data but the return does not indicate the remote termination of the operation. With RECV the service returns when all data has been moved to the destination area. On a single processor the operation is implemented as a memcpy operation.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
int    VideoNode;
char   Image1[1024]
char  *DisplayMem;

KS_MoveData(VideoNode,1024, &Image1, DisplayMem, SEND);
```

• SEE ALSO. . . . . iface.h

• SPECIAL NOTES . . This service must be used with care. While it provides for the highest datarates, copying to a wrong memory location can have unpredictable results. Therefore the task that receives the data must provide the sending task with a valid pointer. Note that the user must take account of the data representation. If the source and destination areas overlap, the results of this operation are unpredictable.

The size parameter is the number of 8 bit bytes to be transferred.

Care needs to be taken when specifying the size parameter on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

This service is only available on Virtuoso Classico.

11.38. KS_Nop

• SUMMARY . . . . . No operation.

• CLASS. . . . . Special

• SYNOPSIS . . . . .

```
void KS_Nop(void);
```

• DESCRIPTION . . . The KS_Nop microkernel service is included in the microkernel services for completeness. It is used as a benchmark for measuring the minimum interval to enter and exit the microkernel.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
int       i;
K_TICKS   timestamp, et;

KS_Elapse(&timestamp);
for (i = 0; i = 10000; i++) {
   KS_Nop();
}
et = KS_Elapse(&timestamp);
printf("10000 NOPs in %d ticks\n", et);
```

11.39. KS_NodeId

• SUMMARY . . . . . Node identifier of the calling task.

• CLASS . . . . . Task

• SYNOPSIS . . . . .

```
extern K_NODE KS_NodeId;
```

• DESCRIPTION . . . The KS_NodeId microkernel variable provides a means of knowing which node the task resides on.

• RETURN VALUE . . The node identifier.

• EXAMPLE . . . . . .

```
K_NODE MyNode = KS_NodeId;
```

• SPECIAL NOTES . . Modifying KS_NodeId will almost certainly have undesirable side effects. In future versions this variable may become read-only or be replaced by a function call.

11.40. KS_PurgeQueue

• SUMMARY . . . . . Purge the queue of all entries

• CLASS. . . . . Queue

• SYNOPSIS . . . . .

```
void KS_PurgeQueue(K_QUEUE queue);
```

• DESCRIPTION . . . The KS_PurgeQueue microkernel service forces a queue to a known EMPTY state. The KS_Enqueue waiting list is purged of all waiters and all entries are discarded.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
K_QUEUE  DATAQ;
K_TASK   PUTTER, GETTER;

KS_PurgeQueue(DATAQ); /* reset DATAQ to empty and restart the
        tasks*/
KS_Start(PUTTER); /* start producer task */
KS_Start(GETTER); /* start consumer task */
```

• SEE ALSO. . . . . KS_InqQueue

• SPECIAL NOTES . . Waiting tasks will receive RC_OK when they are restarted. This will be changed in a future version to the correct value, RC_FAIL.

11.41. KS_Receive

• SUMMARY . . . . . Receive a message.

• CLASS . . . . . Mailbox

• SYNOPSIS . . . . .

```
int KS_Receive(K_MBOX  mailbox,
               K_MSG  *msgstruc);
```

• DESCRIPTION . . . The KS_Receive message microkernel service is used to retrieve a message from a mailbox. If a matching message was found, the receiver's K_MSG will be updated using information from the sending task's K_MSG. If the receiving task has provided a valid destination pointer, the message data will be copied automatically, the service returns and the sending task is rescheduled. Otherwise, if the receiver has provided a NULL pointer, the copying of the message data is delayed until the receiving task issues a KS_ReceiveData service call. When the size given by the sender and receiver differ, the copy operation is limited to the smallest size. If the sending task was filled in as ANYTASK, the first matching message in the mailbox will be received. The KS_Receive call returns with an error if no matching message is available in the mailbox.

• RETURN VALUE . . RC_OK if successful, RC_FAIL if not successful.

• EXAMPLE . . . . . .

```
K_MSG       msg;
char        data[256];

msg.size = 256;
msg.tx_task = ANYTASK;
msg.rx_data = data;

if (KS_Receive (MAIL1, &msg)== RC_OK) {
    printf("Received %d bytes from %d\n",
            msg.size,
            msg.tx_task);
} else {
    printf("No matching message\n");
}
```


• SEE ALSO . . . . . KS_Send
KS_SendW
KS_SendWT
KS_ReceiveW
KS_ReceiveWT
KS_ReceiveData
Practical hints section of this manual

• SPECIAL NOTES . . The size member is the number of 8 bit bytes in the message.

Care needs to be taken when specifying sizes on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.42. KS_ReceiveData

• SUMMARY . . . . . Get the message data

• CLASS . . . . . Mailbox

• SYNOPSIS . . . . .

```
void KS_ReceiveData(K_MSG *msgstruc);
```

• DESCRIPTION . . . The KS_ReceiveData microkernel service is used to retrieve the message data that belongs to the previously received message. The message data is copied to the destination address provided by the receiving task and the sending task is rescheduled. If the size member of the message structure is set to zero, only the latter action is performed.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
K_MSG       msg;
K_MBOX      MBOX1;
char        data[256];

msg.size = 9999999;
msg.tx_task = ANYTASK;
msg.rx_data = data;

KS_ReceiveW (MBOX1, &msg);
if (msg.size > 256) {
     printf("message too large\n");
     msg.size = 0;
}
KS_ReceiveData(&msg);
```

• SEE ALSO. . . . . KS_Send
KS_SendW
KS_SendWT
KS_Receive
KS_ReceiveW

KS_ReceiveWT

• SPECIAL NOTES . . The size member is the number of 8 bit bytes in the message.

Care needs to be taken when specifying sizes on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.43. KS_ReceiveW

• SUMMARY . . . . . Receive a message with wait.

• CLASS . . . . . Mailbox

• SYNOPSIS . . . . .

```
int KS_ReceiveW(K_MBOX  mailbox,
                K_MSG  *msgstruc);
```

• DESCRIPTION . . . The KS_ReceiveW message microkernel service operates like the KS_Receive microkernel service except that the call only returns when there is a matching message in the mailbox.

• RETURN VALUE . . RC_OK.

• EXAMPLE . . . . . .

```
K_MSG       msg;
K_MBOX      MAIL1;
char        data[256];

msg.size = 256;
msg.tx_task = ANYTASK;
msg.rx_data = data;

KS_ReceiveW(MAIL1, &msg);
printf ("Received %d bytes from %d\n", msg.size, msg.tx_task);
```

• SEE ALSO. . . . . KS_Send
KS_SendW
KS_SendWT
KS_Receive
KS_ReceiveWT
KS_ReceiveData
Section 7.5 of this manual.

• SPECIAL NOTES . . The size member is the number of 8 bit bytes in the message.

Care needs to be taken when specifying sizes on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.44. KS_ReceiveWT

• SUMMARY . . . . . Receive a message with timed out wait.

• CLASS. . . . . Mailbox

• SYNOPSIS . . . . .

```
int KS_ReceiveWT(K_MBox   mailbox,
                 K_MSG   *msgstruc,
                 K_TICKS  ticks);
```

• DESCRIPTION . . . The KS_Receive message microkernel service operates like the KS_ReceiveW microkernel service except that the waiting is limited to time-out ticks.

• RETURN VALUE . . RC_OK if successful, RC_TIME if not successful.

• EXAMPLE . . . . . .

```
K_MSG    msg;
K_MBOX   MAIL1;
K_TASK   ANYTASK;
char     data[256];

msg.size = 256;
msg.tx_task = ANYTASK;
msg.rx_data = data;

if (KS_ReceiveWT (MAIL1, &msg, 100) == RC_OK) {
    printf("Received %d bytes from %d\n",
           msg.size,
           msg.tx_task);
} else {
    printf("Timed out on receive\n");
}
```

• SEE ALSO. . . . . KS_Send
KS_SendW
KS_SendWT
KS_Receive
KS_ReceiveW
KS_ReceiveData

• SPECIAL NOTES . . The size member is the number of 8 bit bytes in the message.

Care needs to be taken when specifying sizes on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.45. KS_ResetSema

• SUMMARY . . . . . Reset the semaphore count to zero.

• CLASS. . . . . Semaphore

• SYNOPSIS . . . . .

```
void KS_ResetSema(K_SEMA sema);
```

• DESCRIPTION . . . The ResetSema microkernel service resets the semaphore count to zero and hence erases all previous signalling operations. This microkernel service can be of use while recovering from a system error.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_SEMA    DeviceReady;

KS_ResetSema(DeviceReady);   /* reset semaphore */
KS_Wait(DeviceReady);        /* and wait on it  */
```

• SEE ALSO. . . . . KS_Signal
KS_Wait

11.46. KS_ResetSemaM

• SUMMARY . . . . . Reset a list of semaphores.

• CLASS . . . . . Semaphore

• SYNOPSIS . . . . .

```
void KS_ResetSemaM(K_SEMA *semalist);
```

• DESCRIPTION . . . The ResetSemaM microkernel service performs like the KS_ResetSema microkernel service except that it operates on a semaphore list. A semaphore list is an array of semaphores terminated by the predefined constant ENDLIST. This microkernel service reduces the number of Virtuoso microkernel service operations needed when multiple semaphores must be reset.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_SEMA  event;
K_SEMA  semalist[] = {
    SWITCH1,
    SWITCH2,
    SWITCH3,
    ENDLIST;
};

KS_ResetSemaM(&semalist);   /* forget switch history */
event = KS_WaitM(semalist); /* wait for switches     */
```

• SEE ALSO. . . . . KS_ResetSema
KS_Signal
KS_SignalM
KS_Wait
KS_WaitM

11.47. KS_RestartTimer

• SUMMARY . . . . . Restart a timer.

• CLASS. . . . . Timer

• SYNOPSIS . . . . .

```
void KS_RestartTimer(K_TIMER  *timer,
                     K_TICKS   delay,
                     K_TICKS   cyclic_period);
```

• DESCRIPTION . . . The KS_RestartTimer service restarts a timer with a new delay and optional cyclic period. It does not matter if the timer has already expired or not. The semaphore parameter given in a previous KS_StartTimer call remains in effect.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_TIMER   *timer4;
K_SEMA     TRIGGER;

timer4 = KS_AllocTimer();

KS_StartTimer(timer4,10,0,TRIGGER); /* signal in 10 ticks */
....
KS_RestartTimer (timer4,10,0);      /* restart countdown  */
```

• SEE ALSO. . . . . KS_StartTimer
KS_StopTimer

• SPECIAL NOTES . . KS_RestartTimer should be used only after a KS_StartTimer call for the same timer, otherwise the timer semaphore will be undefined.

11.48. KS_Resume

• SUMMARY . . . . . Resume a task.

• CLASS . . . . . Task

• SYNOPSIS . . . . .

```
void KS_Resume(K_TASK task);
```

• DESCRIPTION . . . The KS_Resume microkernel service clears the SUSPENDED state of a task caused by a KS_Suspend microkernel service.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_TASK    tapereader;

KS_Resume(tapereader);   /* resume the task */
```

• SEE ALSO. . . . . KS_ResumeG
KS_Suspend
KS_SuspendG

11.49. KS_ResumeG

• SUMMARY . . . . . Resume a task group.

• CLASS. . . . . Task

• SYNOPSIS . . . . .

```
void KS_ResumeG(K_TGROUP taskgroup);
```

• DESCRIPTION . . . The KS_ResumeG microkernel service clears the SUSPENDED state of a task group. It is equivalent to calling KS_Resume for every task in the specified groups but guarantees that the operation is performed as a single atomic action.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_TGROUP   DEVICES;

KS_ResumeG(DEVICES);   /* resume all device drivers */
```

• SEE ALSO. . . . . KS_ResumeG
KS_Suspend
KS_SuspendG

11.50. KS_Send

• SUMMARY . . . . . Send a message to a mailbox.

• CLASS . . . . . Mailbox

• SYNOPSIS . . . . .

```
int KS_Send(K_MBOX  mbox,
            K_PRIO  prio,
            K_MSG  *msg);
```

• DESCRIPTION . . . The KS_Send microkernel service inserts a message into the mailbox in order of the indicated priority. If the receiver used a valid pointer as the destination address, the microkernel initiates a data copy operation limited to the smallest of the message sizes indicated by the sender and receiver. At the end of the datacopy the sending task is rescheduled. If the rx_task field in the K_MSG was set to the predefined constant ANYTASK, the first receiver of the waiting list with a matching sender field will receive the message. If no receiver is waiting on a matching message, the call returns an error code and no mailbox insertion is made.

• RETURN VALUE . . RC_OK if successful, RC_FAIL if not successful.

• EXAMPLE . . . . . .

```
K_MSG     msg;
K_TASK    RECEIVER;
char      datastring[] = "testdata";

msg.size = strlen(datastring) + 1;
msg.tx_data = datastring;
msg.rx_task = RECEIVER;
if (KS_Send (BOX1, 2, &msg) == RC_OK) {
    printf("MSG Send OK, receiver was waiting\n");
```

• SEE ALSO. . . . . KS_SendW
KS_SendW
KS_Receive
KS_ReceiveW
KS_ReceiveWT
KS_ReceiveData

• SPECIAL NOTES . . The size member is the number of 8 bit bytes in the message.

Care needs to be taken when specifying sizes on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.51. KS_SendW

• SUMMARY . . . . . Send a message and wait for receiver.

• CLASS . . . . . Mailbox

• SYNOPSIS . . . . .

```
int KS_SendW(K_MBOX  mbox,
             K_PRIO  prio,
             K_MSG  *msg);
```

• DESCRIPTION . . . The send message and wait microkernel service is similar to the send message microkernel described above, except that the sending task will wait until a receiver is ready to accept the message.

• RETURN VALUE . . RC_OK.

• EXAMPLE . . . . . .

```
K_MSG   msg;
K_TASK  LOWTASK;
K_MBOX  POBOX9;
char    datastring[] = "testdata";

msg.task = LOWTASK;
msg.size = strlen(datastring) + 1;
msg.data = datastring;

(void)KS_SendW(POBOX9, 3, &msg);
printf("Message transmitted and received\n");
```

• SEE ALSO. . . . . KS_Send
KS_SendWT
KS_Receive
KS_ReceiveW
KS_ReceiveWT

• SPECIAL NOTES . . The size member is the number of 8 bit bytes in the message.

Care needs to be taken when specifying sizes on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.52. KS_SendWT

• SUMMARY . . . . . Send a message with timed out wait.

• CLASS. . . . . Mailbox

• SYNOPSIS . . . . .

```
int KS_SendWT(K_MBOX   box,
              K_PRIO   prio,
              K_MSG   *msg,
              K_TICKS  ticks);
```

• DESCRIPTION . . . The KS_SendWT microkernel service is similar to the KS_SendW microkernel service described above, except that the waiting time is limited to the number of ticks specified. If the call times out, no message is left in the mailbox.

• RETURN VALUE . . RC_OK if successful, RC_TIME if timed out.

• EXAMPLE . . . . . .

```
K_MSG     msg;
K_MBOX    BOX1;
K_TASK    RECEIVER;
char      datastring[] = "testdata";

msg.size = strlen(datastring) + 1;
msg.tx_data = datastring;
msg.rx_task = RECEIVER;

if (KS_SendWT (BOX1, 2, &msg, 100) == RC_OK) {
    printf("MSG sent OK, receiver was waiting\n");
} else {
    printf("Timed out no receiver\n");
```

• SEE ALSO. . . . . KS_Send
KS_SendWT
KS_Receive
KS_ReceiveW
KS_ReceiveW

• SPECIAL NOTES . . The size member is the number of 8 bit bytes in the message.

Care needs to be taken when specifying sizes on systems where characters are stored one per word, or where sizeof does not return the number of 8 bit bytes in a type.

11.53. KS_SetEntry

• SUMMARY . . . . . Set the entry point of a task.

• CLASS. . . . . Task

• SYNOPSIS . . . . .

```
void KS_SetEntry(K_TASK task,
                 void   (*function)(void));
```

• DESCRIPTION . . . The KS_SetEntry function will set the entry point of a task to a given function. At the next KS_Start call for the given task, the task will execute the specified function.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
extern void   new_function (void);
K_TASK        LOWTASK;

KS_SetEntry(LOWTASK, new_function);
KS_Abort(LOWTASK);
KS_Start(LOWTASK);  /* starts in new_function() */
```

• SEE ALSO. . . . . KS_Start
KS_Aborted

• SPECIAL NOTE. . . You can only set the entry point of a local task. Attempting to set the entry point of a non-local task will have unpredictable side effects.

11.54. KS_SetPrio

• SUMMARY . . . . . Change the priority of a task.

• CLASS . . . . . Task

• SYNOPSIS . . . . .

```
void KS_SetPrio(K_TASK  task,
                int     priority);
```

• DESCRIPTION . . . The KS_SetPrio microkernel service will change the priority of the specified task. A task switch will occur if the calling task is no longer the highest priority runable task.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
int   MyPrio;

MyPrio = KS_TaskPrio();        /* save my priority * /
KS_SetPrio(KS_TaskId, 3);      /* raise my priority */

KS_Lock(FILERES);
readfile();
KS_Unlock (FILERES);

KS_SetPrio(KS_TaskId, MyPrio); /* reset my priority */
```

• SEE ALSO. . . . . KS_TaskPrio

• SPECIAL NOTES . . This microkernel service does not modify the order of insertion in any of the waiting lists. If KS_SetPrio is used to raise the probability of becoming runable, if must be used before invoking the desired microkernel service. Generally speaking changing the priority of a task must be used with caution and only used for addressing time limited scheduling problems.

A low numeric value is a high priority. Thus the highest priority task has a priority of one.

11.55. KS_SetSlice

• SUMMARY . . . . . Set timeslicing period.

• CLASS. . . . . Task

• SYNOPSIS . . . . .

```
void KS_SetSlice (K_TICKS ticks,
                  K_PRIO  prio);
```

• DESCRIPTION . . . The KS_SetSlice kernel service will timeslice equal priority tasks if they have a priority lower than the one indicated. Note that if there is only one task with the same priority that is runnable, the service will have no effect as the task will be rescheduled immediately.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
/*
 * set the timeslice period to 300 ms
 * for all tasks with a priority below 20
 */
KS_SetSlice(300, 20);
```

• SEE ALSO. . . . . KS_Yield

• SPECIAL NOTE. . . A low numeric value is a high priority. Thus the highest priority task has a priority of one.

This service is only available in Virtuoso Classico.

11.56. KS_SetWlper

• SUMMARY . . . . . Set workload period.

• CLASS . . . . . Special

• SYNOPSIS . . . . .

```
void KS_SetWlper(int period);
```

• DESCRIPTION . . . The KS_SetWlper microkernel service is used to specify the workload measuring interval. The KS_Workload service will return the average workload over the last full period and the current one. The period must be specified in milliseconds and be between 10 and 1000 milliseconds..

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
/*
 * set the workload period to 300 ms
 */
KS_SetWlper(300);
```

• SEE ALSO. . . . . KS_Workload

The workload monitor section in the debugging chapter of this manual.

• SPECIAL NOTES . . The actual range of the period is processor dependent as each processor has a different granularity for its timer.

11.57. KS_Signal

• SUMMARY . . . . . Signal a semaphore.

• CLASS. . . . . Semaphore

• SYNOPSIS . . . . .

```
void KS_Signal(K_SEMA sema);
```

• DESCRIPTION . . . The KS_Signal microkernel service is used to signal a semaphore. If the semaphore waiting list is empty, the semaphore counter is incremented, otherwise the first (highest priority) waiting task is rescheduled.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_SEMA   SWITCH;

KS_Signal(SWITCH);   /* signal semaphore SWITCH */
```

• SEE ALSO. . . . . KS_SignalM
KS_ResetSemaM
KS_Test
KS_Test(M)(W)(T)

11.58. KS_SignalM

• SUMMARY . . . . . Signal a list of semaphores.

• CLASS . . . . . Semaphore

• SYNOPSIS . . . . .

```
void KS_SignalM(K_SEMA semalist);
```

• DESCRIPTION . . . The KS_SignalM microkernel service is equivalent to (but much faster than) calling KS_Signal for every semaphore in the list. The signaling of the semaphores is atomic if this service is used instead of multiple KS_Signal calls. A semaphore list is an array of semaphores terminated by the predefined constant ENDLIST.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_SEMA Alarms[] = {
    ALARM1,
    ALARM2,
    PANIC,
    ENDLIST
}

KS_SignalM(Alarms);
```

• SEE ALSO. . . . . KS_Signal
KS_ResetSemaM
KS_Test
KS_Test(M)(W)(T)

11.59. KS_Sleep

• SUMMARY . . . . . Deschedule for a number of ticks.

• CLASS. . . . . Task

• SYNOPSIS . . . . .

```
void KS_Sleep(K_TICKS ticks);
```

• DESCRIPTION . . . The KS_Sleep microkernel service deschedules the calling task (itself) and reschedules it after the specified number of ticks. It allows a task to delay execution for a certain time in a simple way, without using timers or semaphores.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
KS_Sleep(100); /* delay for at least 100 ticks */
```

• SEE ALSO. . . . . KS_Suspend

• SPECIAL NOTES . . A sleep of zero ticks is equivalent to a call to KS_Yield.

11.60. KS_Start

• SUMMARY . . . . . Start a task

• CLASS . . . . . Task

• SYNOPSIS . . . . .

```
void KS_Start(K_TASK task);
```

• DESCRIPTION . . . The KS_Start task microkernel service is used to start a task from its beginning. The specified task is made runable and its entry point is called. If the task is of higher priority than the current task, a context switch is performed and the new task runs, otherwise control is returned to the caller.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
K_TASK  SHUTDOWN;

KS_Start(SHUTDOWN);   /* start SHUTDOWN */
```

• SEE ALSO. . . . . KS_Abort

• SPECIAL NOTES . . This is NOT a recommended way to start a task at short notice (or to perform a ‘programmed’ task switch). A much faster way to start execution of a task is to start it before it is actually needed, and make it wait on a semaphore or use a KS_Suspend/KS_Resume pair.

If this service is used in an attempt to restart a task that is already running the result is unpredictable (but almost certainly not what was wanted).

11.61. KS_StartG

• SUMMARY . . . . . Start a group of tasks.

• CLASS. . . . . Task

• SYNOPSIS . . . . .

```
void KS_StartG(K_TGROUP group);
```

• DESCRIPTION . . . The KS_StartG microkernel service is used to start a group (or a number of groups) of tasks. This is much more efficient than starting each task individually, as at most one task switch will be needed. Its use also guarantees that on each node marking the tasks READY is an atomic operation.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
K_TGROUP   WORKERS;
K_TGROUP   SLAVES;

KS_StartG(WORKERS | SLAVES);
```

• SEE ALSO. . . . . KS_Abort

• SPECIAL NOTES . . If this service is used in an attempt to restart a task that is already running the result is unpredictable (but almost certainly not what was wanted).

11.62. KS_StartTimer

• SUMMARY . . . . . Start a timer.

• CLASS . . . . . Timer

• SYNOPSIS . . . . .

```
void KS_StartTimer(K_TIMER *timer,
                   K_TICKS  delay,
                   K_TICKS  cyclic_period,
                   K_SEMA   sema);
```

• DESCRIPTION . . . KS_StartTimer starts the timer with an initial delay, specified in ticks, (one shot operation) and from then on with an optional cyclic period. The semaphore will be signalled each time the timer triggers.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_TICKS  *timer4;
K_SEMA    MySema;

timer4 = KS_AllocTimer();
/*
 * signal Mysema after 100 ticks,
 * and then every 20 ticks
 */
KS_StartTimer (timer4, 100, 20, MySema);
```

• SEE ALSO. . . . . KS_RestartTimer
KS_StopTimer

• SPECIAL NOTES . . The delay and cyclic period should be greater than zero, or unpredictable side effects may occur.

11.63. KS_StopTimer

• SUMMARY . . . . . Stop a timer.

• CLASS. . . . . Timer

• SYNOPSIS . . . . .

```
void KS_StopTimer(K_TIMER *timer);
```

• DESCRIPTION . . . KS_StopTimer removes an active timer from the timer queue. If the timer has already expired, this call has no effect.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_TIMER   *timer4;
K_SEMA     MySema;

timer4 = KS_AllocTimer();
/*
 * signal MySema after 100 ticks,
 * and then every 20 ticks
 */
KS_StartTimer(timer4, 100, 20, MySema);
....
KS_StopTimer(timer4);
/*
 * no more signals to MySema now
 */
```

• SEE ALSO. . . . . KS_StartTimer

11.64. KS_Suspend

• SUMMARY . . . . . Suspend execution of a task

• CLASS . . . . . Task

• SYNOPSIS

```
void KS_Suspend(K_TASK task);
```

• DESCRIPTION . . . The KS_Suspend microkernel service causes the specified task to be placed into a suspended state. The suspended state will remain in force until it is removed by a KS_Resume or KS_Abort microkernel service. A task may suspend itself.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_TASK   DETECT;

KS_Suspend(DETECT);     /* suspend task DETECT */
KS_Suspend(KS_TaskId);  /* suspend myself      */
```

• SEE ALSO. . . . . KS_SuspendG
KS_Resume
KS_ResumeG

11.65. KS_SuspendG

• SUMMARY . . . . . Suspend execution of a group of tasks.

• CLASS. . . . . Task

• SYNOPSIS . . . . .

```
void KS_SuspendG(K_TGROUP group);
```

• DESCRIPTION . . . The KS_SuspendG microkernel service is equivalent to calling KS_Suspend for every task that is a member of the specified group(s). The service is performed atomically on each node.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
K_TGROUP   CONTROL;

KS_SuspendG(CONTROL); /* suspend all tasks in CONTROL group */
```

• SEE ALSO. . . . . KS_Suspend
KS_Resume
KS_ResumeG

11.66. KS_TaskId

• SUMMARY . . . . . Read task identifier.

• CLASS . . . . . Task

• SYNOPSIS . . . . .

```
K_TASK KS_TaskId;
```

• DESCRIPTION . . . The KS_TaskId microkernel variable contains the calling task's identifier.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
printf("Hi, I am task %08x\n", KS_TaskId);
```

• SEE ALSO. . . . . KS_TaskPrio
KS_NodeId

• SPECIAL NOTES . . This variable should be treated as read-only. This may be enforced by future versions of the microkernel, or the variable may be replaced by a function.

11.67. KS_TaskPrio

• SUMMARY . . . . . Read current task priority.

• CLASS. . . . . Task

• SYNOPSIS . . . . .

```
int KS_TaskPrio;
```

• DESCRIPTION . . . The KS_TaskPrio microkernel variable contains the calling task's current priority.

• RETURN VALUE . . NONE.

• EXAMPLE . . . . . .

```
K_MBOX        BOX1;
K_MSG         msg;


...
KS_SendW(BOX1,KS_TaskPrio,&msg);/* Send at current priority */
```

• SEE ALSO. . . . . KS_SetPrio

• SPECIAL NOTES . . This variable is actually accessed via a macro and should be treated as read-only. This may be enforced by future versions of the microkernel, or the variable may be replaced by a function.

11.68. KS_Test

• SUMMARY . . . . . Test a semaphore.

• CLASS . . . . . Semaphore

• SYNOPSIS . . . . .

```
int KS_Test(K_SEMA sema);
```

• DESCRIPTION . . . The KS_Test microkernel service is used to test whether a specified event has occured. The event must be associated with the given semaphore. If the semaphore count is greater than zero, the call returns RC_OK and the semaphore count is decremented by one. Otherwise the calling task returns with an RC_FAIL.

• RETURN VALUE . . RC_OK or RC_FAIL.

• EXAMPLE . . . . . .

```
K_SEMA semaphore;

if (KS_Test(semaphore) == RC_OK) {
     printf("semaphore was signalled\n");
} else {
    printf("semaphore not signalled\n");
}
```

• SEE ALSO. . . . . KS_TestW
KS_TestWT
KS_TestMW
KS_TestMWT
KS_Signal
KS_SignalM

• SPECIAL NOTES . . In Virtuoso Micro this service is implemented as a macro.

11.69. KS_TestMW

• SUMMARY . . . . . Test multiple semaphores.

• CLASS. . . . . Semaphore

• SYNOPSIS . . . . .

```
K_SEMA KS_TestMW(K_SEMA *list);
```

• DESCRIPTION . . . The KS_TestMW microkernel service performs the same function as the KS_TestW microkernel service except that it uses a semaphore list. This function operates as a logical OR. The occurrence of an event associated with any one of the semaphores in the list will cause resumption of the waiting task.

• RETURN VALUE . . Semaphore identifier of the event that occurred.

• EXAMPLE . . . . . .

```
K_SEMA   Event;
K_SEMA   List1 [] = {
    SWITCH1,
    SWITCH2,
    TIMERA,
    ENDLIST
};

Event = KS_TestMW(List1); /* wait for any of 3 events */
```

• SEE ALSO. . . . . KS_Test
KS_TestWT
KS_TestMWT
KS_Signal
KS_SignalM

• SPECIAL NOTES . . In the situation where multiple events occur, only the first one will be returned. The rest will be serviced correctly on subsequent KS_Test(M)(W) calls. Note that a significant overhead can result when the semaphores reside on remote processors.

In Virtuoso Micro this service is implemented as a macro.

11.70. KS_TestMWT

• SUMMARY . . . . . Test multiple semaphores with timed out wait.

• CLASS . . . . . Semaphore

• SYNOPSIS . . . . .

```
K_SEMA KS_TestMWT(K_SEMA  *list,
                  K_TICKS  ticks);
```

• DESCRIPTION . . . The KS_TestMWT microkernel service performs the same function as the KS_TestMW microkernel service except that the waiting time is limited to the specified number of ticks. This function operates as a logical OR. The occurrence of an event associated with any one of the semaphores in the list, or a timeout, will cause resumption of the waiting task.

• RETURN VALUE . . Semaphore identifier of the event that occurred, or the predefined constant ENDLIST if timed out.

• EXAMPLE . . . . . .

```
K_SEMA   Event;
K_SEMA   List1[] = {
    SWITCH1,
    SWITCH2,
    TIMERA,
    ENDLIST
};

Event = KS_TestMWT(List1, 100);
if (Event == ENDLIST) {
    printf("Timed out after 100 ticks\n");
} else {
    printf("one of the three events happened\n");
}
```

• SEE ALSO. . . . . KS_Test
KS_TestMW
KS_TestMWT
KS_Signal
KS_SignalM

• SPECIAL NOTES . . In the situation where multiple events occur, only the first one will be returned. The rest will be serviced correctly on subsequent KS_Test calls. Note that a significant overhead can result when the semaphores reside on remote processors.

In Virtuoso Micro this service is implemented as a macro.

11.71. KS_TestW

• SUMMARY . . . . . Test a semaphore.

• CLASS . . . . . Semaphore

• SYNOPSIS . . . . .

```
int KS_TestW(K_SEMA sema);
```

• DESCRIPTION . . . The KS_TestW microkernel service is used to make a task wait for a specified event to occur. The event must be associated with the given semaphore. If the semaphore count is greater than zero, the call returns immediately and the semaphore count is decremented by one. Otherwise the calling task is put into the semaphore waiting list in order of the task priority.

• RETURN VALUE . . RC_OK.

• EXAMPLE . . . . . .

```
K_SEMA    ADC_SEMA;

KS_TestW(ADC_SEMA);
```

• SEE ALSO. . . . . KS_Test
KS_TestWT
KS_TestMW
KS_TestMWT
KS_Signal
KS_SignalM

• SPECIAL NOTES . . In Virtuoso Micro this service is implemented as a macro.

11.72. KS_TestWT

• SUMMARY . . . . . Test a semaphore with time out.

• CLASS. . . . . Semaphore

• SYNOPSIS . . . . .

```
int KS_TestWT(K_SEMA   sema,
              K_TICKS  ticks);
```

• DESCRIPTION . . . The KS_TestWT microkernel service is used to make a task wait for a specified event to occur. The event must be associated with the given semaphore. If the semaphore count is greater than zero, the call returns immediately and the semaphore count is decremented by one. Otherwise the calling task is put into the semaphore waiting list in order of the task priority. The task is removed from the waiting list when the semaphore is signalled or when the timeout expires.

• RETURN VALUE . . RC_OK if the semaphore was signalled, RC_TIME if timed out.

• EXAMPLE . . . . . .

```
K_SEMA   ADC_SEMA;

if (KS_TestWT(ADC_SEMA, 10) == RC_TIME) {
    printf("No ADC event in 10 ticks\n");
}
```

• SEE ALSO. . . . . KS_Test
KS_TestW
KS_TestMW
KS_TestMWT
KS_Signal
KS_SignalM

• SPECIAL NOTES . . In Virtuoso Micro this service is implemented as a macro.

11.73. KS_Unlock

• SUMMARY . . . . . Release logical resource.

• CLASS . . . . . Resource

• SYNOPSIS . . . . .

```
void KS_Unlock(K_RES resource);
```

• DESCRIPTION . . . The KS_Unlock microkernel service decrements the lock level of a logical resource. If the new lock level is zero, the resource is unlocked and available for other users. The call is ignored if the calling task is not the owner of the resource.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
K_RES    DATABASE;

KS_Lock(DATABASE);     /* grab shared resource           */
Add_Record();          /* and update it                  */
KS_Unlock(DATABASE);   /* now release it for other tasks */
```

• SEE ALSO. . . . . KS_Lock
KS_LockW
KS_LockWT

• SPECIAL NOTES . . Upon unlocking, the resource is allocated to the next task in the (priority ordered) waiting list.

11.74. KS_User

• SUMMARY . . . . . Execute function at microkernel level.

• CLASS. . . . . Special

• SYNOPSIS . . . . .

```
int KS_User(int  (*function)(void *),
                 void *ArgList)
```

• DESCRIPTION . . . The KS_User microkernel service is used to execute a user function at the priority level of the microkernel. All microkernel service requests (from ISR's, other nodes, or the timer system) will be queued until the user function returns. The user function should be short, and must not issue any microkernel service calls. Practically speaking, during the execution a preemption of tasks is disabled.

• RETURN VALUE . . The return value of the function.

• EXAMPLE . . . . . .

```
struct arg {
    ....
} MyArgs;

extern int  MyFunction(MyArgs *);

result = KS_User(MyFunction, &MyArgs);
```

• SPECIAL NOTE. . . The second parameter to the user function should be a pointer to a structure containing any parameters the function needs. Needless to say, grave disorder will result if the caller and the routine do not agree on the structure layout.

This service cannot be used to run a the function on a remote node.

11.75. KS_Wait

• SUMMARY . . . . . Wait on a semaphore.

• CLASS . . . . . Semaphore

• SYNOPSIS . . . . .

```
int KS_Wait(K_SEMA sema);
```

• DESCRIPTION . . . The KS_Wait microkernel service is used to test whether a specified event has occurred. The event must be associated with the given semaphore. If the semaphore count is greater than zero, the call returns RC_OK and the semaphore count is decremented by one. Otherwise the calling task returns with an RC_FAIL.

• RETURN VALUE . . RC_OK or RC_FAIL.

• EXAMPLE . . . . . .

```
K_SEMA semaphore;

if (KS_Wait(semaphore) == RC_OK) {
    printf("semaphore was signalled\n");
} else {
    printf("semaphore not signalled\n");
}
```

• SEE ALSO. . . . . KS_WaitT
KS_WaitM
KS_WaitMT
KS_Signal
KS_SignalM

• SPECIAL NOTES . . In Virtuoso Classico this service is implemented as a macro.

11.76. KS_WaitM

• SUMMARY . . . . . Wait on a list of semaphores.

• CLASS. . . . . Semaphore

• SYNOPSIS . . . . .

```
K_SEMA KS_WaitM(K_SEMA *list);
```

• DESCRIPTION . . . The KS_WaitM microkernel service performs the same function as the KS_Wait microkernel service except that it uses a semaphore list. This function operates as a logical OR. The occurrence of an event associated with any one of the semaphores in the list will cause resumption of the waiting task.

• RETURN VALUE . . Semaphore identifier of the event that occurred, or the predefined constant ENDLIST if no semaphore was signalled.

• EXAMPLE . . . . . .

```
K_SEMA   Event;
K_SEMA   List1[] = {
    SWITCH1,
    SWITCH2,
    TIMERA,
    ENDLIST
};

Event = KS_WaitM(List1); /* test for any of 3 events */
```

• SEE ALSO. . . . . KS_Wait
KS_WaitT
KS_WaitMT
KS_Signal
KS_SignalM

• SPECIAL NOTES . . In the situation where multiple events occur, only the first one will be returned. The rest will be serviced correctly on subsequent KS_Wait(M) calls. Note that a significant overhead can result when the semaphores reside on remote processors.

This call is only available on Virtuso Micro.

11.77. KS_WaitMT

• SUMMARY . . . . . Wait on multiple semaphores with time out.

• CLASS . . . . . Semaphore

• SYNOPSIS . . . . .

```
K_SEMA KS_WaitMT(K_SEMA  *list,
                 K_TICKS  ticks);
```

• DESCRIPTION . . . The KS_WaitMT microkernel service performs the same function as the KS_WaitM microkernel service except that the waiting time is limited to the specified number of ticks. This function operates as a logical OR. The occurrence of an event associated with any one of the semaphores in the list, or a timeout, will cause resumption of the waiting task.

• RETURN VALUE . . Semaphore identifier of the event that occurred, or the predefined constant ENDLIST if timed out.

• EXAMPLE . . . . . .

```
K_SEMA   Event;
K_SEMA   List1[] = {
    SWITCH1,
    SWITCH2,
    TIMERA,
    ENDLIST
};

Event = KS_WaitMT(List1, 100);
if (Event == ENDLIST) {
    printf("Timed out after 100 ticks\n");
} else {
    printf("one of the three events happened\n");
}
```

• SEE ALSO. . . . . KS_Wait
KS_WaitM
KS_Signal
KS_SignalM

• SPECIAL NOTES . . In the situation where multiple events occur, only the first one will be returned. The rest will be serviced correctly on subsequent KS_Wait(M) calls. Note that a significant overhead can result when the semaphores

reside on remote processors.

In Virtuoso Classico this service is implemented as a macro.

11.78. KS_WaitT

• SUMMARY . . . . . Wait on a semaphore with time out.

• CLASS . . . . . Semaphore

• SYNOPSIS . . . . .

```
int KS_WaitT(K_SEMA   sema,
             K_TICKS  ticks);
```

• DESCRIPTION . . . The KS_WaitT microkernel service is used to make a task wait for a specified event to occur. The event must be associated with the given semaphore. If the semaphore count is greater than zero, the call returns immediately and the semaphore count is decremented by one. Otherwise the calling task is put into the semaphore waiting list in order of the task priority. The task is removed from the waiting list when the semaphore is signalled or when the timeout expires.

• RETURN VALUE . . RC_OK if the semaphore was signalled, RC_TIME if the call timed out.

• EXAMPLE . . . . . .

```
K_SEMA   ADC_SEMA;

if (KS_WaitT(ADC_SEMA, 10) == RC_TIME) {
    printf("No ADC event in 10 ticks\n");
}
```

• SEE ALSO. . . . . KS_Wait
KS_WaitM
KS_WaitMT
KS_Signal
KS_SignalM

• SPECIAL NOTES . . In Virtuoso Classico this service is implemented as a macro.

11.79. KS_Workload

• SUMMARY . . . . . Read the current workload.

• CLASS. . . . . Special

• SYNOPSIS

```
int KS_Workload(void);
```

• DESCRIPTION . . . The workload microkernel service returns the current workload as a number ranging from 0 to 1000. Each unit equals .1 % of the time the CPU was not idling during the last workload measuring interval.

• RETURN VALUE . . The measured workload.

• EXAMPLE . . . . . .

```
int   wl;

wl = KS_Workload();
```

• SEE ALSO. . . . . KS_SetWlper

11.80. KS_Yield

- SUMMARY . . . . . Yield the CPU to another task
- CLASS . . . . . Task
- SYNOPSIS . . . . .

```
void KS_Yield(void);
```

- DESCRIPTION . . . The KS_Yield microkernel service will voluntary yield the processor to the next equal priority task that is runable. Using KS_Yield, it is possible to achieve the effect of round robin scheduling. If no task with the same priority is runable and if no task switch occurs, the calling task resumes execution.
- RETURN VALUE . . NONE
- EXAMPLE . . . . . .

```
KS_Yield();
```

- SEE ALSO. . . . . KS_SetSlice

12. Hostserver and netloader

12.1. Host server functionality

The host server is a program that does not run on the target but on a host computer, e.g. a PC or workstation. It communicates with the target using the available communication mechanism (most often a serial line, an ISA bus interface, or a VME interface). Some custom solutions can be more complicated and communicate between the target and the host computer using intermediate processors, ethernet, etc.

The host server is optional when developing programs with Virtuoso (as all Virtuoso code is rommable) but it greatly helps during the development and maintenance phase.

The host server provides two main functions :

1. Resetting and booting the target;
2. Providing runtime I/O and executing Remote Procedure Calls.

The host server is board and host dependent. We will use the generic name HOST_X to describe its use. Note that on older versions of the software, a different syntax for the options maybe in use.

12.1.1. Resetting and booting the target

The command line syntax is as follows :

```
HOST_X [- options] <network file>
```

Options are :

-r :	Reset and set up all boards listed in the network file.
-l :	Load the executable files on all nodes listed in the network file.
-s :	Enable server functions. If this option is not given, the program will terminate after loading the network.
-v :	Make the program more verbose
-q :	Make the program less verbose
-z :	Single node operation, no call to netload() in main1.c. Normally used for single processor targets.

Options can be separated (-l -s -q), or combined (-lsq), and may be given in any order.

A network file must have the extension .NLI. You don't have to type it as it will be appended by the host program.

Note also that in older versions of the software, the single processor versions did not take the *.NLI file but the executable image file as first parameter. To facilitate portability we are applying the multiprocessor approach for single as well as the multiprocessor packages, even if this means that the node information and the interconnection topology is not used.

Examples :

```
HOST_X <ENTER>
     displays the help message
HOST_X test <ENTER>
     reads and syntax checks the network file 'test.nli'
HOST_X -rlsvv test <ENTER>
     runs the application described in 'test.nli', shows full
     details of booting operations, and provides services to
     application.
```

12.2. Network file

The network file (*.NLI) is board and target dependent. See the read.me files and the examples for the right contents to use with your board. We provide here a generic explanation.

Network (.NLI) files are text files that can be created using a program editor. Most of the data is in tabular form, with fields separated by spaces or tabs. Comment lines (# as first printing character) can be freely inserted.

All *.NLI file can have different sections. But not all boards require the same set to be defined in the *.NLI file !

12.2.1. Host interface definition.

<INTERFACE_TYPE> <IO_ADDRESS>

Examples :

```
LINKC012 150 /* HEMA TA1 transputer link interface */
MEGALINK 200 /* Sang Megalink */
HEPC2 300 /* Hunt Engineering HEPC2 */
```


12.2.2. List of boards

Example :

```
# TYPE ID IOB_0 IOB_1 DPRAM CTRL CONF
#-------------------------------
DPCC40 B1 300 0 D400 0800 0000
DPCC40 B2 340 0 0000 0 0
```

TYPE : keyword for board definition

ID : name for the board

IOB_0 : PC I/O Block 0 address. This will be required for all boards.

I0B_1 : PC I/O Block 1 address. Required if you use DB40.EXE

DPRAM : This is the SEGMENT address of the Dual Port Ram on the PC side. This is required on the root node only.

CTRL : Value written to the Control Register after reset.

CONF : Value written to the Config Register after reset.

See the DPCC40 manual for a detailed description of CTRL and CONF.

12.2.3. List of nodes.

Example :

```
# TYPE ID LOCATION SITE LBCR GBCR IACK FILE
#------------------ ---------------------------------
TIM40 NODE1 B1 PRI 3deba050 32778010 80000000 test1.out
TIM40 NODE2 B1 SEC 3deba050 32778010 80000000 test2.out
```

TYPE : keyword for node definition

ID : symbolic name of the node

LOCATION : symbolic name of board and site (PRI or SEC), if any

MEMORY CONTROL WORDS (target dependent)

The following three values are required by all C40 boot loaders.

LBCR : local bus control register

GBCR : global bus control register

IACK : iack address

FILE : the executable file to be loaded on the node.

Note : on some recent board specific ports, two files need to be defined. The first one is the bootstrap loader, while the second one is the executable image.

12.2.4. Root node definition.

ROOTNODE <symbolic name of root node>

Example :

```
ROOTNODE NODE1
```

The root node is the node that is directly interfaced to the host.

When booting, the root node is loaded first, then all others are booted from one of their comports.

When the application is running, driver tasks placed on the root node will interface to the server program on the host, and provide services for the C40 network.

12.2.5. List of comport links available for booting.

Example :

```
# NODE PORT NODE PORT
#-----------------------
BOOTLINK NODE1 1 NODE2 4
BOOTLINK NODE2 1 NODE3 4
```

BOOTLINK : keyword for bootlink definition

NODE : symbolic name of node

PORT : comport number

You don’t have to specify all available comport links, a minimal set that connects all nodes is all that is required. The hostprogram will find out the shortest path to boot each node.

12.3. Host server interface

The hostserver is accessed from the target using a low level driver. This driver establishes a protocol between the server and the target. The target always takes the initiative. This is done by writing to a memory location called TXDATA using the call_server() function. The reply from the server can be read in the RXDATA memory location. Access to the host as well as to the memory locations must be protected by locking on the HOSTRES resource.

The host server functionality can easily be extended by the user. See the \user subdirectory. You will need to recompile the host server.

12.4. Host interface low level driver

These functions permit a direct access to the host server. They are intended for the kernel developer and are only available on the root processor.

`void server_echo (unsigned c);`

As many entries as indicated by the parameter of the function are transferred from the array TXDATA to the host server. The host server will then return a copy of that packet to the array RXDATA.

`void server_exit (int c);`
`#define server_terminate ( )server_exit (0);`

This function will terminate the host server. The value of its parameter is used as a return value for the server program.

This macro will terminate the host server and return 0 (zero) to the environment of the server program.

`int server_getarg(int argnum, char *buffer, int buflen);`

It can be used for reading arguments following the network filename on the host program commandline. The function returns nonzero if the argument is present, and zero otherwise. Argument 0 is the *.NLI filename.

`int server_getenv (int *srce, char *dest, int len);`

It can be used for reading arguments following the network filename on the host program commandline. The function returns nonzero if the argument is present, and zero otherwise. Argument 0 is the *.NLI filename.

```
void call_server (void);
```

This function transfer the contents of the TXDATA memory area to the hostserver and returns when the reply has been written in RXDATA.

```
void server_system (char *comm);
```

This function will execute the command pointed to by the parameter (in text format) in the environment of the host program and return the value of that system call.

```
void server_putchar (int c);
```

This function requests the host server to put a character on the host server screen.

```
void server_putstring (char *s);
```

This function requests the host server to put the string that is pointed to by the parameter, on the host server screen.

```
int server_pollesc (void);
int server_pollkey (void);
```

These functions poll the host server for a keyboard input, the escape key (used for starting the debugger), resp.

```
UNS32 server_time (void);
```

This function requests the environment of the host program, the calendar time and returns that value.

12.5. Higher level drivers

Three drivers are provided that communicate with the hostserver using the call_server() function of the low level driver. Thanks to the use of system wide queues they permit any task to access the hostserver, even if is located on a processor node that is not connected to the host. These drivers are Virtuoso tasks. They must be defined in the SYSDEF file together with the used queues and resources.

Driver	Use	Input queue	Outputqueue	Resource
CONODRV	Console_out		CONOQ	CONRES
CONIDRV	Console_in	CONIQ		CONRES
STDIODRV	Std I/O	STDIQ	STDOQ	STDIORES
GRAPHDRV	Graphics	GRAPHIQ	GRAPHOQ	GRAPHRES

Note : The Borland PC version simulates the hostserver using a hostserver task. The interface also uses two semaphores to synchronize the communication. See the section on the 80X86 version for more details.

12.5.0.a. Console input and output

The console drivers provide a character based input and output with the hostserver. The following functions are provided :

`KS_EnqueueW(CONOQ, char, sizeof(char) )`

Output a character. Can be used from a remote processor.

`KS_DequeueW(CONIQ, char, sizeof(char) )`

Input a character. Can be used from a remote processor.

`printl(char *string, K_RES CONRES, K_QUEUE CONOQ)`

Output a string ofcharacters.

For formatting the string use the sprintf() function from the stdio library.

These character based functions are used by the Virtuoso Task Level debugger when used with a terminal connection. While its operation is slower than using std I/O, it has the advantage that the full stdio.lib is not needed. This reduces memory requirements as well as permits an easy port when that target is only accessible using a terminal.

12.5.0.b. Standard I/O driver

This driver provides standard I/O. They should not be used with the console drivers as first characters might be lost. The following section details the available functions.

12.5.0.c. Graphics driver

This driver implements an emulation of Borland DOS graphics calls. The following section lists the available calls.

13. Runtime libraries

13.1. Standard I/O functions

13.1.1. Implementation limits

These functions provide the same functionality as the equivalent ones provided with the runtime library of your compiler. Note however that they were adapted to work in conjunction with the multitasking and distributed environment of Virtuoso and use the host server program. See the manual of the compiler for a description.

```
#define BUFSIZ = 512 /* Default buffer size use by
        “setbuf” function */
#define MAXBLKSIZE 0x4000 /* internal use */
#define MAXPACKETLEN 256 /* internal use */
#define MAXFILEPTRS 32 /* internal use */
#define MAXSTRINGLEN 240 /* maximum string length */
#define MAXPATHLEN 220 /* maximum path length */
#define MAXFMODELEN 20 /* internal use */
```

These limitations find their origin in the communication protocol. The communication protocol uses 64 words. The first word is the header, followed by a command and/or the data.

13.1.2. Standard I/O functions

/* _stdio.h - standard input/output functions */

```
FILE *fopen (
     const char *path,
     const char *mode);
FILE *freopen (
     const char *path,
     const char *mode,
     FILE *stream);
int fclose (
     FILE *stream);
int fgetc (
     FILE *stream);
```



```
int fputc (
     int c,
     FILE *stream);
int ungetc (
     int c,
     FILE *stream);
char *fgets (
     const char *string,
     int n,
     FILE *stream);
int fputs (
     const char *string,
     FILE *stream);
char *gets (
     char *string);
int puts (
     const char *string);
size_t fread (
     void *ptr,
     size_t size,
     size_t nmembs,
     FILE *stream);
size_t fwrite (
     const void *ptr,
     size_t size,
     size_t nmembs,
     FILE *stream);
int fgetpos (
     FILE *stream,
     fpos_t *pos);
int fsetpos (
     FILE *stream,
     const fpos_t *pos);
int feof (
     FILE *stream);
int ferror (
     FILE *stream);
int fflush (
     FILE *stream);
```

```
int fseek (
     FILE *stream,
     long int offset,
     int origin);
long ftell (
     FILE *stream);
void setvbuf (
     FILE *stream,
     char *buf, int mode,
     size_t size);
int rename (const char *old,
     const char *new);
int unlink (const char *name);
int fileno (FIE *stream);
fstat (int filedesc,
     struct stat *info);
stat (const char *name,
     struct stat *info);
#define getc fgetc
#define putc fputc
#define getchar() getc (stdin)
#define putchar(c) fputc (c,stdout)
#define rewind(f) fseek (f,0,SEEK_SET)
#define setbuf(f,b) setvbuf (f,b,_IOFBF,BUFSIZ)
#define remove(f) unlink(f);
int fprintf (
     FILE *F,
     const char *format, ...);
int vfprintf (
     FILE *F,
     const char *format,
     va_list vargs);
int printf (
     const char *format, ...);
     int vprintf (
     const char *format,
     va_list vargs);
int sprintf (
     char *s,
     const char *format, ...);
```



```
int vsprintf (
     char *s,
     const char *format,
     va_list vargs);
```

13.2. PC graphics I/O

13.2.1. Overview

From version 1.2 on, an optional graphical server is included. The distributed graphics server introduced with version 1.2 enables you to perform graphics operations on the PC screen from any task on any processor in the system.

The interface is not designed for heavy-duty graphics work but will be useful in many applications, e.g. to provide an attractive ‘human interface’.

The graphics server works as follows:

1. Application tasks use the function calls defined in GRINT.C. These functions format your data into command packets that are send to a graphics driver task.
2. Two queues (GRAPHOQ and GRAPHIQ) establish the system wide connection between the application tasks and the driver task.
3. The driver task, GRAPHDRV, executes on the root processor and communicates with the host PC server program. GRAPHDRV should run at fairly high priority, normally just below the console drivers.
4. The PC server program has been recompiled with the Borland C++ compiler, and is extended to interface to the Borland BGI drivers. These drivers do the actual plotting on the screen. The extended server is fully compatible with the standard version, and you can use it to run the compiler tools as well.

The two queues must be protected by a resource named GRAPHRES. This is necessary for two distinct reasons:

1. The queues are 4 bytes wide, and a graphic command packet can take any number of words. While a packet is being sent to the driver, other tasks must be blocked from interfering.
2. Tasks should not be allowed to modify the graphics context (current color, position, textsize, etc.) while another task is using the graphics server. For this reason, a lock and unlock call built into every function would not be sufficient.

To use the graphics calls, you should:

1. Lock GRAPHRES
2. Restore your graphics context (if necessary)
3. Perform the plotting actions
4. Save the graphics context (if necessary)
5. Unlock GRAPHRES

The original Borland functions used to read the current context are somewhat impractical if frequent saving and restoring is required. For this reason, these calls have been modified to use a more symmetric syntax. Most of these now take the same predefined struct as a parameter for both reading or writing a selected part of the context. For some other calls, the syntax has been modified for technical reasons, e.g. it doesn't make sense to have pointers into to the PC memory space on a transputer.

Most graphics functions are of void type and will not even wait for anything to return on the GRAHIQ queue. This means that graphics actions are not necessarily executed when the interface functions return. To wait until execution has finished (e.g. after a change to text mode, and before using the console driver), use a value returning call such as graphresult (). This will deschedule the calling task until the result is available.

The best way to explore the graphics server is to use one of the demonstration programs as a starting point and play with it. Much practical information can be found in GRINT.H. The next section gives only a short description of all functions, classified by type. For detailed information you should consult the documentation supplied with the Borland compiler.

13.2.2. Driver and mode selection

```
void detectgraph (
     int *driver,
     int *mode);
```

Tests the graphics hardware and returns the correct driver number for the installed graphics adapter, and the highest mode number available from this driver. The returned driver value will be zero, if the BGI library is unable to use the hardware.

```
void initgraph (
     int driver,
     int mode);
```

Switches to graphics display using the selected driver and mode. The driver parameter should be the value returned by detectgraph (), or the predefined constant DETECT (see GRINT.H).In the latter case,

the graphics library will use detectgraph() to find out about the hardware. The mode number can be any mode supported by that driver. This call clears the display and presets all graphics parameters to their default values.

`void setgraphmode (int mode);`

Switches to the selected display mode using the current driver. This can only be used if the graphics system is already initialized, i.e. after a call to initgraph ().

`void restorecrtmode (void);`

Switches back to the screen mode, that was active before the graphics system was initialized. This will normally be a text mode. The current driver remains installed, and you can return to a graphics display using setgraphmode ().

`void closegraph (void);`

Closes the current graphics driver. To use graphics again, a call to initgraph () must be used.

`void setactivepage (int page);`

If the graphics adapter supports multiple pages, this call can be used to select the page used by the plotting routines, i.e. the page that is written to.

`void setvisualpage (int page);`

If the graphics adapter supports multiple pages, this call can be used to select the page used by the video display circuits, i.e. the currently displayed page.

`void graphdefaults (void);`

sets all graphics settings to their defaults.

The next two calls exist mainly for the benefit of the debugger task, which has to be able to switch to text mode at any moment. If correctly used, as described below, these calls can be 'nested' and used in application code as well.

`int savescreen (void);`

Switches to text mode after having saved the current graphics screen and context in DOS memory. This calls returns non zero only if it actually did save the graphics screen and context. In this case, it should be matched afterwards by a call to restscreen ().A zero return value indicates that the call was not necessary (already in text mode), or that not enough DOS memory was available.

```
int graphresult (void);
```

Returns an error code for the last unsuccesful graphics operation.

13.2.3. Read or write graphics parameters and context

```
void getmodepars (struct modepars *mpars);
```

Obtains parameters describing the current graphics mode. This can be used to make a program adapt itself to the available hardware, and so be more 'portable'. See GRINT.H for details on the values returned in the modepars structure. The pixel aspect ratio parameters describes the relative visual size in X and Y of a pixel. This can be used to obtain the same scaling in X and Y.

```
void setviewport (struct viewport *vport);
```

Sets the viewport for the following plot actions. A 'viewport' is a rectangular part of the screen. The origin (the point referenced by coordinates (0, 0)) is set at the upper left corner of this rectangle. If the 'clip' field is not zero, all following plot actions are limited to the area determined by the viewport.

```
void getviewport (struct viewport *vport);
```

Reads the current viewport and 'clip' option.

```
void setallpalette (struct palette *pal);
```

Sets the current palette. The palette determines the mapping of colornumbers to visual colors. The coding depends heavily on the selected driver and mode. For details, see GRINT.H and documentation supplied with your hardware.

```
void getallpalette (struct palette *pal);
```

Reads the current palette.

```
void setpalette (
     int colornum,
     int color);
```

Sets the current color mapping for one color only.

```
void setrgbpalette (
     int color,
     int r,
     int g,
     int b);
```

As setpalette, but for the IBM 8514 adapter (driver 6) only.


```
void setfillstyle (struct fillstyle *fstyle);
```

The 'fillstyle' determines how filled object are plotted. You can select the color to be used, a number of standard patterns, or set a user fill pattern. The user fill pattern is an 8 by 8 pixel pattern, represented as an 8 character array. See GRINT.H for parameter details.

```
void getfillstyle (struct fillstyle *fstyle);
```

Reads the current fillstyle.

```
void getuserpars (struct userpars *upars);
```

Reads the current user parameters. See the USERPARS typedef for details. There is no corresponding setuserpars () - these values can be set by the five next calls documented below.

```
void setcolor (int color);
```

Sets the current line and text color. The solid filling color is set by the setfillstyle function.

```
void setbkcolor (int color);
```

Sets the current background color.

```
void setlinestyle (
     int style,
     int patt,
     int thick);
```

Sets the line drawing parameters. You can select the line thickness, a number of standard patterns, or a user-defined pattern.

```
void settextstyle (
     int font,
     int direct,
     int size);
```

Sets the text plotting parameters. There are five standard fonts supplied with BGI drivers. Font 0 is an 8 by 8 pixel bitmap font. The others are 'line' fonts. You can also specify the direction of the text and its size.

```
void settextjustify (
     int horiz,
     int vert);
```

Selects text justifications options for the outtext() function.

```
void setwritemode (int mode);
```

Sets writemode for plotting operations. This can be 'overwrite' (mode = 0) or 'exclusive or' (mode = 1).

```
void getcurrcoords (struct point *xy);
```

Reads the current graphics position.

13.2.4. Drawing pixels and lines

```
void putpixel (
     int x,
     int y,
     int color);
int getpixel (
     int x,
     int y);
```

Plots or read pixel.

```
void moveto (
     int x,
     int y);
void moverel (
     int dx,
     int dy);
```

Moves to absolute or relative position.

```
void lineto (
     int x,
     int y);
void linerel (
     int dx,
     int dy);
void line (
     int x1,
     int y1,
     int x2,
     int y2);
void rectangle (
     int left,
     int top,
     int right,
     int bot);
```



```
void circle (
     int xc,
     int yc,
     int r);
void arc (
     int xc,
     int yc,
     int a0,
     int a1,
     int r);
void ellipse (
     int xc,
     int yc,
     int a0,
     int a1,
     int xr,
     int yr);
```

Draws full lines, rectangle, circle, circular arc, or elliptical arc. using current color and linestyle.

Notes:

1. xc, yc = centre of circle or ellipse, a0, a1 = start and end angle of arc,
2. r, xr, yr = radii.

The driver performs any necessary aspect ratio corrections, so a circle will always be a real circle and not an ellipse, even with non-square pixels. The next function can be used to read the x, y values calculated by the driver:

```
void drawpoly (
     int npoints,
     int *points);
```

Draws polygon. The second parameter is an array of integers, with alternating x and y coordinates. To obtain a closed polygon, repeat the first point at the end. This is much more faster than repeated line drawing calls.

13.2.5. Drawing filled forms

```
void pieslice (
     int xc,
     int yc,
     int a0,
     int a1,
     int r);
void sector (
     int xc,
     int yc,
     int a0,
     int a1,
     int xr,
     int yr);
void fillellipse (
     int xc,
     int yc,
     int xr,
     int yr);
void bar (
     int left,
     int top,
     int right,
     int bottom);
void bar3d (
     int left,
     int top,
     int right,
     int bottom,
     int depth,
     int topflag);
void fillpoly (
     int n,
     int *points);
void floodfill (
     int x,
     int y,
     int border);
```

Draws filled shapes using the current fill color or pattern.

13.2.6. Text plotting

```
int installuserfont(char *name);
```

Adds a user-supplied font to the list of five standard fonts known by the driver.

```
void setusercharsize (
     int multx,
     int divx,
     int multy,
     int divy);
```

This function can be used for fine scaling of a ‘line font’. It has no effect on bitmapped fonts.

```
void textdimensions (
     char *text,
     int *x,
     int *y);
```

Requests pixel dimensions of text.

```
void outtext (char *text);
void outtextxy (
     int x,
     int y,
     char *text);
```

Outputs a text at current position or at x, y.

13.2.7. Other graphical calls

The next three calls can be used to implement a simple ‘icon’ system.

```
int getimage(
     int left,
     int top,
     int right,
     int bottom);
```

Saves part of the screen to DOS memory. This function allocates a block of memory on the host PC and saves part of the graphics screen to this area. Returns 0 if the allocation failed, a block number otherwise. The size of a block is limited to 64K minus a few bytes.

```
void putimage(
     int left,
     int top,
     int block,
     int op);
```

Puts a previously saved block on the screen at left, top, using a selected logical operation (see GRINT.H for details). The block remains in DOS memory and can be replotted any number of times.

```
void freeimage(int block);
```

Forgets saved blocks and releases the DOS memory they use.

```
void cleardevice (void);
void clearviewport (void);
```

These calls are supposed to fill the entire screen or the current viewport with the current background color. There seems to be some problems (at least in the EGA-VGA driver) with these calls, and the best thing is to avoid using them. An alternative way to clear (part of) the screen is to plot a solid rectangle (bar).

```
void getarccoords (struct arccoords *arc);
```

Returns the actual start and end coordinates used for plotting circle or ellipse based objects (e.g. arc, pieslice, etc.).

14. System Configuration

14.1. System configuration concepts

Central to the concept of Virtuoso is the use of a system generation tool that provides two basic functions to the designer. Firstly, it provides a means to change the system configuration without the need to change the application program, by regenerating all the necessary system files automatically. Secondly, it generates all necessary tables and code to initialize the application.

The system generation tool enables the user to write topology independent code, so that when processors are added or removed and kernel objects (such as tasks, queues, etc.) are moved or their attributes changed, the application source code does not need modification.

To define a Virtuoso system configuration a system definition file is used. In this file, following a simple syntax, the user should describe his application in terms of topology and kernel objects with their attributes. When done, Sysgen is invoked to parse the system definition file and to generate all necessary include files.

14.1.1. Kernel objects

The concept of the Virtuoso kernel is those of objects that are used by the programmer to achieve the desired result. Each object is of a predefined type and the application task can invoke the kernel services to operate on these objects. During the system definition phase, the user must supply the names and attributes of each object. This will automatically create the kernel objects as datatypes in C syntax in the include files. The predefined kernel objects are listed and explained below. The names of the kernel object relate with the predefined kernel datatypes.

NODE	Virtuoso network node where an instance of the kernel is running
NETLINK	defines a communication channel between two nodes.
DRIVER	defines a driver function.
TASK	keyword for a Virtuoso task.
RES	keyword for a Virtuoso resource.
MAP	keyword for a Virtuoso memory map.
MAILBOX	keyword for a Virtuoso message mailbox.
QUEUE	keyword for a Virtuoso message queue.
SEMA	keyword for a Virtuoso semaphore.

Note that the K_TICKS are defined in the mainx.c file by the user.

14.2. Sysdef : system definition file format

The system definition file can be made with any text editor.

Many of the kernel objects are known to the kernel by an identifier. This is generally an integer, e.g. a 32 bit word. The higher bits represent the node identifier, while the lower bits represent the object identifier. The identifiers are automatically assigned by Sysgen and must be unique in the system. Therefore, when adding or deleting an object, make sure that Sysgen is invoked so that all include files are regenerated.

Sysgen will require the entry of names for the various system objects. All names follow a standard naming convention. A name is a maximum of 10 characters (if you need more, Sysgen can be adapted), the first of which must be alphabetic. No embedded spaces are allowed while upper and lower case can be used. A name must be unique. All attributes of an object are numbers. Note that the system definition file syntax follows some conventions of C. As a result, you can define numbers as symbolic names, use include files, and use comments. Entries must be separated by one or more spaces and must start with a keyword followed by the symbolic name and the attributes. Note that some definitions and configuration data must be specified in the main#.c file.

Note that on single processor versions or versions that support only one type of processor, some of the entries need not to be defined as these are known by default. As such there are no node and link definitions and the node identifier is not present as an attribute of the objects. As an example we provide here a possible system definition file.

```
/* Example System Definition File */

#define BYTE 1
#define WORD 4
NODE ROOT T8
NODE NODE2 T8
NODE NODE3 C40
NODE NODE4 C40

NETLINK
ROOT ‘NetLink_Driver (1)’ , NODE2 ‘NetLink_Driver (3)’
```



```
NETLINK
ROOT ‘Netlink_Driver(2)’ > NODE3 ‘NetLinkDma (3,1)’
NETLINK
ROOT ‘Netlink_Driver(2)’ < NODE3 ‘NetLinkDma (0,1)’
NETLINK
NODE3 ‘NetLinkDma(1,1)’ , NODE4 ‘NetLinkDma (4,1)’
/*
DRIVER ROOT ‘HL03_Driver ()’
*/
DRIVER ROOT ‘HostLinkDma (0, 3, PRIO_ALT)’
DRIVER NODE1 ‘RawLinkDma (2, PRIO_DMA)’
DRIVER NODE2 ‘RawLinkDma (5, PRIO_DMA)’

DRIVER ROOT ‘Timer0_Driver (tickunit)’
DRIVER NODE2 ‘Timer0_Driver (tickunit)’

/* taskname node prio entry stack groups */
/* ------------------------------------------------ */

#ifdef DEBUG
TASK TLDEBUG ROOT 1 tldebug 400 [SYS EXE]
TASK POLLESC ROOT 1 pollesc 128 [EXE]
#endif
TASK CONIDRV ROOT 2 conidrv 128 [EXE]
TASK CONODRV ROOT 3 conodrv 128 [EXE]
TASK GRAPHDRV ROOT 4 graphdrv 128 [EXE]
TASK WLMON1 ROOT 10 wlmon 256 [WLM]
TASK WLMON2 NODE2 10 wlmon 256 [WLM]
TASK MASTER ROOT 5 master 256 [EXE]

TASK WLGEN ROOT 20 wlgen 256 [EXE]

TASK DIGIT11 ROOT 15 digit11 256 [DIG]
TASK DIGIT12 NODE2 14 digit12 256 [DIG]
TASK DIGIT13 NODE3 13 digit13 256 [DIG]
TASK DIGIT14 NODE2 12 digit14 256 [DIG]
TASK DIGIT15 ROOT 11 digit15 256 [DIG]
TASK DIGIT41 NODE2 15 digit41 256 [DIG]
TASK DIGIT42 NODE3 14 digit42 256 [DIG]
TASK DIGIT43 NODE2 13 digit43 256 [DIG]
```

```
TASK DIGIT44 ROOT 12 digit44 256 [DIG]
TASK DIGIT45 NODE4 11 digit45 256 [DIG]

/* queue node depth width */
/* ---------------------------------- */
#ifdef DEBUG
QUEUE DEBUGIN ROOT 16 4
#endif
QUEUE CONIQ ROOT 16 1
QUEUE CONOQ ROOT 256 1
QUEUE GRAPHIQ ROOT 16 4

/* map node blocks blsize */
/* ---------------------------------- */
MAP MAP1 ROOT 4 1K

/* mailbox node */
/* -------------------------- */
MAILBOX MAILB1 ROOT

/* sema node */
/* ------------------------ */
SEMA CTICK ROOT

SEMA SM11 ROOT
SEMA SM12 NODE2
SEMA SM13 NODE3
SEMA SM14 NODE4
SEMA SM15 ROOT
SEMA SM16 NODE1

/* resource node */
/* ------------------------ */
RESOURCE HOSTRES ROOT
RESOURCE CONRES ROOT
RESOURCE GRAPHRES ROOT
```

14.2.1. Description requirements for the kernel object types

The relative order of the definitions is not important, except that all NODEs

should be defined before anything else.

14.2.2. Node description

Virtuoso nodes are composed of a processor with its local memory and communications ports. They are identified by a symbolic name followed by their type if the distinction is relevant.

The syntax of a NODE definition is :

- syntax . . . . . NODE <name> <type>
- name. . . . . The symbolic name
- type. . . . . A processor type. e.g. C40 for TMS320C40.

This information is not actually used in this version. More parameters may be used in future versions, e.g. the number of task priorities (currently fixed at 64).

- example . . . . . .

```
NODE ROOT T8
NODE NODE2 T8
NODE NODE3 C40
NODE NODE4 C40
```

This describes 4 nodes, of which two are TMS320C40 DSPs and two are T800 transputers.

14.3. Driver description

Drivers are defined with their exact C syntax and parameters as they are intended to be used in the application.

The syntax of a DRIVER definitions is :

- syntax . . . . . DRIVER <node> '<driver(param1, param2, ...)>'
- node . . . . . The symbolic name for the node.
- driver. . . . . The C function call for the driver. Parameters must be correct for the application at hand. The default drivers are declared in iface.h.

• example . . . . . .

```
DRIVER ROOT ‘HostLinkDma (0, 3, PRIO_ALT)’
DRIVER NODE1 ‘RawLinkDma (2, PRIO_DMA)’
DRIVER NODE2 ‘RawLinkDma (5, PRIO_DMA)’
DRIVER NODE1 ‘Timer1_Driver (tickunit)’
```

14.3.1. Link descriptions

Each Virtuoso node is connected with other Virtuoso nodes through links. A link is a communication channel composed of a physical carrier between two ports. Each port is processor dependent and requires a specific driver to be installed on the port.

Two types of links are distinguished :

1. Netlinks : exclusively used by the Virtuoso system, and only indirectly by the user upon issuing a kernel service. A Virtuoso specific kernel protocol is used. The netlink connections are described as pairs of ports owned by the nodes the driver resides on. SYSGEN.EXE will find all shortest routes from any node to any other, and compile the routing tables. An error will be reported if the network is not fully connected
2. Rawlinks : exclusively used by the application tasks for direct communication with a task on a neighboring processor or an I/O device. The datatransfer is a raw bit protocol. The raw links are only indicated by their driver.

Note : the physical carrier for any type of link can be any type of hardware (e.g. twisted wire, cable, bus connector, common memory). The only requirement is a correctly working driver.

The syntax of a NETLINK definitions is :

• syntax . . . . . NETLINK

```
<node1> ‘<driver1>’ <connector> <node2> ‘<driver2>’
```

• node1, node2 . . . . The two connected nodes

• driver1, driver2. . . . The drivers used on the respective nodes

• connector . . . . . The <connector> can be :

, : bidirectional link

> : unidirectional link (from left to right)

< : unidirectional link (from right to left)

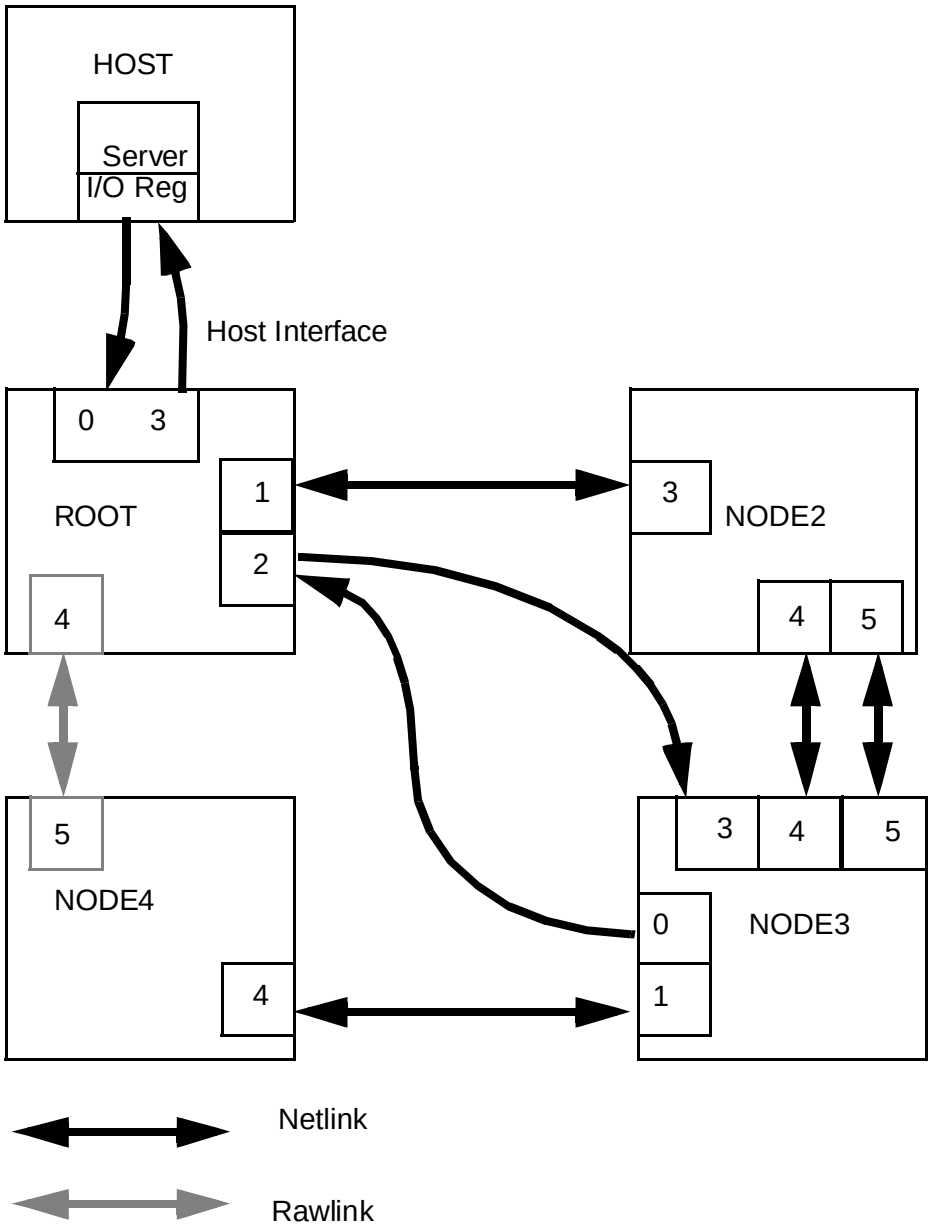
Note that the driver description between quotes must be syntactically correct as it is copied literally into the nodex.c file by sysgen. The separator symbol is defined as follows :

• example . . . . . .

```
NETLINK
ROOT ‘NetLink_Driver(1)’ , NODE2 ‘NetLink_Driver (3)’
/* bidirectional link between ROOT and NODE2 */
NETLINK
ROOT ‘Netlink_Driver(2)’ > NODE3 ‘NetLinkDma
        (3,PRIO_DMA)’
/* Unidirectional link between ROOT and NODE3 */
NETLINK
ROOT ‘Netlink_Driver(2)’ < NODE3 ‘NetLinkDma
        (0,PRIO_DMA)’
NETLINK
NODE3 ‘NetLinkDma(1,PRIO_ALT)’, NODE4
        ‘NetLinkDma(4,PRIO_ALT)’

DRIVER ROOT ‘RawLinkDma (4, PRIO_DMA)’
DRIVER NODE4 ‘RawLinkDma (5, PRIO_DMA)’
```

This describes the following network :



14.3.2. The routing tables

On each processor, the kernel uses a local routing table to know how to reach other processors. This means that there is no master processor in the system.

Virtuoso uses a local routing schema. Basically it is a table indexed by the

Processor Number, where each entry presents the local driver that can be used to forward further the packet to the target node. Parallel routes with the same minimal length are included. Below is the routing table for NODE3 of the previous example. The routing tables are part of the generated nodex.c file and should not be modified by the user.

```
static int R0001 [] = { 0, -1 };
static int R0002 [] = { 4, 5, -1 };
static int R0003 [] = { 1, -1 };

int *RouteInd [] =
{
R0001,
R0002,
NULL,
R0003
};
```

Obviously, some ports are not part of the routing network. It could be connected to a peripheral device or simply be left unconnected.

Note that unless there are specific reasons (such as to optimize the network traffic to suit better the application), the routing tables should never be edited.

14.4. Task definitions

- syntax . . . . . TASK <name> <node_name> <priority> <entry_point> <taskgroups>
- name . . . . . The name used for reference by the application source code for explicit reference to the task in the Virtuoso function calls. It is also optionally used by the debugging task.
- node_name . . . . . The network node the task is residing on.
- priority . . . . . The priority of the task at boot time. The task's priority is a number ranging from 1 (highest priority) to 64 or 255 (lowest priority). Equal priorities may be specified.
- entry _point . . . . . The task entry point is the name of the "main()" routine for the task. It follows normal C conventions for function names.
- stack Size . . . . . The task stack size in bytes may be defined. The default size is 512 bytes. Note that an some word oriented 32 bit processors, the stack increments in

words with 1 word being 4 bytes.

• taskgroups (optional) A task can be part of a taskgroup. The following groups are predefined:

SYS: this includes all the tasks that are not blocked when the task level debugger is invoked (normally only the debugger task).

EXE: this includes all the tasks that must start up at system initialization.

FPU: this includes all the tasks that use the extended context, e.g. tasks that use the Floating Point with CPU context on every task switch. Real-time systems, typically have only a small number of the tasks in the entire suite of tasks which require floating point support. Context switch time is minimized by performing the FPU context swap only on demand.

• example . . . . . TASK TLDEBUG ROOT 1 tldebug 400 [SYS EXE]

14.5. Semaphore definitions

• syntax . . . . . SEMA <name> <node>

• name . . . . . The name of the semaphore according to the standard name convention.

• node . . . . . This is the network node the semaphore is placed on.

• example . . . . . SEMA ALARM P345

14.6. Resource definitions

• syntax . . . . . RES <name> <node>

• name . . . . . The name of the Resource according to the standard name convention.

• node . . . . . This is the network node that manages the access to the resource. E.g. when the resource is the console, the node is the one on which the drivers are placed for accessing the console.

• example . . . . . RES HOSTRES ROOT

14.7. Queue definitions

• syntax . . . . . QUEUE <name> <node> <width> <depth>:

• name . . . . . The name of the Queue according to the standard name convention.

• width . . . . . The width of the queue is defined as the number of words in a single entry.

The number must be nonzero.

- depth . . . . . The depth field defines the number of entries in the queue. The depth must be nonzero. The memory space required for the queue body is the product of the width times the depth.

- example . . . . . QUEUE CONOQ ROOT 4 1

14.8. Mailbox definitions

- syntax . . . . . MAILBOX <name> <node>

- name . . . . . The name of the Mailbox according to the standard name convention.

- node . . . . . This is the network node the mailbox is placed on.

- example . . . . . MAILBOX slave 3

14.9. Memory map definitions

- syntax . . . . . MAP <name> <#blocks> <size> <node>

- node . . . . . This is the network node the memory map is placed on.

- name . . . . . The name of the Memory Partition according to the standard name convention.

- #blocks . . . . . The number of blocks in each map. The count must be nonzero. The memory space required for the map is the product of count and size fields.

- size . . . . . The size field defines the size of a memory block in bytes. The minimum block size is 4 bytes.

- example . . . . . MAP ADBuffer 6 16 ROOT

14.10. Note on the size parameters

As Virtuoso supports processors that allow byte aligned addressing as well as processors that only allow word aligned addressing, there is a potential problem. This is due to the fact that the sizeof() function always gives the size in as a difference between consecutive addresses. Hence on e.g. a DSP like the 96K a char is represented as a 32bit word with the most left 8bit representing the char (from 0 to 255) and this has a size of 1. While this is generally not a problem when working with integers, it can be a problem when transferring between two different types of processors.

14.11. Other system information and system initialization

At the moment of writing, some of the necessary system information has to be defined in the mainx.c file.

This mainx.c file is the actual program that is executed. It starts by initializing the kernel, the drivers and the ISR system. Some of the global system variables and constants are defined here as well. These depend on the target processor as well as on the application. For example:

```
/*
 * Internal timer clock frequency of 8.333 MHz
 */
#define CLCKFREQ 8333000

/*
 * K_TICK period in microseconds
 */
#define TICKTIME 1000

/*
 * K_TICK frequency in Hertz
 */
#define TICKFREQ 1000

/*
 * K_TICK in CLCKFREQ units
 */
#define TICKUNIT 8333

/*
 * number of timers
 * You need at least one timer for each timer
 * that can allocated at the same time plus
 * one timer for each task that may call a WT
 * service at the same time
 */
#define NTIMERS 20

/*
 * number of command & mail packets.
 * Each packet is 52 bytes long
 */
#define NCPACKS 20
```



```
/*
 * the size of data packets in BYTES.
 * This MUST be the same value on all nodes
 */
#define DATALEN 1024

/* data packets are used to buffer raw and message data
 * passing through a node
 * (not at the endpoints - if you have less than three
 * nodes, you don't need data packets).
 * 4 buffers for each concurrent data transmission will
 * usually be enough.
 */
#define NDPACKS 5

/*
 * number of multiple wait packets
 */
#define NWPACKS 20
```

See part 3 (the Virtuoso Binding Manual) for more details.

15. Debugging environment under Virtuoso

15.1. Task level debugger concepts

The task level debugger provides snapshots of the Virtuoso object states. It can be called from within the program or from the keyboard. The debugger operates as a task and is usually set up as the task with the highest priority on the root node. On other nodes in the system, specific peek-poke functions are linked in with the Virtuoso kernel so the debugger task on the root can exam the object states on any node in the system. Whenever The debugger runs, it freezes the rest of the system thereby permitting coherent views of the Virtuoso kernel objects. The debugger is not intended as a replacement for other debugging tools but is meant to assist the user in tuning the performance or checking out problems within the Virtuoso environment.

The debugger outputs on the screen of the PC or workstation that acts as the host of the target board. Some versions also work with a simple character terminal. Because the debugger usually operates as the highest priority task in the system, all other tasks are suspended. Interrupts are serviced as usual while the debugger is active, but the system TICKS timer will ignore its interrupt, so that it is effectively halted. Therefore, time spent in the debugger is invisible at the task level.

Tightly integrated with the task level debugger is the tracing monitor. This lists an execution trace of the events that were seen at the microkernel level (and sometimes at the ISR level).

The task level debugger and tracing monitor can be removed by relinking the application without the debugger and tracing specific parts. Disabling the DEBUG switch in the makefile will automatically regenerate a new application without the debugger and tracing monitor. Make sure that you follow the outline for the makefile and sysdef file of the examples provided

15.2. Entry into the debugger

15.2.1. Invoking the debugger from the keyboard

The POLLESC task driver will intercept the ESCAPE key, and start the debugger when this key is pressed. Note that this happens only 1O times per second and that ongoing activity that uses the host resource (like stdio) will be finished first before the debugger can get access to the host and starts up.

Once the debugger is entered, the version of the debugger is displayed as :

```
VIRTUOSO TL Debugger 3.09 NODE 1 NODE1
Started by user interrupt
```

From the command prompt, the user may enter any of the primary debugger commands which appear in the following paragraph. All commands must be terminated by an Enter key. The question mark “?” displays the list of available commands.

In each command, the user can scroll up and down using the “+” and “-” keys.

15.2.2. Invoking the debugger from within your program

The task level debugger task can be started by enqueueing the local NodeId in the DEBUGIN queue. The advantage of this method is the debugger is started almost immediately,

Example :

```
K_Node k;
k = KS_NodeId;
KS_Enqueue(DEBUGIN, &k, 4);
```

Note :

Wirh older versions of Virtuoso, semaphores were used to start up the task level debugger. Refer the previous version of the manual if you still have an older version.

15.2.3. Differences at system generation time

In order to use the debugger, you must create a task (normally called TLDEBUG) on the root processor and give it the highest priority in the system. The entry point for the debugger task is a Virtuoso library function named tldebug. Next there is a POLLESC task that scans the keyboard fro the ESCAPE key. You also need to define a queue (normally called DEBUGIN) for the debugger as well. Finally your application must be linked with a version of the Virtuoso library that includes the debug code.

The example programs provided with the distribution show how this can be organized in such a way that a simple modification of the makefile is all you need to build a system with or without debugger.

Note that on older versions of Virtuoso (v.3.0) a slightly different mechanism

was used. This used a semaphore instead of a queue to start up the debugger. Refer to the examples or the read.me files.

15.2.4. Debugger commands

Entering a response of “H” (or “h”) followed by a carriage return to the TLDEBUG prompt causes the debugger command menu to be displayed. It appears as:

```
VIRTUOSO TL Debugger 3.09 NODE 1 NODE1
Started by user interrupt
-----------------------------------------------
A - Allocation of memory
C - Clock & Timers
K - Stack use
L - Monitor listing
M - Mailboxes
N - Network info
O - Other node
Q - Queues
R - Resources
S - Semaphores
T - Task status
X - Exit debugger
Z - Reset Counts
$ - Task Manager Mode
? - This help screen
```

15.2.5. Tasks

Selection of this option produces a snapshot of the state of all the tasks in the system as shown below. The snapshot contains three columns of information:

Name	the task’s symbolic name
Prio	the current task priority
State	current task state

Example :

```
 VIRTUOSO TL Debugger 3.09 NODE 1 NODE1 TASK STATES
 # Name  PrioState
-------------------------------------------------------
 0 TLDEBUG 11 +
 1 POLLESC   1- Waiting for Timer
 2 CONIDRV   2- Waiting for Resource
 3 CONODRV   3- Waiting for Dequeue
 4 GRAPHDRV 4 -
 5 WLMON1   10- Waiting for Resource
 6 MASTER    5- Waiting for Sema
 7 WLGEN    20- Waiting for Dequeue
 8 DIGIT11  15- Waiting for Resource
 9 DIGIT13  13- Waiting for Sema
10 DIGIT14  12- Waiting for Sema
11 DIGIT15  11- Waiting for Sema
12 DIGIT22  14- Waiting for Sema
13 DIGIT24  12- Waiting for Sema
14 DIGIT31  15- Waiting for Resource
15 DIGIT33  13- Waiting for Sema
16 DIGIT35  11- Waiting for Sema
17 DIGIT42  14- Waiting for Sema
```

The legitimate state descriptions are:

-	Task suspended by debugger
+	Task active
Inactive	Task not started
Terminated	Task terminated
Suspended	Task suspended
Waiting for Timer	Sleep state
Waiting for Driver	Waiting on return of driver function
Waiting for Datamove	Waiting on return of KS_MoveData (may be part of service using mailbox)
Waiting for Event	Waiting on return of KS_EventW
Waiting for Enqueue	Waiting on return of KS_Enqueue(W)(T)
Waiting for Dequeue	Waiting on return of KS_Dequeue(W)(T)
Waiting for Send	Waiting on return of KS_Send(W)(T)
Waiting for Receive	Waiting on return of KS_Receive(W)(T)
Waiting for Sema	Waiting on return of KS_Wait(T)

Waiting for Semalist	Waiting on return of KS_WaitM(T)
Waiting for Resource	Waiting on return of KS_Lock
Waiting for Allocation	Waiting on return of KS_Alloc(W)
Waiting for Network	Waiting for network reply - transient state

15.2.6. Queues

This command produces a snapshot of the queues in the system as shown below. Six columns are used in the snapshot.

Name	the queue's symbolic name
Ncurr	the current number of entries in the queue
Nmax	the maximum number of entries ever been in use
Size	the queue size as defined at system generation
Count	the number of times the queue was enqueued or dequeued.
Waiters	a list of waiting tasks to enqueue or dequeue.

The Queue snapshot appears as:

```
>q
QUEUE STATES
Name    Level     Nmax    Size    Count    Waiters
-----------------------------------------------------
CONIQ       0        0      32        1
CONOQ       0      121     256      731    CONODRV
DEMOQX1     0      953       4       45   QUEUETST
DEMOQX4     0     1000       8       87
```

15.2.7. Semaphores

The semaphores' state are represented in four columns:

Name	the semaphore's symbolic name
Level	the current level value of the semaphore
Count	the total number of times the semaphore was signalled
Waiters	the tasks waiting on the semaphore to be signaled

```
> s
SEMAPHORE STATES
Name    Level    Count  Waiters
--------------------------------
SEM0        0      123    TEST1
SEM1        0        0
```



```
SEM2        3        0
SEM3        3        0
SEM4        0        0
DEBUGGO     0        1
```

15.2.8. Resources

The resource states are provided in 5 columns.

Name	the symbolic name
Count	the number of KS_Lock (W) requests that were made
Confl	the number of times the resource was locked when a KS_Lock(W) request was made
Owner	the current owner task
Waiters	the tasks in the resource waiting list

```
> r
RESOURCE STATES
Name         Count Confl   Owner  Waiters
-----------------------------------------------------
 0 HOSTRES   26370   83  TLDEBUG
 1 CONRES        0    0
 2 GRAPHRES    574  449  TLDEBUG BALL1 BALL3 ...
```

15.2.9. Memory Partitions

The memory partition information is given six columns:

Name	the map's symbolic name
NCurr	the number of blocks in use
Nmax	the maximum number of blocks ever used
Size	the size of the map as defined at system generation
Count	the numbers of time a block of the map was allocated or deallocated
Waiters	the list of waiting tasks

```
> a
MAP STATES
 # Name Ncurr     Nmax     Size    Count    Waiters
-------------------------------------------------------
 0 MAP1     2        3        4        7
 1 MAP2    40       40       40       67      Task3
```

15.2.10. Tracing monitor

The tracing monitor is a part of the Virtuoso Task Level Debugger. During normal kernel activities, relevant information about the system is saved in a circular buffer, so that the recent history of the system can be traced back. The main access point to this information is through the List monitor command of the debugger. This was explained above.

The monitor is configured by two defines in the MAIN#.C files. These can be modified if necessary.

• MONITSIZE . . . . . the length of the circular buffer used to store monitoring information (number of items stored).

• MONITMASK . . . . the type of information to be traced. This value can be any combination (bitwise OR) of the following values (defined in iface.h):

```
#define MON_TSWAP 1 /* task swaps */
#define MON_STATE 2 /* task state changes */
#define MON_KSERV 4 /* kernel service calls */
#define MON_EVENT 8 /* events */
#define MON_ALL 15  /* all the above */
```

MONITMASK is used to initialize a global variable ‘int monitmask’, that can be modified at run time, e.g. if you want to monitor only part of an application.

• Note . . . . . Each traced event adds a few micoseconds to the execution time !

The following information is listed :

#	the event number in the trace
dt	the time difference in high precision timer ticks with the previous event
time	the absolute time of the event
object	the kernel object related to the action
action	what happened, described below

The ‘object‘ and ‘action’ fields depend on the type of information.

For task swaps, object is the task name, and action will be the string “Swapped in”. The task name “-- idle --” refers to the NULL task.

For task state changes, object is the task name, and action will be the name of the bit that was changed, prefixed by ‘+’ (bit set), or ‘-’ (bit reset). A task is ready to run if all its state bits are reset. The state bits are described in section 14.2.5 above.

For events, the object field is not used, and the action field shows the event number.

The MON_KSERV option shows all commands received by the kernel on the current processor. Some of these correspond to kernel services requested by local tasks. In this case, the object field shows the task name. Others are commands that arrive from other processors in the network In this case, the node ID of the source of the command will be shown. The commands, shown in the action field, are:

NOP	KS_Nop
USER	KS_User
READWL	KS_Workload
SIGNALS	KS_Signal
SIGNALM	KS_SignalM
RESETS	KS_ResetSema
RESETM	KS_ResetM
WAITS_REQ	KS_Wait(T)
WAITS_RPL	internal message for KS_Wait(T)
WAITS_TMO	id.
WAITMANY	KS_WaitM(T)
WAITM_REQ	internal message for KS_WaitM(T)
WAITM_RDY	id.
WAITM_CAN	id.
WAITM_ACC	id.
WAITM_END	id.
WAITM_TMO	id.
INQSEMA	KS_InqSema
LOCK_REQ	KS_Lock(W)(T)
LOCK_RPL	internal message for KS_Lock(W)(T)
LOCK_TMO	id.

UNLOCK	KS_Unlock
ENQUE_REQ	KS_Enqueue
ENQUE_RPL	internal message for KS_Enqueue(W)(T)
ENQUE_TMO	id.
DEQUE_REQ	id.
DEQUE_RPL	id.
DEQUE_TMO	id.
QUE_OP	KS_InqQueue or KS_PurgeQueue
SEND_REQ	KS_Send(W)(T)
SEND_RPL	internal message for KS_Send(W)(T)
SEND_TMO	id.
SEND_ACK	id.
RECV_REQ	KS_Receive(W)(T)
RECV_RPL	internal message for KS_Receive(W)(T)
RECV_TMO	id.
RECV_ACK	id.
ELAPSE	KS_Elapse
SLEEP	KS_Sleep
WAKEUP	internal message for KS_Sleep
TASKOP	all operations on a single task
GROUPOP	all operations on a task group
SETPRIO	KS_SetPrio
YIELD	KS_Yield
ALLOC	KS_Alloc(W)(T)
DEALLOC	KS_DeAlloc(W)(T)
TALLOC	KS_AllocTimer
TDEALLOC	KS_DeallocTimer
TSTART	KS_StartTimer
TSTOP	KS_StartTimer
DRIV_REQ	driver call starts
DRIV_ACK	driver call returns
ALLOC_TMO	internal message for KS_Alloc(W)(T)
REMREPLY	internal message (many services)
DEBUG_REQ	internal message used by debugger task
DEBUG_ACK	id.
TXDATA_REQ	KS_MoveData
TXDATA_ACK	internal message for data transfer protocol

RXDATA_REQ	KS_MoveData
RXDATA_ACK	internal message for data transfer protocol
RAWDATA_REQ	id.
RAWDATA_ACK	id.
DATAWAIT	id.
EVENTWAIT	KS_EventW

An example is given below :

```
> l
 VIRTUOSO TL Debugger 3.09 NODE 1 NODE1
 TL MONITOR
 #         dt     time     object  action
-----------------------------------------------
 1004    8181 67020119 -- idle --  Swapped in
 1005   10002 67030121             Event # 48
 1006    9998 67040119             Event # 48
 1007   10000 67050119             Event # 48
 1008   10000 67060119             Event # 48
 1009   10000 67070119             Event # 48
 1010   10000 67080119             Event # 48
 1011    2538 67082657             Event # 14
 1012      89 67082746  GRAPHDRV   - Event
 1013     102 67082848  GRAPHDRV   Swapped in
 1015     139 67083142   TLDEBUG   - Dequeue
 1016     114 67083256   TLDEBUG   Swapped in
 1017     156 67083412   TLDEBUG   LOCK_REQ
 1018     129 67083541   TLDEBUG   + Resource
 1019     137 67083678  GRAPHDRV   Swapped in
 1020     148 67083826  GRAPHDRV   UNLOCK
 1021     133 67083959   TLDEBUG   - Resource
 1022     106 67084065   TLDEBUG   Swapped in
 1023     145 67084210   TLDEBUG   GROUPOP
```

The trace above retraces the execution history on a 40 MHz TMS20C40 at the moment the task level debugger was called.

Note:

On target boards with no high precision timer, Virtuoso often implements a low resolution timer (e.g. with a tick of 1 millisecond). As a result, the timing interval rather coarse grain and it is the order of the events that is the most relevant information.

15.2.11. Mailboxes

This command lists the current requests for sending and receiving messages in a particular mailbox.

Name	the mailbox' symbolic name
Count	the number of current entries
Waiters	List of waiting tasks
Waiters - W	tasks waiting for return of KS_Send(W)(T)
Waiters - R	tasks waiting for return of KS_Receive(W)(T)

Example :

```
> m
 MAILBOX STATES
 #       Name    Count  Waiters
-----------------------------------------------------------
0     GETPOSM      504        W
                              R BALL2 BALL4 BALL6 BALL8
1      REPORT      508        W
                              R MASTER
```

15.2.12. Network buffers

(Subject to change).

15.2.13. Clock/Timers

This debugger command will list the remaining events on the timer list. Four informations are provided:

Time	the remaining time before the timer event will arrive
Repeat	an eventual periodically scheduled timer interval
Action	the action to be done when the timer event arrives

Object	the type of object on which the actions applies.

The possible actions are:

TO xxx	Kernel service xxx times out
Wakeup	KS_Sleep ends
Timed Signal	Associated semaphore will be signalled

```
> c
Time   Repeat   Action   Object
--------------------------------
145          -   Wakeup CONDEMO
```

15.2.14. Stack Limits

This function is intended to assist the user in tuning the use of RAM needed for stack space by tasks as well as by Virtuoso. The snapshot consists of four columns.

Task	the task's symbolic name
Current	the current amount of stack being used
Used	the maximum amount of stack ever used
Size	the defined stack size

```
> k
 STACK STACKS
 Task Current     Used     Size
 -------------------------------------
 TLDEBUG   45      201      256
 CONODRV   16       38       64
 CONIDRV   16       35       64
 HIQTASK   16       45       64
 HIMTASK   16       45       64
 SEMATAS   64       16       64
 CONDEMO   60      227      256
```

15.2.15. Zero Queue/Map/Resource Statistics

This command will cause all of the usage statistics for queues, memory partitions, and resources to be reset.

No other user input is required.

15.2.16. Other processor

When invoked, you are prompted for a valid processor number and the debugger interface is opened on the requested processor. From then on, all debugging commands will be executed on the requested processor.

Example :

> o2

15.2.17. Task Manager

Note : not always implemented.

Task Manager Mode allows the user to do some types of task management operations via the debug console. Selection of this command causes a special prompt to indicate that the debugger is in Task Manager Mode. The prompt appears as:

```
TLDEBUG: 1>$
$TLDEBUG
```

The Task Manager Mode menu may be displayed by responding to the prompt with a “H” (or “h”) followed by an Enter key. The Task Manager Mode menu is displayed as shown below.

```
S - Suspend
R - Resume
A - Abort
T - Start
X - Exit $TLDEBUG
```

Except for the Exit (X) command, all of the commands in the Task Manager Mode require that a task number be entered. The task identifier prompt appears as:

```
Task >
```

The user’s response to the prompt is a decimal task number or the task ‘s symbolic identifier as defined during the system generation procedure. The entry is terminated by an Enter key.

15.2.18. Suspend

Execution of this command causes the specified task to be suspended. The task cannot be restarted until it is resumed by another task or by operator

command via the debugger.

15.2.19. Resume

This command removes the state of suspension on the specified task. If no other blocking condition exists, the task is made ready to run.

15.2.20. Abort

This command causes the specified task to be aborted. The waiting lists are cleared and the task abort handler will become executable upon leaving the debugger. This command is not implemented for all target processors.

15.2.21. Start

A task may be restarted by the selection of this command. The specified task is started at the entry point specified during the system generation procedure. This command is not implemented for all target processors.

NOTE:

The Abort and Start commands should be used with caution. As the normal flow of execution is suspended when in the debugger. Therefore, after aborting a task, the user should exit from the debugger to allow the task to terminate normally before restarting the task.

15.2.22. Exit $TLDEBUG

This command causes the Task Manager Mode to terminate. The standard debugger command prompt is reissued.

15.2.23. Exit TLDEBUG

Invocation of this command causes the debugger to unblock all other tasks and to suspend operations. Control is given to the highest priority task that is runnable.

15.2.24. Help

This command causes the debugger Command Menu to be displayed.

15.3. The Workload Monitor

Two kernel services have been added for reading the workload on a processor.

```
int KS_Workload (void);
```

This returns the average workload as an integer value from 0 to 1000 and is to be read as tens of percentage.

```
void KS_SetWlper (int T);
```

This sets the workload measuring interval to T ms. T will automatically be reduced to the nearest limit of the range 10 ... 1000 ms if a value outside this range is given. The default value is 100 ms.

Note :

For calibration reasons, the current implementation is based on the presence of a high resolution timer. The workload measurement is not implemented on target processors that only have a low resolution timer.

The workload monitor works as follows:

At any moment a processor executing a Virtuoso application can be occupied in one of three different ways. It can

1. execute kernel code,
2. execute a microkernel task,
3. or be ‘idle’.

While it is ‘idle’, the processor actually executes the lowest priority microkernel task in the system, sometimes called the ‘nulltask’. This task is created automatically by the kernel, and it is in fact task (N +1), not zero. In the monitor listings it is shown as ‘- idle -’. It is in fact the continuation of the C main () function after it has installed the kernel and started the user tasks.

The nulltask code is little more than an endless loop, counting its own number of iterations. By reading the iteration counter at times t0 and t1, we can determine the amount of time, Ti, a processor has been idle during the interval t0 ... t1.

The workload is then calculated as follows:

```
W = 1 - (Ti / (t1 - t0))
```

The kernel function KS_Workload () actually returns the integer value 1000*W.

The measuring interval can be changed by a calling the kernel function KS_SetWlperiod (T). The default value is 100 ms, while legal values for T range from 10 to 1000 ms. Every T ms, the kernel takes a look at the iteration counter in the nulltask. It keeps a record of the two most recent values, and of the times when these were read. When workload() is called, the returned value is computed from the current time and iteration count, and the values taken at the start of the last full interval. The actual measuring period can therefore be any value from T to 2T.

To obtain meaningful results, the workload measuring interval should be significantly longer than the time ‘granularity’ of your application. A special situation arises when some tasks are executed at a constant frequency (e.g. when scheduled by a periodic timer). In this case it is sometimes possible to observe interference patterns between the task and the workload monitor period As an extreme case, consider what will happen if both periods are more or less equal. If too short a measuring interval is used, the results will be largely determined by the phase relationship between the two periodic events. This could give a very misleading indication of the real average workload.

Note also that in communication intensive parts of the application, the workload can be relatively high even if the application tasks are not very busy. This is caused by the communication drivers that are part of the kernel.

16. Practical hints for correct use

16.1. Flexible use of the messages

16.1.1. General features

The Virtuoso message system has been designed to fully support the 'Virtual Single Processor Model', which is the very essence of Virtuoso. To achieve this, it has the following key features:

1. Messages are synchronous: both tasks involved in a message exchange are descheduled until the transfer of data is finished.
2. The default action by the kernel is always to copy the message data, even if the sending and receiving tasks are running on the same processor. The user must explicitly indicate that he wants it otherwise.
3. All stages of the message mechanism are prioritized: a high priority message is never forced to wait while a lower priority operation occupies the network resources.

Synchronous operation enables the kernel to defer the actual copying of the data until both sides are ready for the transfer. As a result, it is not possible to flood the network buffers with messages that nobody wants to receive.

Always passing messages 'by value' facilitates the design of user code that will work on any node. If a copy of the message data is made, the receiver is free to specify the exact destination of the data. If only a pointer is passed, the receiving task has no such choice, and special code will be needed to handle this situation. In many cases, a copy would still be required for the receiver to function correctly.

Both features also cooperate to ensure that correct behavior of an application does not depend on the hidden sequentialisation which is imposed when sender and receiver are on the same node.

It is still possible to pass messages 'by reference', if your application demands it. This, however, has to be done explicitly. In this way, the use of a node-dependent operation is clearly documented in the source code.

In many applications the message system will be the primary user of the network resources. In these circumstances, prioritizing is essential to preserve the real time nature of the kernel.

16.1.2. Mailboxes

When two tasks want to exchange a message, they do not talk to each other directly. They will both use the services of an intermediate object, called a ‘mailbox’. Mailboxes provide a degree of isolation between the two parties involved in a data transfer. As an example, a mailbox could be associated with a device. Tasks wishing to use the device would then not need to know the task identity of the device driver. This could even change at run time without the ‘clients’ of the device being aware of the replacement. Note also that a mailbox can reside on a processor node where neither the sender nor the receiver reside.

A mailbox acts as a ‘meeting place’ for tasks wishing to transmit or receive a message. It maintains two waiting lists: one for senders, and one for receivers. When a new send or receive request arrives, the mailbox searches one of the lists for a corresponding request of the other type. If a match is found, it is removed from the waiting list, and the data transfer is started. When this has finished both tasks are allowed to resume execution. If no match can be found, the caller is suspended and put on a waiting list, or the operation fails.

In above scheme, one should consider the data copy as a side effect of obtaining a match between a sender and a receiver message request. The sender and the receiver exchange the request in the mailbox. Therefore, the message operation is actually composed of the two requests and an eventual copy of the data referenced by the message.

16.1.3. Using messages

The receiving task may want to take different actions depending on the size, type, or sender of the message. It is possible for the receiver to obtain the requested message first, and delay the copy until it has decided what to do with the data. When the ‘rx_data’ field in the requested message of the receiver is NULL, the kernel will return to the receiving task without performing the data copy, and the sending task will not be rescheduled.

The receiver can then base its decision on the ‘size’, ‘info’ or ‘tx_task’ fields of the message, which will have been updated by the kernel. Finally it issues a KS_ReceiveData call to obtain the data and to reschedule the sending task. This is illustrated in the following example.

This receiver inspects the ‘tx_task’ field to separate messages from a number of senders. One of the sending tasks uses the ‘info’ field to specify an array index.

```
void Receiver() {
    K_MSG      M;
    MyType     MyArray[100];

M.tx_task = ANYTASK;
M.rx_data = NULL;
M.size = 999999; /* larger than any expected message */
KS_ReceiveW(MAILBOX1, &M);

switch (M.tx_task) {

  case SENDER1:
    /*
     * put data in MyArray, at index given by sender
     */
    M.rx_data = &MyArray[M.info];
    KS_ReceiveData(&M);
    ...
    break;

  case SENDER2:
    /*
     * allocate space, then receive data
     */
    M.rx_data = KS_AllocW(MAP_2);
    KS_ReceiveData(&M);
    ...
    KS_Dealloc(MAP_2, M.rx_data);
    break;

  default:
    /*
     * unknown sender, dump message
     */
    M.size = 0;
    KS_ReceiveData (&M);
  }
}
```

If a receiving task does not really need a private copy of the message data in order to function correctly, it can be written in such a way that a copy is made only if the sender is on a different node. The next example shows how this can be achieved. As in the previous example, the receiver calls KS_Receive with its data pointer equal to NULL. When the call returns, the receiver inspects the message to see whether the sending task is on the same node.

If it is, the receiver uses the pointer provided by the sender to access the data, and reschedules the sender by performing a KS_ReceiveData call with the data size set to zero. Otherwise the receiver uses KS_ReceiveData in the normal way, to obtain a copy of the data and reschedule the sender.

Note that it is not necessary to modify the sending task in order to use this method.

```
void Receiver() {
    K_MSG    M;
    MYTYPE   MyData, *MyPtr;
    int      SameNode;

    M.tx_task = ANYTASK;
    M.rx_data = NULL;
    M.size = sizeof (MYTYPE);

    KS_ReceiveW(MAILBOX1, &M);
    SameNode = (KS_NodeId(M.tx_task) == KS_NodeId(M.rx_task));
    if (SameNode) {
        MyPtr = M.tx_data;
    } else {
        M.rx_data = MyData;
        KS_ReceiveData(&M);
        MyPtr = M.rx_data
    }
    /*
     * use message, access via MyPtr
     */

    if (SameNode) {
        M.size = 0;
        KS_ReceiveData(&M);
    }
}
```

If both tasks are known to be on the same node, this could be simplified further, as shown below. Note that a KS_ReceiveData call (with size = 0) is still necessary.

This is the recommended way to pass messages ‘by reference’, as it is transparent to the sender, and leaves the ‘info’ field available for other purposes.

```
void Receiver() {
    K_MSG   M;
    MYTYPE *MyPtr;

    M.tx_task = ANYTASK;
    M.rx_data = NULL;
    M.size = 0;
    KS_ReceiveW(MAILBOX1, &M);
    MyPtr = M.tx_data;
    /*
     * use message, access via MyPtr
     */
    KS_ReceiveData(&M); /* will reschedule sender, acts as an ack */
}
```

16.2. On the abuse of semaphores

In order to synchronize tasks, one might be tempted to use semaphores. If however, also data is to be transferred following the synchronization, one should be aware that the use of a semaphore is superfluous as the data transfer itself will synchronize the two tasks.

Even more, a semaphore will not assure perfect synchronization as a semaphore is counting. Hence, a task could signal more than once while the synchronizing task has not even started up. When using queues to transfer data, one obtains this effect as long as the queue is not full. On the other hand, a message transfer will assure perfect synchronization as the sending as well as the receiver task synchronize at the same time.

16.3. On using the single processor versions for multiple processors

If you only need a few processors and if you need to minimize on memory usage at all cost, it might be advisable to use the multi processor version of Virtuoso (/MP implementations). The difference is that the (/VSP) distributed version of the kernel has an embedded router enabling to keep the original Virtuoso source code programs while you only will need to reallocate the different tasks, queues, semaphores, etc. So simply by adding processors and by invoking the configurer of the compiler, the system will run faster without changing any of the source code (provided you used separate compilation in the first phase). Of course, this kernel uses more memory.

In order to use the single processor version on multiple processor, you will develop the program running on each processor as usual and configure the system using the configurer from the compiler. This time however, you will need to use the kernel services KS_LinkinW() and KS_LinkoutW() or equiva-

lent to communicate between the different Virtuoso tasks. So it is best to keep this interaction as simple as possible, preferably under the form of block transfers. The reason is first of all that you are talking to the naked hardware, just as you would do without support from the kernel. So both receiving and sending tasks need to speak the same protocol. If for example, the length of the block sent is different from the length of the message received, the communication might stall and never terminate. Normally, this is called deadlock. Normally, you can detect this with the debugger. Note however that the debugger can only be used on the root processor, while with the distributed version, the debugger task can be invoked on each processor.

In addition, as you are using a priority based scheduler, the timing behavior might change! Remember that with the single processor version, you loose the prioritizing capability of the routing system. Hence, the programmer himself must ensure that the link communication does not jeopardize the real time performance.

When you keep these guidelines in mind, using the single processor version of Virtuoso on multiple processors, should be not much of a problem.

16.4. Hints on system configuration

Although Virtuoso permits you to place tasks, semaphores, mailboxes and queues on any processor you want, the actual placement will influence the system's performance.

As a general rule, you are advised to place all semaphores, mailboxes and queues on the same processor as the tasks that use them or at least with a minimum of communication distance between them. In general, it is best to place the queues, mailboxes and semaphores on the processors that contain the receiving tasks.

Note that you can force the routing a little bit by carefully moving a kernel object to node part of the communication path you want to be used. This works well for all command packets, however datatransfer always happens using the shortest path. If you want to privilege certain data transfers, you should assign a higher priority to the message.

As to the global system configuration, best is to keep the main control program centrally located and close to the host server if fast operator interaction or filing on the host system is required.

In general one can see that system performance will benefit if highly interactive tasks are grouped together while less interactive tasks are placed more remotely. In general this means that computation intensive tasks should be distributed evenly over the system while receiving a lower priority.

Note however that this is only a guideline, especially as some of the requirements impose a compromise between conflicting demands. No algorithm exists to find the optimum solution. For the moment, use your common sense and the debugging monitor to fine tune the system.

16.5. Customized versions and projects

Virtuoso is delivered as a standard kernel with standard features and provides an adequate solution in most circumstances. This is especially true for embedded systems, where maximum performance is desired while memory usage is minimized and the system is rather static over its life time.

In some cases, more functionality is wished, such as:

1. capability to monitor the system at runtime
2. capability to dynamically change the system
3. capability to use a different scheduling algorithm

Eonic Systems is aware of these special requirements, especially as most of these less standard features are typical for distributed and parallel processing systems. As such, Eonic Systems is already working on new Virtuoso versions that support dynamic real time features and fault-tolerance. Customers with special needs can be supported in different ways:

1. by adapting the kernel themselves
2. by buying a customized version
3. talking to our team to find an alternative but equivalent solution

As the kernel itself is highly modular, although compact, most customer specific functions can be provided on short notice.

If you would prefer Eonic Systems to integrate the kernel as part of a full custom solution, this can be negotiated. For more information contact Eonic Systems or your distributor.

17. Microkernel C++ interface

17.1. Microkernel C++ classes

In section 7.1. on page 3 an overview was given of the microkernel objects and their services. The C++ interface for the microkernel provides a tighter coupling between an object type and its services. Also, global C++ objects representing each kernel object defined by the user in the sysdef file can be generated automatically by the sysgen tool.

The following table indicates the relationship between kernel object types and the corresponding C++ class that encapsulates these objects:

Kernel Object Type	Related C++ class(es)
Tasks	KTask, KActiveTask
Task Groups	KTaskGroup
Semaphores	KSemaphore
Mailboxes	KMailBox
Queues	KQueue
Memory Maps	KMemoryMap
Resources	KResource
Timers	KTimer

At this moment, no C++ class has been provided for encapsulating semaphore lists. One extra class is available that provides a C++ shell for the KMSG datastructure. This class is named KMessage.

The C++ classes have been designed to introduce minimal overhead both in memory usage (code and data size) and run-time behavior. This is accomplished by making extensive use of inlined functions.

In the following section, we will indicate how sysgen takes care of generating the C++ objects corresponding to the ones defined by the user in the sysdef file. Thereafter, we will give a reference type overview of the microkernel classes and their functions. Finally, a complete example on how to use the interface is given.

17.2. Kernel object generation by sysgen

Sysgen normally generates a number of C andheader files to define kernel object ID's and data structures. Using command line switches, it becomes

possible to have sysgen generate C++ header and implementations files that declare and define the C++ kernel objects.

In order to generate the C++ files, the user must pass the flag -oCH to sysgen. The usual C files and header files are still needed to make a valid application, so one should always use -ochCH as command line flags to generate all necessary files.

When given the C and H parameters to the -o flag, sysgen will generate the following files:

- vkobjcts.hpp:this file contains the declarations for the kernel objects that are visible to all nodes of the target. This are all objects defined in the sysdef file, with the exception of the memory map objects.
- vkobjcts.cpp: this file defines and initializes the objects declared in vkobjcts.hpp. The correct parameters are passed to the constructors of the objects.
- vkob*nnnn*.hpp: this file declares the objects that are local to node *nnnn*. This is restricted to the memory maps objects.
- vkob*nnnn*.cpp: this file defines and initializes the objects declared in vkob*nnnn*.hpp. The correct parameters are passed to the constructors of the objects.

The objects defined in the files written by the sysgen correspond to objects defined in the sysdef file. The only exception is the object representing the active task. The name of the object representing the currently active task is ActiveTask. The names of the other C++ objects are related to the name ID's given in this sysdef file.

The cpp files where these objects are defined and initialized makes the relationship between the object names and the name ID's very explicit. The argument to the constructor for a global kernel object is indeed the name ID of the object. If the user wants to control the naming of the C++ objects, he must understand the mechanism that maps an ID name to a C++ object name.

The C++ object name is constructed from the sysdef name ID using the following scenario:

1. If the first three or more characters of the ID name are identical to the first characters of the object class name (without the leading K of course) then these characters will be replaced by the class name. The remainder of the ID name will be converted to lowercase, except for the first character, which is forced to uppercase.

   Examples: QUEUE1 becomes Queue1. The ID MAILBRESULT of a mailbox object is seen as MAILB + RESULT. As the first five characters match the first five characters of "mailbox", sysgen will generate the name MailboxResult for this mailbox object.
2. If the last three or more characters of the ID name are identical to the

first characters of the object class name, then these identical characters will be replaced by the class name. The remainder of the ID name will be converted to lowercase, except for the first character, which is forced to uppercase.

Examples: HIPRIOTASK becomes HiprioTask, HOSTRES becomes HostResource.

3. In case we don't find (part) of the object type name in the Id given by the user, we convert the ID name to lowercase, except the first which is converted to uppercase, and append the object type name.

   Example: a task object named MASTER becomes MasterTask.

For a complete example listing, we refer to section 17.13. on page 188.

17.3. KTask

The class that encapsulates kernel tasks has the name KTask. The class definition is as follows:

```
class KTask
{
private:
      K_TASK m_TaskID;
// construction/destruction
public:
      KTask(K_TASK taskid);
// access functions
public:
      K_TASK GetID() const;
// operations
public:
      void SetEntryFunction(void (*taskentryfunction)(void));
      void SetPrio(int priority);
      void Start();
      void Abort();
      void Suspend();
      void Resume();
}
```

The constructor takes the task ID number as an argument. The GetID() member function allows retrieving the ID of a task object. Other member functions called upon a KTask object will result in kernel service calls with the task ID of the task object as the first argument. The mapping between member functions and kernel services is the following:

`task.SetEntryFunction(...)` calls `KS_SetEntry(TASKID, ...)`

```
task.SetPrio(...)          calls   KS_SetPrio(TASKID, ...)
task.Start()               calls   KS_Start(TASKID)
task.Abort()               calls   KS_Abort(TASKID)
task.Suspend()             calls   KS_Suspend(TASKID)
task.Resume()              calls   KS_Resume(TASKID)
```

17.4. KActiveTask

The KActiveTask class is introduced to cover the kernel services that operate on the task that calls the service. There is thus only one instance of this class, and this object is always named ActiveTask in the files generated by sysgen. The class definition looks as follows:

```
class KActiveTask
{
private:
// construction/destruction
public:
// access functions
public:
       KTask GetTask();
       K_PRIO GetPriority() const
       UNS32 GetGroupMask();
// operations
public:
       void JoinGroup(KTaskGroup group);
       void LeaveGroup(KTaskGroup group);
}
```

The GetTask() member is used to retrieve a KTask object representing the current task. This (temporary) object can then be used to call kernel services mapped to the KTask class.

GetPriority() returns the priority of the current running task. GetGroupMask() returns an unsigned integer that holds a bit mask, where each bit that is set to one indicates a group that the task belongs to.

JoinGroup can be used to associate the active task with the taskgroup given in the argument. LeaveGroup removes this association.

17.5. KTaskGroup

The class that is used to make objects corresponding to groups of tasks has

the name KTaskGroup. The class definition is as follows:

```
class KTaskGroup
{
private:
      K_TGROUP m_TaskGroupID;
// construction/destruction
public:
      KTaskGroup(K_TGROUP taskid);
// access functions
public:
      K_TGROUP GetID() const;
// operations
public:
      void Start();
      void Abort();
      void Suspend();
      void Resume();
//implementation:
private:
}
```

The constructor takes the task group ID number as an argument. The GetID() member function allows to retrieve the ID of a task group object. Other member functions called upon a KTaskGroup object will result in kernel service calls with the task group ID of the task group object as the first argument. The mapping between member functions and kernel services is the following:

`taskgroup.Start()`	calls	`KS_Start(TASKGROUPID)`
`taskgroup.Abort()`	calls	`KS_Abort(TASKGROUPID)`
`taskgroup.Suspend()`	calls	`KS_Suspend(TASKGROUPID)`
`taskgroup.Resume()`	calls	`KS_Resume(TASKGROUPID)`

17.6. KSemaphore

Objects representing kernel semaphores are of the class type KSemaphore. The class definition is as follows:

```
class KSemaphore
{
private:
      K_SEMA m_semaphoreID;
// construction/destruction
public:
      KSemaphore(K_SEMA semaid);
// access functions
public:
      K_SEMA GetID();
      int GetCount();
// operations
public:
      void Reset();
      void Signal();
      int TestW();
      int TestWT ();
      // old, for compatibility
      int Wait();
      int WaitT();
}
```

The constructor takes the sempahore ID number as an argument. The GetID() member function allows to retrieve the ID of a semaphore object. Other member functions called upon a KSemaphore object will result in kernel service calls with the semaphore ID of the semaphore object as the first argument. The mapping between member functions and kernel services is the following:

`semaphore.GetCount()`	calls	`KS_InqSema(SEMAID)`
`semaphore.Reset()`	calls	`KS_ResetSema(SEMAID)`
`semaphore.Signal()`	calls	`KS_Signal(SEMAID)`
`semaphore.TestW()`	calls	`KS_TestW(SEMAID)`
`semaphore.TestWT(...)`	calls	`KS_TestWT(SEMAID, ...)`
`semaphore.Wait()`	calls	`KS_Wait(SEMAID)`
`semaphore.WaitT(...)`	calls	`KS_WaitT(SEMAID, ...)`

17.7. KMailBox

Objects representing kernel mailboxes are of the class type KMailBox. The class definition is as follows:


```
class KMailBox
{
private:
      K_MBOX m_MailBoxID;
// construction/destruction
public:
      KMailBox(K_MBOX mailboxid);
// access functions
public:
// operations
public:
      int Send(K_PRIO priority, K_MSG* message);
      int SendW(K_PRIO priority, K_MSG* message);
      int SendWT( K_PRIO priority,
                  K_MSG* message,
                  K_TICKS timeout);
      int Receive(K_MSG* message);
      int ReceiveW(K_MSG* message);
      int ReceiveWT(K_MSG* message, K_TICKS timeout);
}
```

The constructor takes the mailbox ID number as an argument. Other member functions called upon a KMailBox object will result in kernel service calls with the mailbox ID of the mailbox object as the first argument. The mapping between member functions and kernel services is the following:

`mailbox.Send(...)`	calls	`KS_Send(MBID, ...)`
`mailbox.SendW(...)`	calls	`KS_SendW(MBID, ...)`
`mailbox.SendWT(...)`	calls	`KS_SendWT(MBID, ...)`
`mailbox.Receive(...)`	calls	`KS_Receive(MBID, ...)`
`mailbox.ReceiveW(...)`	calls	`KS_ReceiveW(MBID, ...)`
`mailbox.ReceiveWT(...)`	calls	`KS_ReceiveWT(MBID, ...)`

17.8. KMessage

The KMessage class provides a C++ interface to the K_MSG data structure. Because public derivation is used, one can still access all members of the K_MSG structure directly. A conversion operator to a pointer to a K_MSG is also provided, so one can use a KMessage object wherever a pointer to a K_MSG structure is needed. The class interface for KMessage is defined as follows:

```
class KMessage
       : public K_MSG
{
private:

// construction/destruction
public:
       KMessage();
       KMessage(K_MSG message);
// access functions
public:
       INT32 GetSize() const;
       INT32 GetInfo() const;
       KTask RequestedSender() const;
       KTask RequestedReceiver() const;
// conversion operators:
public:
       operator K_MSG*();
// operations
public:
       void ReceiveData();
}
```

Most of the operators are defined as const, and do not allow any changes to be made to the data members of the K_MSG class. If they have to be changed, one must use the data members of the K_MSG class directly. An extra functionality is also that the KMessage class creates and returns KTask objects with the task IDs as defined in the K_MSG struct. The correspondence between the access function of the KMessage class and the data members of the K_MSG struct are as follows:

`message.GetSize()`	corresponds to `msg.size`
`message.GetInfo()`	corresponds to `msg.info`
`message.RequestedSender()`	corresponds to `msg.tx_task`
`message.RequestedReceiver()`	corresponds to `msg.rx_task`

17.9. KQueue

Objects representing kernel queues are of the class type KQueue. The class definition is as follows:


```
class KQueue
{
private:
      K_QUEUE m_QueueID;
// construction/destruction
public:
      KQueue(K_QUEUE queueid);
// access functions
public:
      K_QUEUE GetID();
// operations
public:
      int Enqueue(void* data, int size);
      int EnqueueW(void* data, int size);
      int EnqueueWT (void* data, int size, K_TICKS timeout);
      int Dequeue(void* data, int size);
      int DequeueW(void* data, int size);
      int DequeueWT (void* data, int size, K_TICKS timeout);
      void Purge();
      int NumberOfEntries();
}
```

The constructor takes the queue ID number as an argument. Other member functions called upon a KQueue object will result in kernel service calls with the queue ID of the queue object as the first parameter. The mapping between member functions and kernel services is the following:

`queue.Enqueue(...)`	calls	`KS_Enqueue(QID, ...)`
`queue.EnqueueW(...)`	calls	`KS_EnqueueW(QID, ...)`
`queue.EnqueueWT(...)`	calls	`KS_EnqueueWT(QID, ...)`
`queue.Dequeue(...)`	calls	`KS_Dequeue(QID, ...)`
`queue.DequeueW(...)`	calls	`KS_DequeueW(QID, ...)`
`queue.DequeueWT(...)`	calls	`KS_DequeueWT(QID, ...)`
`queue.Purge()`	calls	`KS_PurgeQueue(QID)`
`queue.NumberOfEntries()`	calls	`KS_InqQueue(QID)`

17.10. KMemoryMap

Objects representing kernel memory maps are of the class type KMemory-Map. The class definition is as follows:

```
class KMemoryMap
{
private:
      K_MAP m_MemMapID
// construction/destruction
public:
      KMemoryMap(K_MAP memmapid);
// access functions
public:
// operations
public:
      void Alloc(void** memblockaddress);
      void AllocW(void** memblockaddress);
      void AllocWT(void** memblockaddress, K_TICKS timeout);
      void Dealloc(void** memblockaddress);
      int NumberOfFreeBlocks();
}
```

The constructor takes the memory map ID number as an argument. Other member functions called upon a KMemoryMap object will result in kernel service calls with the memory map ID of the memory map object as the first parameter. The mapping between member functions and kernel services is the following:

`memmap.Alloc(...)`	calls	`KS_Alloc(MMID, ...)`
`memmap.AllocW(...)`	calls	`KS_AllocW(MMID, ...)`
`memmap.AllocWT(...)`	calls	`KS_AllocWT(MMID, ...)`
`memmap.Dealloc(...)`	calls	`KS_Dealloc(MMID, ...)`
`memmap.NumberOfFreeBlocks()`	calls	`KS_InqMap(MMID)`

17.11. KResource

Objects representing kernel resource are of the class type KResource. The class definition is as follows:


```
class KResource
{
private:
      K_RES m_ResourceID
// construction/destruction
public:
      KResource(K_RES resourceid);
// access functions
public:
      K_RES GetID();
// operations
public:
      void Lock();
      void LockW();
      void LockWT(K_TICKS timeout);
      void UnLock();
}
```

The constructor takes the resource ID number as an argument. Other member functions called upon a KResource object will result in kernel service calls with the resource ID of the resource object as the first parameter. The mapping between member functions and kernel services is the following:

`resource.Lock()`	calls	`KS_Lock(RESID)`
`resource.LockW()`	calls	`KS_LockW(RESID)`
`resource.LockWT(...)`	calls	`KS_LockWT(RESID, ...)`
`resource.UnLock()`	calls	`KS_UnLock(RESID)`

17.12. KTimer

Objects representing kernel timers are of the class type KTimer. The class definition is as follows:

```
class KTimer
{
private:
      K_TIMER* m_pTimer;
// construction/destruction
public:
      KTimer();
      ~KTimer();
// access functions
public:
// operations
public:
      void Start(K_TICKS delay, K_TICKS period, KSemaphore
      sema);
      void Restart(K_TICKS delay, K_TICKS period);
      void Stop();
}
```

It is important to note that the calls to KS_AllocTimer en KS_DeallocTimer are wrapped in the constructor respectively destructor of the KTimer object. This implies that if a KTimer object goes out of scope, the timer that was wrapped by it cannot be used anymore, because it will be explicitly deallocated. If one wants to use one specific timer from different places in a program, one will either have to allocate a global timer object statically, or one must use the operator new to allocate a KTimer object on the heap. In the latter case, the (pointer to) the KTImer object can be freely passed around, but it becomes also the responsibility of the programmer to delete the KTimer object when it is not used anymore.

The mapping between member functions and kernel services is thus the following:

`constructor`	calls	`KS_AllocTimer()`
`destructor`	calls	`KS_DeallocTimer(KTIMER*)`
`timer.Start(...)`	calls	`KS_StartTimer(KTIMER*, ...)`
`timer.Restart(...)`	calls	`KS_RestartTimer(KTIMER*, ...)`
`timer.Stop()`	calls	`KS_Stoptimer(KTIMER*)`

17.13. A sample C++ application

The D1P sample (test) program (included in the distribution of Virtuoso Classico) calls most of the kernel services. A C++ version of this program, called D1Pobj is also included. We will highlight the differences between the two

versions here. We will first focus on the files generated by sysgen in the two cases, and then we will discuss the changes in the program files.

17.13.1. Sysgen generated files

the sysdef file for the D1p and the D1Pobj sample programs are the identical. Therefore, the *.C and *.H files generated by sysgen are identical. For D1Pobj, we specify the -ochCH flags to sysgen to let it also generate *.CPP and *.HPP files. The following lines in the makefile reflect this (the parts changed from D1P are underlined):

```
allnodes.h: sysdef
      pp -v sysdef $(DD)
      sysgen -ochCH sysdef.pp
      $(RM) sysdef.pp
```

The sysdef file for the D1P and D1Pobj sample is given below. Based on this sysdef file, sysgen will write the C++ specific files vkobjcts.cpp, vkobjcts.hpp, vkob1.cpp and vkob1.hpp. The following observations can be made from these three files:

- The mapping of name ID's of kernel objects given in the sysdef file to C++ kernel object names in the *.CPP files.
- The presence of all kernel objects, except memory maps, in vkobjcts.cpp. Vkob1.cpp contains the node-specific memory map object.
- The correct initialization of all C++ objects by passing the correct ID (name) in the definition of the objects. This explicit passing of the name ID also allows explicitly defines the name mapping between ID names and C++ object names.
- The presence of one C++ object in vkobjcts.cpp that is not defined in the sysdef file. This is the ActiveTask object representing the currently active task.

These C++ files generated by sysgen also have to be compiled and linked in in the executables. In general, the (compiled) vkobjcts.cpp file must be included in the executable for all nodes. The node specific file(s) vkob*nnnn*.cpp must only be linked in with the executable that has to run on node *nnnn*. The user bears the responsibility to edit the makefile in an appropriate way.

A consequence of the fact that almost all kernel objects are defined in vkobjcts.cpp is that all nodes will carry the overhead associated with the C++ objects, irrespective of the fact that the C++ object is used or not. Although the overhead per object is small,* it may be too large in those cases where a lot of objects are defined and not much memory is present at all nodes. In these cases, the user has the possibility to make node specific versions of the vkobjcts.cpp file, with those objects that are not referenced at a given

Sysdef

```
/*      taskname    node   prio   entry     stack   groups         */
/* ------------------------------------------------------------- */
TASK    STDIODRV    ROOT    3     stdiodrv    128   [EXE]
TASK    HIQTASK     ROOT    4     hiqtask     128
TASK    HIMTASK     ROOT    5     himtask     128
TASK    HISTASK     ROOT    6     histask     128
TASK    MASTER      ROOT    7     master      400   [EXE]

/*      queue       node   depth     width   */
/* ---------------------------------------- */

QUEUE   STDIQ       ROOT      64      WORD
QUEUE   STDOQ       ROOT      64      WORD
QUEUE   DEMOQX1     ROOT    1000      BYTE
QUEUE   DEMOQX4     ROOT    1000      WORD

/*      map         node    blocks    blsize  */
/* ---------------------------------------- */
MAP     MAP1        ROOT       4      1K

/*         sema       node     */
/* -------------------------- */
SEMA       SEM0       ROOT
SEMA       SEM1       ROOT
SEMA       SEM2       ROOT
SEMA       SEM3       ROOT
SEMA       SEM4       ROOT

/*         mailbox    node     */
/* -------------------------- */
MAILBOX    MAILB1     ROOT

/*         resource   node     */
/* -------------------------- */
RESOURCE   HOSTRES    ROOT
RESOURCE   STDIORES   ROOT
RESOURCE   DEMORES    ROOT
```

vkob1.cpp

```
/*
-- FILE MADE BY VIRTUOSO SYSGEN  Version 3.00
-- DO NOT MODIFY - EDIT SOURCE FILE ("sysdef") AND REMAKE
*/

#include "node1.h"
#include "vkob1.hpp"
#include "VKMemMap.hpp"

/* Node ROOT */
KMemoryMap Map1Memorymap(MAP1);
```

*. The actual overhead is compiler dependent. The overhead of 1 C++ object is at least 4 bytes, but most of the memory overhead is actually caused by the size of the C++ initialization routines.

vkobjcts.cpp

```
/*
-- FILE MADE BY VIRTUOSO SYSGEN  Version 3.00
-- DO NOT MODIFY - EDIT SOURCE FILE ("sysdef") AND REMAKE
*/

/* System wide kernel objects */
#include "vkobjcts.hpp"
#include "vkacttsk.hpp"
#include "vktask.hpp"
#include "vktskgrp.hpp"
#include "vkmbox.hpp"
#include "vksema.hpp"
#include "vkres.hpp"
#include "vktimer.hpp"
#include "vkqueue.hpp"
#include "allnodes.h"

KActiveTask ActiveTask;

KTaskGroup ExeTaskgroup(EXE_GROUP);
KTaskGroup SysTaskgroup(USR_GROUP);
KTaskGroup FpuTaskgroup(FPU_GROUP);

/* Node ROOT 0x00010000 */
KTask StdiodrvTask(STDIODRV);
KTask HiqTask(HIQTASK);
KTask HimTask(HIMTASK);
KTask HisTask(HISTASK);
KTask MasterTask(MASTER);
KQueue StdiqQueue(STDIQ);
KQueue StdoqQueue(STDOQ);
KQueue Demoqx1Queue(DEMOQX1);
KQueue Demoqx4Queue(DEMOQX4);
KSemaphore Semaphore0(SEM0);
KSemaphore Semaphore1(SEM1);
KSemaphore Semaphore2(SEM2);
KSemaphore Semaphore3(SEM3);
KSemaphore Semaphore4(SEM4);
KResource HostResource(HOSTRES);
KResource StdioResource(STDIORES);
KResource DemoResource(DEMORES);
KMailBox Mailbox1(MAILB1);
```

node stripped out. It is then also the users responsibility to remake these stripped down files each time the vkobjcts.cpp file is regenerated.

17.13.2. Changes to the program files

The file with the main function is situated in main1.c for the D1P sample, and in main1.cpp for the D1Pobj sample. Two blocks of changes can be distinguished here. The first is related to the fact that main1.cpp is a C++ file that must link with C functions. The second change involves the use of the C++ objects to start the tasks in the EXE group.

In main1.c, we find the following lines:

```
#include "iface.h"
#include "node1.h"
#include "_stdio.h"

...

extern int kernel_init (void);
extern int kernel_idle (void);
extern int netload (void);
```

In main1.cpp, these lines are wrapped in an extern “C” construct to ensure proper linkage:

```
extern "C" {
#include "iface.h"
#include "_stdio.h"
 extern int kernel_init (void);
 extern int kernel_idle (void);
 extern int netload (void);
}
```

In the main() function, the tasks in the EXE group are started. This is accomplished by the following call in main1.c:

```
KS_StartG (EXE_GROUP);
```

In main1.cpp, the C++ object encapsulating to the EXE task group is used to start all the tasks in the EXE group:

```
ExeTaskgroup.Start();
```

In test1.cpp, we find similar extern “C” constructs as in main1.cpp. We also discover extensive use of the C++ objects to perform the benchmarking tests of the D1P sample. These changes are highlighted in the file listing given below.

It is clear from this modified sample that the use of the C++ interface to Virtuoso allows a more object-centered way of writing the code. It is also possible to write simplified code with less pointers or “address-of” operators. All the mailbox operations that send or receive a message in the test1.cpp file can actually pass a KMessage object as an argument, instead of a pointer to a K_MSG structure. This implies that one can simply write “Message” as a parameter, instead of “&Message”, which improves code readability.

test1.cpp

```
extern "C"
{
#include <string.h>
#include <_stdio.h>
#include "iface.h"
#include "node1.h"
}

#include "vkobjcts.hpp"
#include "vkob1.hpp"
#include "vkmsg.hpp"

extern "C" void stdiodrv (void);
extern "C" void hiqtask (void);
extern "C" void himtask (void);
extern "C" void histask (void);
extern "C" void master (void);

static char  string [100];
static char  text2 [4096];
static KMessage Message;

#define FORMAT "%-60s - %5d\n"

#ifdef DEBUG
static void start_debug ()
{
   K_TASK k = KS_TaskId;
   KS_Enqueue (DEBUGIN, &k, sizeof (K_TASK));
}
#else
 static void start_debug () { }
#endif

void message_test (int size)
{
   int i, t;
   Message.rx_task = ANYTASK;
   Message.tx_data = string;
   Message.size = size;
   KS_Elapse (&t);
   for (i = 0; i < 1000; i++)
   Mailbox1.Send (1, Message);
   t = KS_Elapse (&t);
   t *= ticktime;
   t /= 1000;
   printf ("%20d bytes :%40d\n", size, t);
   KS_Sleep (250);
}
```

test1.cpp

```
 void benchm (void)
{
   K_TICKS et;

...

   Message.rx_task = ANYTASK;
   Message.tx_data = string;
   Message.size = 0;
   puts ("send message to waiting high priority task and wait -");
   KS_Sleep (250);
   HimTask.Start();
   KS_Sleep (100);
   KS_Elapse (&et);
   for (i = 0; i < 1000; i++)
   {
      Mailbox1.SendW (1, Message);
   }
   et = KS_Elapse (&et);
   et *= ticktime;
   et /= 1000;
   printf ("               Header only :%40d\n", et);
   KS_Sleep (250);
   message_test (8);
...
   message_test (4096);

   KS_Elapse (&et);
   for (i = 0; i < 1000; i++)
      Demoqx1Queue.EnqueueW (string, 1);
   et = KS_Elapse (&et);
   et *= ticktime;
   et /= 1000;
   printf (FORMAT, "enqueue 1 byte", et);
   KS_Sleep (250);

...

   HiqTask.Start ();
   KS_Elapse (&et);
   for (i = 0; i < 1000; i++)
      Demoqx1Queue.EnqueueW (string, 1);
   et = KS_Elapse (&et);
   et *= ticktime;
   et /= 1000;
   printf (FORMAT, "enqueue 1 byte to a waiting higher priority
task", et);

...
```


test1.cpp

```
   KS_Elapse (&et);
   for (i = 0; i < 1000; i++)
      Semaphore1.Signal();
   et = KS_Elapse (&et);
   et *= ticktime;
   et /= 1000; /* convert to us */
   printf (FORMAT, "signal semaphore", et);
   KS_Sleep (250);
   Semaphore1.Reset();
   HisTask.Start();
   KS_Elapse (&et);
   for (i = 0; i < 1000; i++)
      Semaphore1.Signal();
   et = KS_Elapse (&et);
   et *= ticktime;
   et /= 1000; /* convert to us */
   printf (FORMAT, "signal to waiting high pri task", et);
   KS_Sleep (250);

...

   KS_Elapse (&et);
   for (i = 0; i < 1000; i++)
      Semaphore4.Signal();
   et = KS_Elapse (&et);
   et *= ticktime;
   et /= 1000;
   printf (FORMAT, "signal to waitm (4), with timeout", et);
   KS_Sleep (250);
```

test1.cpp

```
   KS_Elapse (&et);
   for (i = 0; i < 5000; i++)
   {
      DemoResource.LockW();
      DemoResource.Unlock();
   }
   et = KS_Elapse (&et);
   et *= ticktime;
   et /= 10000;
   printf (FORMAT, "average lock and unlock resource", et);
   KS_Sleep (250);
   KS_Elapse (&et);
      for (i = 0; i < 5000; i++)
   {
      Map1Memorymap.AllocW(&p);
      Map1Memorymap.Dealloc(&p);
   }
   et = KS_Elapse (&et);
   et *= ticktime;
   et /= 10000;
   printf (FORMAT, "average alloc and dealloc memory page", et);
   KS_Sleep (250);
   }
void hiqtask(void)
{
   int i;
   for (i = 0; i < 1000; i++)
      Demoqx1Queue.DequeueW (text2, 1);
   for (i = 0; i < 1000; i++)
      Demoqx1Queue.DequeueW (text2 ,4);
}

void himtask (void)
{
   int   i, size;
   KMessage Message;
   Message.tx_task = ANYTASK;
   Message.rx_data = text2;
   Message.size = 0;
   for (i = 0; i < 1000; i++)
      Mailbox1.ReceiveW (Message);
   for (size = 8; size <= 4096; size <<= 1)
   {
      Message.size = size;
      for (i = 0; i < 1000; i++)
      {
         Mailbox1.ReceiveW (Message);
      }
   }
}
```


test1.cpp

```
void histask (void)
{
   int i;
   K_SEMA slist [5];
   slist [0] = SEM1;
   slist [1] = SEM2;
   slist [2] = ENDLIST;
   slist [3] = ENDLIST;
   slist [4] = ENDLIST;
   for (i = 0; i < 1000; i++) Semaphore1.Wait();
   for (i = 0; i < 1000; i++) Semaphore1.WaitT(5000);
   for (i = 0; i < 1000; i++) KS_WaitM (slist);
   for (i = 0; i < 1000; i++) KS_WaitMT (slist, 5000);
   slist [2] = SEM3;
   for (i = 0; i < 1000; i++) KS_WaitM (slist);
   for (i = 0; i < 1000; i++) KS_WaitMT (slist, 5000);
   slist [3] = SEM4;
   for (i = 0; i < 1000; i++) KS_WaitM (slist);
   for (i = 0; i < 1000; i++) KS_WaitMT (slist, 5000);
}

void master (void)
{
int k;
puts ("Type '#' to start a benchmark sequence");
puts ("Demo can be terminated with ^C\n");
while (1)
{
k = server_pollkey ();
if (k) server_putchar (k);
if (k == '#') benchm ();
}
}
```

17.14. Traps and Pitfalls of C++

Caution is required when a developer wishes to use global or static variables that are instances of C++ classes. Initialisation of C++ objects that are not allocated on the stack is done before the main() routine is executed. The C++ compiler will generate initialisation code for all global or static C++ objects. This code will initilaise these C++ objects by calling their constructor (with the parameters supplied by the user).

C++ object initialisation code will thus run before the main() routine is executed, and thus also *before the kernel is initialised*. One may thus not use any kernel service calls in the constructors of objects that have global or static scope. If any C++ object initialisation requires calling kernel services,

then a separate initialisation member function must be added to the class. The programmer must then call this initialisation function on his objects *after* the kernel_init() call in the main() routine.

An example:

Wrong:

```
/* allocate a global object, and call its constructor.
Constructor calls some kernel services, based on the parameters
passed. */
MyClass aGlobalObject(param1, param2, param3);

int main ()
{
   netload ();
   kernel_init ();
   if (K_ThisNode == 0x00010000) KS_StartG (EXE_GROUP);
   kernel_idle ();
   return 0;
}
```

Correct:

```
/* allocate a global object, and call its default constructor.
Default constructor calls no kernel services. */
MyClass aGlobalObject;

int main ()
{
   netload ();
   kernel_init ();
   /* Explicitly call initialsation function that will call
   some kernel services, based on the parameters passed. */
   aGlobalObject.Init(param1, param2, param3);
   if (K_ThisNode == 0x00010000) KS_StartG (EXE_GROUP);
   kernel_idle ();
   return 0;
}
```


Virtuoso ™

The Virtual Single Processor Programming System

Covers :

Virtuoso Classico ™
Virtuoso Micro ™

Version 3.11

Part 3: Binding Manual

18. Virtuoso on the Analog Devices 21020 DSP

18.1. Virtuoso implementations on the 21020

At this moment, both VIRTUOSO MICRO as VIRTUOSO CLASSICO exists for the ADSP-21020. These implementations contain the microkernel and the nanokernel and are dedicated to single processor and multiprocessor systems. Until now, the implementation of VIRTUOSO CLASSICO for multiprocessor systems is restricted to communication by shared memory.

18.2. DSP-21020 chip architecture

This section contains a brief description of the ADSP-21020 processor architecture. It is not intended to be a replacement of the Processor's User Manual, but as a quick lookup for the application programmer. Detailed information can be found in the "ADSP-21020 User's Manual" from Analog Devices, Inc.

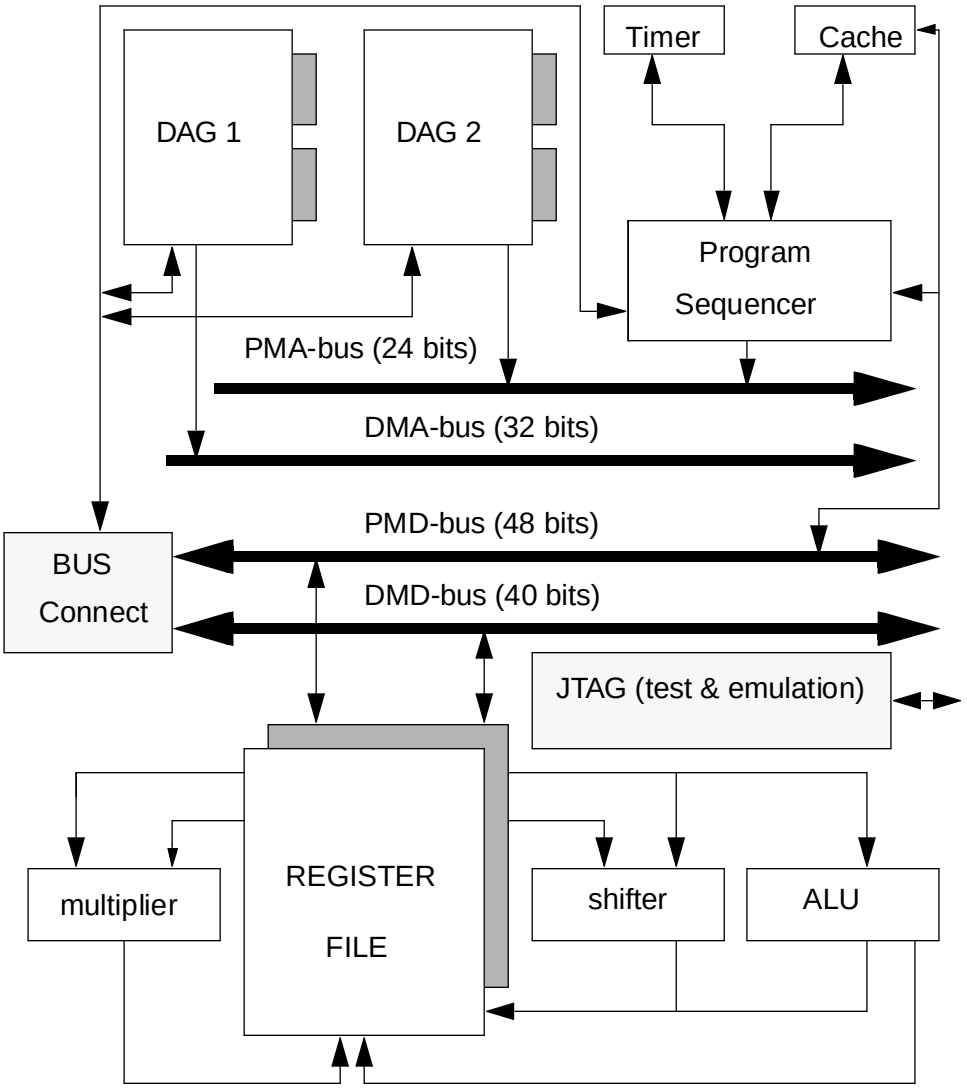
The ADSP-21020 is the first processor in the Analog Devices ADSP-21000 family of floating-point digital signal processors. The technological foundation for the ADSP-21020 is the powerful Harvard architecture. The processor has two distinct but similar memory interfaces: one for program memory, which contains both instructions and data, and one for data memory, which contains data only. Each address space on the ADSP-21020 can be divided into banks for selection. The program memory address space is divided into two banks. The data memory is divided into four banks. (see also chapter about the architecture file)

The computation units of the ADSP-21020 provide the numeric processing power for performing numerically operations. The 21020 contains three computation units:

- an arithmetic / logic unit (ALU)
- a multiplier
- a shifter

The computation units are arranged architecturally in parallel. The output of any computation unit may be the input of any computation unit on the next cycle. The computation units input data from and output data to a register file, that contains sixteen 40-bit registers and sixteen alternate registers. The register file is accessible to the ADSP-21020 program and data memory data buses for transferring data between the computation units and external memory or other parts of the 21020.

Both fixed-point and floating-point operations are supported. Floating-point data can be either 32 or 40 bits wide. Extended precision floating-point format (8 bits of exponent and 32 bits of mantissa) is selected if the RND32 bit in the MODE1 register is cleared. If this bit is set, the normal IEEE-precision is used (8 bits of exponent and 24 bits of mantissa). Fixed-point numbers are always represented in 32-bit format and occupy the 32 most significant bits in the 40-bit data field. They may be treated as fractional or integer numbers and as unsigned or twos-complement.



The ALU performs arithmetic operations on fixed-point or floating-point data and logical operations on fixed-point data. ALU-instructions are:

- floating-point (fixed-point) addition, substraction, add / substract, average
- floating-point manipulation: binary log, scale, mantissa
- fixed-point add with carry, substract with borrow, increment, decrement
- logical AND, OR, XOR, NOT
- functions: absolute value, pass, min, max, clip, compare
- format conversion
- reciprocal and reciprocal square root

The multiplier performs fixed-point or floating-point multiplication and fixed-point multiply / accumulate operations. Fixed-point multiply / accumulate operations may be performed with either cumulative addition or cumulative substraction. This can be accomplished through parallel operation of the ALU and the multiplier. The most important multiplier instructions are:

- floating-point (fixed-point) multiplication
- floating-point (fixed-point) multiply / accumulate with addition, rounding optional
- rounding result register
- saturating result register
- clearing result register

The shifter operates on 32-bit fixed-point operands. Shifter operations include:

- shifting and rotating bits
- bit manipulation: bit set, bit clear, bit test, ...

The register file provides the interface between the processor buses (DMD-bus and PMD-bus) and the computation units. It also provides temporary storage for operands and results. The register file consists of 16 primary and 16 alternate registers. All registers are 40 bits wide. The application developper can use the alternate register set to facilitate the context switching. The alternate registers can be activated by setting two bits (bit 7 and bit 10) in MODE1 register.

Managing the sequence of the program, is the job of the program sequencer. The program sequencer selects the address of the next instruction. It also performs some related functions:

- incrementing the fetch address
- maintaining stacks
- evaluating conditions

- decrementing loop counter
- calculating new addresses
- maintaining instruction cache
- handling interrupts

More information about the program sequencer architecture can be found in the ‘User’s Manual’ from Analog Devices, Inc.

The ADSP-21020 has also two data address generators (DAG1 & DAG2). The first data address generator produces 32-bit addresses for data memory, while the second data address generator produces 24-bit addresses for program memory. Each DAG contains four types of registers: index (I), modify (M), base (B) and lenght (L) registers. An I register acts as a pointer to memory, while the M-register controls the step size for advancing the pointer. B registers and L registers are only used for circular buffers. B holds the starting address, while L contains the length of the circular buffer. The application developper can also use the alternate registers, provided for each DAG. This facilitates context switching. Again, more information about that issue can be found in the Analog Devices ‘User’s Manual’.

The ADSP-21020 has a programmable interval timer that can generate periodic interrupts. Three registers control the timer period:

- TPERIOD : contains the timer period (32-bit register)
- TCOUNT: the counter register (32-bit register)
- MODE2: contains the timer enable bit (bit 5)

When the timer is enabled, TCOUNT is decremented each clock cycle. An interrupt is generated when TCOUNT reaches zero. Next, TCOUNT is reloaded with the value of TPERIOD.

The instruction cache is a 2-way cache for 32 instructions. The operation of the cache is transparant to the user. The processor caches only instructions that conflict with program memory data accesses. The cache is controlled by two bits in the MODE2 register:

- CADIS (bit 4): Cache disable bit
- CAFRZ (bit 19): Cache freeze bit

An IEEE JTAG boundary scan serial port provides both system test and on-chip emulation capabilities.

18.3. ADSP-21020 addressing modes

As already explained in the previous chapter, the 21020-processor has two

data address generators. This DAGs allow the processor to address memory indirectly. An instruction specifies a DAG register containing an address instead of the address value itself.

The first DAG is used in combination with the data memory, while the second DAG produces 24-bit addresses for program memory.

Other functions supported by the DAGs are:

- circular buffers, using the L-registers and B-registers
- DAG1 can support bit-reverse operations. This operation places the bits of an address in reverse order to form a new address.

There are two ways in which an I-register can be modified, using an M-register:+

- pre-modify without update: PM (M, I) & DM (M, I)
- post-modify: PM (I, M) & DM (I, M)

18.4. Special purpose registers on the ADSP-21020

The ADSP-21020 is provided with some special purpose registers. This paragraph contains a brief overview of these registers.

18.4.1. MODE1-register and MODE2-register

MODE1 and MODE2 are both 32-bit registers, that enable various operating modes of the ADSP-21020.

- MODE1:

Bit	Name	Definition
0		Reserved
1	BR0	Bit-reverse for I0
2	SRCU	Alternate register select
3	SRD1H	DAG1 alternate register select (7-4)
4	SRD1L	DAG1 alternate register select (3-0)
5	SRD2H	DAG2 alternate register select
6	SRD2L	DAG2 alternate register select
7	SRRFH	Register file alternate select
10	SRRFL	Register file alternate select
11	NESTM	Interrupt nesting enable
12	IRPTEN	Global interrupt enable

13	ALUSAT	Enable ALU saturation
14		Reserved
15	TRUNC	truncation / rounding
16	RND32	single / extended precision

- MODE2:

Bit	Name	Definition
0	IRQ0E	IRQ0 = edge / level sensitive
1	IRQ1E	IRQ1 = edge / level sensitive
2	IRQ2E	IRQ2 = edge / level sensitive
3	IRQ3E	IRQ3 = edge / level sensitive
4	CADIS	Cache disable
5	TIMEN	Timer enable
6-14		Reserved
15	FLAG0O	FLAG0
16	FLAG1O	FLAG1
17	FLAG2O	FLAG2
18	FLAG3O	FLAG3
19	CAFRZ	Cache freeze
20-31		Reserved

18.4.2. Arithmetic status register (ASTAT)

ASTAT is a 32-bit register, in which the bits can be set as a result of an ALU operation. An overview of the bits in the ASTAT-register is given below:

Bit	Name	Definition
0	AZ	ALU result zero or underflow
1	AV	ALU overflow
2	AN	ALU result negative
3	AC	ALU fixed-point carry
4	AS	ALU X input sign
5	AI	ALU floating-point invalid operation
6	MN	multiplier result negative
7	MV	multiplier overflow
8	MU	multiplier floating-point underflow
9	MI	multiplier floating-point invalid oper.

10	AF	ALU floating-point operation
11	SV	shifter overflow
12	SZ	shifter result zero
13	SS	shifter input sign
14-17		Reserved
18	BTF	bit test flag for system registers (RO)
19	FLG0	FLAG0 value
20	FLG1	FLAG1 value
21	FLG2	FLAG2 value
22	FLG3	FLAG3 value
23		Reserved
24-31		CACC bits

18.4.3. Sticky arithmetic status register (STKY)

STKY is also a 32-bit register. The bits can be set by ALU-operations. By reading this register, the service routine for one of these interrupts can determine which condition caused the interrupt. The routine also has to clear the STKY bit so that the interrupt is not still active after the service routine is done. An overview of the bits is given below:

Bit	Name	Definition
0	AUS	ALU floating-point underflow
1	AVS	ALU floating-point overflow
2	AOS	ALU fixed-point overflow
3-4		Reserved
5	AIS	ALU floating-point invalid operation
6	MOS	multiplier fixed-point overflow
7	MVS	multiplier floating-point overflow
8	MUS	multiplier floating-point underflow
9	MIS	multiplier floating-point invalid oper.
10-16		Reserved
17	CB7S	DAG1 circular buffer 7 overflow
18	CB15S	DAG2 circular buffer 15 overflow
19-20		Reserved
21	PCFL	PC stack full
22	PCEM	PC stack empty

23	SSOV	status stack overflow
24	SSEM	status stack empty
25	LSOV	loop stack overflow
26	LSEM	loop stack empty
27-31		Reserved

18.4.4. Interrupt latch (IRPTL) and Interrupt Mask (IMASK)

These two registers are used in combination with interrupts. Both are 32-bit registers. Each bit in the registers is representing an interrupt. The interrupt bits are ordered by priority (highest ->lowest). Setting bits in the IRPTL activates an interrupt. Example:

```
Bit set IRPTL SFT0I; /* activates software interrupt */
Bit clr IRPTL SFTOI; /* clears a software interrupt */
```

An interrupt can also be masked. Masked means that the interrupt is disabled. Interrupts that are masked are still latched, so that if the interrupt is later unmasked, it is processed. Example:

```
Imask: 1 = unmasked (enabled), 0 = masked (disabled)
/* Interrupts 0 and 1 are not maskable */
```

Bit (IR#)	Address	Function
0	0x00	Reserved for emulation
1	0x08	Reset
2	0x10	Reserved
3	0x18	Status stack / loop stack / PC stack
4	0x20	High priority timer
5	0x28	IRQ3 asserted
6	0x30	IRQ2 asserted
7	0x38	IRQ1 asserted
8	0x40	IRQ0 asserted
9	0x48	Reserved
10	0x50	Reserved
11	0x58	Circular buffer 7 overflow interrupt
12	0x60	Circular buffer 15 overflow interrupt
13		Reserved
14	0x70	Low priority timer
15	0x78	Fixed-point overflow

16	0x80	Floating-point overflow exception
17	0x88	Floating-point underflow exception
18	0x90	Floating-point invalid exception
19-23	0x98-0xb8	Reserved
24-31	0xc0-0xf8	User software interrupts (0-7)

18.4.5. Program memory / Data memory interface control registers

- PMWAIT: 14-bit register, that controls the wait states and the wait mode of the program memory banks. Bits 12-10 set also the program memory page size.
- DMWAIT: 24-bit register, that controls the wait states and the wait mode of the data memory banks. Bits 22-20 set also the data memory page size.
- PMBANKx: defines the begin of the program memory banks.
- DMBANKx: defines the begin of the data memory banks.

18.4.6. PC stack (PCSTK) and PC stack pointer (PCSTKP)

The 21020-ADSP is provided with two registers that control the stack management:

- PCSTK: top of PC stack (24 bits wide and 20 deep)
- PCSTKP: PC stack pointer (5 bits)

PC stack holds return addresses for subroutines and interrupt services and top-of-loop addresses for loops. The PC stack is popped during return from interrupts (RTI), return from subroutine (RTS) and termination of loops. When the PC stack is full, the ‘full’ flag is set in the STKY-register so that an interrupt can be generated.

The PCSTKP is a 5-bit readable and writeable register that contains the address of the top of the PC stack.

Note: This PC-stack is not very useful in a multitasking environment. If instructions that invoke the PC-stack are used, one has to save that PC-stack when a task is swapped out. (see further)

18.4.7. Status Stack

The status stack is five levels deep and is used for interrupt servicing. The ADSP-21020 automatically saves and restores the status and mode regis-

ters of the interrupted program. The four external interrupts and the timer interrupt automatically push ASTAT and MODE1 onto the status stack. These registers are automatically popped from the status stack by the interrupt return instruction (RTI).

18.4.8. USTAT

The User Status Registers, USTAT1 and USTAT2, are general-purpose registers that can be used for user status flags. These system registers are set and tested using system instructions.

18.5. Relevant documentation

1. “ADSP-21020 User’s Manual”, Analog Devices, Inc., 1991
2. “ADSP-21000 Family C Tools Manual”, Analog Devices, Inc., 1993
3. “ADSP-21000 Family Assembler Tools & Simulator Manual”, Analog Devices, Inc., 1993
4. “ADSP-21000 Family C Runtime Library Manual”, Analog Devices, Inc., 1993

18.6. Version of the compiler

Analog Devices, Inc. has different versions of the G21k compiler. We used version 3.0.

18.7. Runtime Environment

This section contains following topics:

- Data types
- Architecture file
- Runtime header

18.7.1. Data types

The ADSP-21020 can process 32-bit operands, with provisions for certain 40-bit operations. The arithmetic types supported directly are listed below. All other arithmetic data types are mapped onto these types.

- type float: IEEE-754 standard single-precision floating-point. It has a 24-bit signed magnitude fraction and a 8-bit exponent. Operations on double-precision numbers are calculated with software emulation.
- type int: a fixed-point 32-bit two’s complement number

- type complex: this type is a numerical C extension to the Standard C. A complex number is seen as two ‘float’ or ‘int’ numbers.

Underlying types are:

C data type	representation
int	32-bit two’s complement number
long int	32-bit two’s complement number
short int	32-bit two’s complement number
unsigned int	32-bit unsigned magnitude number
unsigned long int	32-bit unsigned magnitude number
char	32-bit two’s complement number
unsigned char	32-bit unsigned magnitude number
float	32-bit IEEE single-precision number
double	64-bit IEEE double-precision number
long double	64-bit IEEE double-precision number
complex int	two 32-bit two’s complement numbers
complex float	two 32-bit IEEE single-precision numbers

18.7.2. The Architecture file

The architecture file describes the memory configuration. This configuration is read by the compiler and the linker to determine the memory specification of the target system. For example, you may specify the memory size, memory type, and the number of wait states used in the different banks of data and program memory. The architecture file uses two directives to handle these memory specifications:

- .SEGMENT: tells the linker, which memory segments may be used.
- .BANK: specifies the physical memory on the target board.

The architecture file also uses two other directives, which identify your system:

- .SYSTEM: name of your ADSP-21020 system.
- .PROCESSOR: defines the processor_type (21020 / 2106x)

More information about writing and understanding architecture files can be found in the “ ADSP-21020 Family Assembler Tools and Simulator Manual “ and in the “ ADSP-21020 Family C Tools Manual ” from Analog Devices.

18.7.3. Runtime header (interrupt table)

A portion of memory is reserved for the interrupt table. The interrupt table is where the linker puts the code in the runtime header: “020_hdr.obj”. This runtime header is automatically linked in when you invoke the compiler.

18.8. Assembly language interface

This paragraph shortly shows how to interface assembly language code with C code. It gives an overview of the most important things, you must keep in mind, when you are writing C-callable assembly functions.

There are two registers, called the stack pointer and the frame pointer, that are used to manipulate the runtime stack:

- i7: points to the top of the stack
- i6: points to the start of the frame for the current function

Register i13 is used by the C calling protocol to contain the function return address. When this register is used later on, it must be placed on the stack on function entry.

The compiler makes also some assumptions about how functions treat registers. There are two classes of registers:

- Compiler registers: these registers are preserved across function calls.
- Scratch registers: these registers are not preserved across function calls. (See User’s Manual for detailled list)

Registers may be saved by pushing them on the stack. You can use the following instruction:

- dm (i7, m7) = r3; places r3 onto the stack and decrements the pointer.
- r3 = dm (1, i7); reads the stack. The stackpointer is not incremented.
- modify (i7, 1); increments the stack.

In the C environment, arguments are passed to functions by placing them in registers or on the stack, according to following rules:

1. At most three arguments are passed in registers. (R4, R8, R12)
2. Once one argument has been passed on the stack, all remaining arguments are on the stack.
3. All values wider than 32 bits are passed on the stack.

The return value of the function must be placed in the appropriate registers. If a single word is being returned, it must be returned in register R0. If a two

word value is being returned, it must be returned in registers R0 and R1.

The calling protocol in a C environment involves several steps:

1. The arguments to the function must be passed.
2. The return address must be loaded into register i13.
3. The frame pointer must be adjusted. The current function's frame pointer, I6 is written into R2, and the current stack pointer, i7, is written into i6 to create the called function's frame pointer.
4. When the function returns, it's necessary to adjust the stack pointer.
5. In order for C functions to link with assembly functions, use the .global and .extern assembly language directives. The name has to start with an underscore in assembly.

There is also a possibility to use in-line assembly into your C code. This is done using the asm() construct. Example:

```
asm (“ bit set mode1 IRPTEN;”);
/* enables interrupts */
```

To conclude, an example of a C-callable function is given.

```
/* int add (int a, int b) */
.segment /pm seg_pmco;
.global _add;

_add:
 dm(i7,5) = r2 /* stores old frame pointer */
 dm(-1,i6) = i13; /* stores return address */
 dm(-2,i6) = r4;
 dm(-3,i6) = r8;
 dm(-4,i6) = r0;
 r4 = r4+r8;
 r0 = r4;
 jump(m14,i13)(DB); /* delayed jump to return address */
 i7 = i6;
 i6 = dm(0,i6); /* adjusts stack */
```

```
.endseg;
```

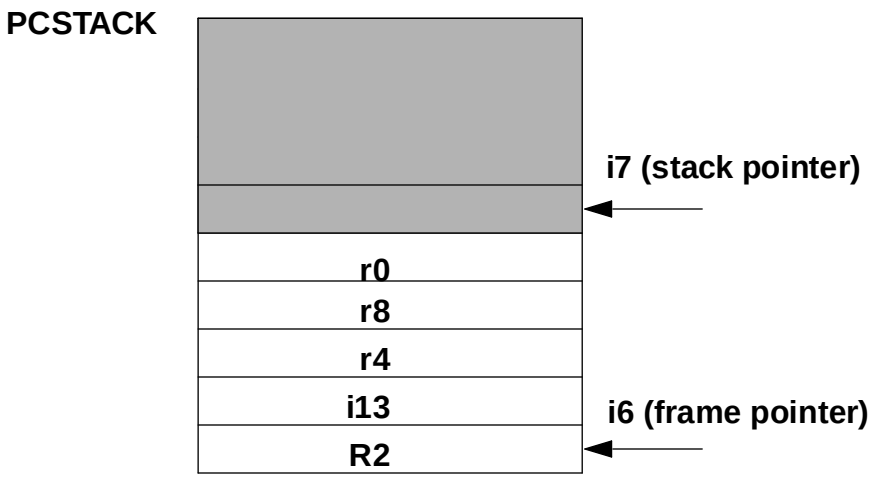


FIGURE 14 Stack usage

18.9. Developing ISR routines on the 21020

18.9.1. Installing an ISR routine

The best place to install and enable an interrupt service routine (ISR), is in the `main()`function, where predefined drivers, like the driver for the timer interrupt, are installed and enabled.

It is possible that additional initialisation of registers and/or peripheral I/O has to be done. The best way is to write a C-callable procedure, that does the necessary additional initialisations, and call this procedure after the call to `KS_EnableISR()`. An example of this method is the installation of the timer ISR in procedure `main()`:

```
#include “iface.h”
extern void timer0_irqh (void);
extern void timer0_init (void);
...
int main (void)
{
...
init_server();
KS_EnableISR (4, timer0_irqh);
timer0_init();
...
}
```

Note: In VIRTUOSO CLASSICO, there is a function provided that does the work: ‘timer0_drv().’ This function installs and initialises the timer. The host interrupt service routine is installed and enabled by using the function init_server().

18.9.2. Writing an ISR routine

A. VIRTUOSO MICRO

It is very easy to develop ISR routines for the ADSP-21020. You just have to keep the right things in mind. An example of an ISR is given and explained below:

example 1: signalling ISR

```
#include <def21020.h>
#define HOSTSIG 8 /* defines event number for host */

.segment /pm seg_pmco;
.extern _kernel_sign_entry; /* external function */
.global _rxhost_irqh; /* makes it C callable */
.endseg;
.segment /pm seg_pmco;
_rxhost_irqh:
 dm(i7,m7) = ASTAT;
 dm(i7,m7) = MODE1;
 dm(i7,m7) = r2;
 dm(i7,m7) = r4;
 r4 = i4;
 dm(i7,m7) = r4;
 jump _kernel_sign_entry (DB);
 r4 = HOSTSIG; (1)
 bit CLR MODE1 IRPTEN; (2)
```

First, ASTAT and MODE1 have to be pushed on the stack. This is to make sure that these registers are correctly saved in all cases. Which registers have to be preserved by an ISR depends on the class of the ISR and on which registers are used in the ISR. If the ISR stands on its own (no signal is made to the kernel), only those registers must be preserved that are used by the ISR. In the case the ISR gives a signal to the kernel, all registers that are used by the ISR must be preserved, except the registers R2, R4, I4: these registers must always be saved at the start of a signalling ISR, regardless if they are used by the ISR or not, because the kernel is relying on the fact that they are saved. The ISR ends with a kernel_sign_entry. The kernel expects the event signal number in register R4 . In this example, the event (8) is signalled to the kernel, for further processing. At that moment interrupts are disabled. Note also the use of the delayed branch. Instructions (1) and (2) are executed before the jump.

example 2: non-signalling ISR (using R2, R4)

```
#include <def21020.h>
#define HOSTSIG 8 /* defines event number for host */
.segment /pm seg_pmco;
.global _irq_service; /* makes it C callable */
.endseg;
.segment /pm seg_pmco;
_irq_service:
```



```
 dm(i7,m7) = ASTAT;
 dm(i7,m7) = MODE1;
 dm(i7,m7) = r2;
 dm(i7,m7) = r4;
 < body of ISR >
r4 = dm(1,i7);
r2 = dm(2,i7);
MODE1 = dm(3,i7);
ASTAT = dm(4,i7);
modify (i7,4);
RTI;
```

Note also that for this release of Virtuoso Micro for the ADSP-21020, it is needed to disable interrupts before signalling the kernel. When interrupts are enabled, it is possible that an interrupt of higher priority interrupts one of lower priority. Entering the kernel is a critical section and may not be interrupted. Therefore, at that moment interrupts must be disabled. Because, it is advised to disable interrupts in a period as short as possible, it must be done just before the jump. The kernel will re-enable interrupts as soon as the critical section is executed.

B. VIRTUOSO CLASSICO

There are two differences between writing IRQ's for VIRTUOSO MICRO and for VIRTUOSO CLASSICO:

1. different set of registers that must be saved.
2. the way, the kernel is entered.

an example: host-interrupt service routine

```
#include <def21020.h>
#include <macro.h>     /* macro's for nanokernel */

.segment /pm seg_pmco;
.global _rxhost_irqh;
.endseg;

.segment /dm seg_dmda;
.var HOST_EVENT = 8;
.extern _K_ArgsP; /* channel for the kernel */
.endseg;

.segment /pm seg_pmco;
```

```
_rxhost_irqh:
/* registers that MUST be saved by the IRQ */
dm(i7,m7) = MODE1;
dm(i7,m7) = ASTAT;
dm(i7,m7) = r0;
dm(i7,m7) = r1;
dm(i7,m7) = r2;
dm(i7,m7) = r4;
dm(i7,m7) = r8;
dm(i7,m7) = r12;
r4 = i4;
dm(i7,m7) = r4;
dm(i7,m7) = i12;
/* registers that MUST be saved by the IRQ */
i4 = dm(_K_ArgsP);
r2 = dm(HOST_EVENT);
PRHI_PSH; /* pushes event on kernel-channel */
ENDISR1; /* ends the interrupt routine */
endseg;
```

For more information: see also the chapter upon the nanokernel.

18.9.3. Alphabetical list of ISR related services

1. _kernel_sign_entry : entering the kernel from within an ISR
2. KS_EventW (int IRQ) : waits on an interrupt at the task level
3. KS_EnableISR(int IRQ, void (*ISR) (void))): installs the ISR
4. KS_DisableISR(int IRQ): disables the ISR

See part 2 for details.

18.10. The nanokernel on the 21020

18.10.1. Introduction

The nanokernel provides the lowest level of functionality in the Virtuoso system. It is designed to perform extremely fast communication and context swapping for a number of processes. It also provides the entry points necessary to integrate interrupt handlers with the rest of the system. The prices to pay for speed is that the nanokernel processes and interrupt handlers must obey very strict rules regarding to their use of CPU registers and the way

they interact with each other.

From the point of view of the nanokernel, an application program consists of a collection of three types code modules:

- a single low priority process (PRLO-process).
- any number of high priority processes (PRHI-process).
- any number of interrupt handlers.

It is important to understand what exactly is meant by a process. A process is a thread of execution that has both an identity and a private workspace. These two properties (logically equivalent) make it possible for a process to be swapped out, and wait for an external event while another process is allowed to continue. Interrupt handlers in contrast, do not have a private workspace.

The PRHI processes are scheduled in strict FIFO order, and must observe the special register conventions mentioned above. The PRLO process is assumed to be a C function (using the compiler register conventions), and must always be ready to execute. You can compare it with the Idle-process of the microkernel.

All communication inside the nanokernel is performed using channels. Several types exit. The simplest type is used for synchronization and corresponds to a counting semaphore. The other types are used for data transfer. The possiblity is provided that a user can add his own channel types.

The microkernel, who manages the tasks is build as an application on top of the nanokernel. The main component is a PRHI process that executes commands it receives from a channel. When the channel is empty, the microkernel looks for the next task to run, replaces the nanokernel IDLE-process by that task and performs the additional register swappings required for C tasks.

The nanokernel is not ‘aware‘ of the manipulations performed by the microkernel. As far as it concerned, there is only one PRLO-process, which it executes whenever no PRHI-process is ready to continue. This makes it possible to use the nanokernel on its own.

18.10.2. Internal data structures

The user does not normally need to access the internal data structures used by the nanokernel. The documentation in this section is provided only for a better understanding of how the nanokernel operates.

A process is represented by a pointer to a Process Control Structure (PCS). For PRHI processes, the PCS occupies the first six words of its stack. Two

entries are placed at the top by reason of a decrementing stackpointer. A static PCS is used for the Idle-process. More details on the PCS will be introduced in the section on process management.

A channel is represented by a pointer to a Channel Data Structure (CDS). The first word of a CDS is a pointer to the PCS of a process waiting on that channel, or NULL. Other fields depend on the type of the channel and will be introduced in the section on nanokernel communications.

The following static variables are used by the nanokernel to keep track of the state of the system:

NANOK_PRHI: Pointer to the PCS of the current PRHI-process, or NULL if there is none.

NANOK_HEAD: Head pointer for a linked list of PRHI-processes that are ready to run. When the current PRLO-process is swapped out, the PRHI-process at the head of the list is removed, and becomes the current process.

NANOK_TAIL: Tail pointer for a linked list of PRHI-processes that are ready to run. When a process becomes ready to execute, it is added to the tail of the list.

NANOK_PRLO: Pointer to the PCS of the PRLO-process. This is a constant as far as the nanokernel is concerned. The microkernel modifies this pointer.

NANOK_CRIT: This is the number of interrupt handlers running with global interrupts enabled that are not yet terminated. The process swapping is disabled while this value is not zero. On the 21020, the return address of the interrupt is stacked on the PC-stack. So, we do not need an extra variable for this purpose. PCSTKP equals NANOK_CRIT.

Symbolic constants for accessing kernel variables and elements of a PCS are defined in the file: ‘nanok.h’

18.10.3. Process managment.

The nanokernel variable are initialized as follows:

```
NANOK_PRHI = 0;
NANOK_HEAD = 0;
NANOK_TAIL = &(NANOK_HEAD)
NANOK_PRLO = &(PCS for IDLE process)
NANOK_CRIT = PCSTKP = 0;
```

This means that when an application is started, the idle-process of the

nanokernel will start.

In the current version of the nanokernel, all PRHI-processes must be created and started by the PRLO-process. Three steps are required to create a process:

- create a stack for the process.
- initialise the PCS
- start the process

The stack can be placed anywhere in memory. It can be a C-array of integers, a memory block allocated by malloc.

The function _init_process (stack, stacksize, entry, i1, i2) is used to initialize the PCS. It writes the following values to the first 10 words of the stack:

```
PR_LINK : link pointer
PR_SSTP: saved stack pointer
PR_PI0: saved i0
PR_PI1: saved i1
PR_PI2: saved i2
PR_MODE1: saved MODE1-register
PR_ASTAT: saved ASTAT-register
PR_BITS: flags
PR_PEND: pointer to terminate code
PR_PRUN: pointer to entry point
```

Calling _start_process (stack) starts the process. As the caller is the PRLO-process, there can be no other PRHI process and the new process will start execution immediately. Control returns to the caller when the new process terminates or deschedules by waiting on a channel.

The first time a PRHI process is swapped in, it continues from its entry point. The stack pointer will point to the PR_PEND field in the PCS. A process terminates by returning to the address in this field. The code at NANOK_TRMP invokes the nanokernel swapper to switch to the next process. To restart a terminated process, repeat the calls to _init_process() and _start_process().

When a PRHI process is swapped in, i0 points to the start of the PCS. A process can create local variables by incrementing the initial stack pointer by the number of words required.

Note: On the 21020, the stacksize is also a parameter. This is because of the fact that the stackpointer is moving from the top to the bottom of the stack.

18.10.4. Nanokernel communications

A channel type is defined by a data structure and a number of nanokernel services that operate on it. Each instance of the data structure is called a channel. Channels provide both porcess synchronization and data communication.

The nanokernel does not itself use or create channels. However, the services that operate on channels should be considered part of the nanokernel, as they may modify the process FIFO or invoke the nanokernel swapper.

All channels have an internal state. What exactly is represented by the state depends on the type of the channel - typically this will be the occurence of an event or the availability of data. An operation on a channel can consist of any combination of the following action types:

```
Wait: The process is said to 'wait on a channel'
Signal: Signalling action. This action can reschedule a
        process.
Test and modify: modifies or tests the state of a
        channel.
```

Three channel types are predefined in the current nanokernel implementation. It is possible to create new channel types if necessary; an example will be given at the end of this chapter. A full description of the nanokernel services for each of these channel types can be found in the alphabetical list in the next chapter.

18.10.5. C_CHAN - counting channel

This is an implementation of a counting semaphore. It is typically used by interrupt handlers to reschedule a process that was waiting for the interrupt. The C_CHAN structure has two fields:

```
CH_PROC: pointer to the PCS of the waiting process or
        NULL
CH_NSIG: event counter
```

Two nanokernel services are available for this channel type:

```
PRHI_WAIT: waiting action
PRHI_SIG: signalling action
```


18.10.6. L_CHAN - List channel

This type of channel maintains a linked list of memory blocks, using the first word in each block as a link pointer. The microkernel uses this type to implement its free list of command packets, data packets and timers. If used for data communication, it behaves as a LIFO buffer.

The L_CHAN structure has two fields:

```
CH_PROC: pointer to the PCS of a waiting process or NULL
CH_LIST: pointer to the first element of the linked list
        or NULL
```

The nanokernel services that operate on this type are:

```
PRHI_GETW: wait action
PRHI_GET: test and modify action
PRHI_PUT: signal action
```

18.10.7. S_CHAN - Stack channel

This type of channel uses a memory block as a data stack. The microkernel uses a stack channel to input commands from tasks and the network drivers, and to receive events from interrupt handlers.

The S_CHAN structure has three fields:

```
CH_PROC: pointer to the PCS of a waiting process or NULL
CH_BASE: pointer to the base of the stack
CH_NEXT: pointer to the next free word on the stack
```

The nanokernel services that operate on this type are:

```
PRHI_POPW: wait action
PRHI_POP: test and modify
PRHI_PSH: signal action
```

18.10.8. REGISTER CONVENTIONS

In order to understand the register conventions adopted by the Virtuoso nanokernel, the following register sets should be introduced:

Csave: r3, r5, r6, r7, r9, r10, r11, r13, r14, r15, mrf, i0, i1, i2, i3, i5, i8, i9, i10, i11, i14, i15, m0, m1, m2, m3, m8, m9, m10, m11, mrf, mrb, mode1, mode2, ustat1, ustat2,

Cfree: r0, r1, r2, r4, r8, r12, i4, i12, m4, m12

Sysset: PCSTKP, IRPTL, IMASK, ...

Nswap: r3, r5, r6, r7, r9, r10, r11, r13, r14, r15, i3, i5, i8, i9, i10, i11, i14, i15, m0, m1, m2, m3, m8, m9, m10, m11, USTAT1, USTAT2, mr0f, mr1f, mr2f, mr0b, mr1b, mr2b, MODE2

Intset: MODE1, ASTAT, r0, r1, r2, r4, r8, r12, i4, i12

The Csave and Cfree sets are defined by the procedure calling standard of the C-compiler. Csave is the set of registers that are preserved across a subroutine call - if a function uses any of these, it must restore the initial value on return. Cfree is the set of registers that are freely available to all functions - the caller of a subroutine is responsible for preserving them if necessary. The definition of these two sets largely determine what the microkernel is expected to do when swapping tasks. When a task is swapped out as a result of calling a kernel service (which to the task is just a C function), only the Csave set need be saved. When a task is preempted (which means that an interrupt handler has woken up the kernel), the Cfree set must be saved as well. Actually, since most of the microkernel is written in C, the Cfree set must be saved before the actual service requested by the interrupt handler is called.

Note: ST is included in the Cfree set because it contains the flags tested by the conditional instructions (bits 0-6). Other bits in ST have system control functions, and should be treated as part of Sysset.

The Sysset register are used for system and periperhal control only. They are never swapped, and shoul be regarded as global resources. Only very low level routines (such as hardware drivers) will ever need to access these registers.

The INTSET registers are those that must have been pushed on the stack when an interrupt handler terminates and wakes up the kernel by calling one of the ENDISR services (this is discussed in more detail in the section on interrupt handling below). At that point, the nanokernel needs some registers to work with. It would be a waste of time to pop all registers saved by the ISR, only to have to push them again when entering the kernel.

The registers in Nswap are saved and restored by the nanokernel when swapping processes. For the PRLO process (assumed to be a C-function, using i0 as its frame pointer) the nanokernel will save and restore i0 in the normal way. When a PRHI-process is swapped in, i0 will be set to point to its process control structure. A PRHI-process can use i0 to access local variables created in its workspace, and should normally not modify this register.

If it does, the initial value can always be reloaded from NANOK_PRHI. I0 must point to the PCS whenever the process calls a nanokernel service and when it terminates.

The Nswap register set is always available, but note the special use of i0.

If a PRHI process is swapped in as the result of a C-function call by the PRLO-process, then the Cfree set is available for use by the PRHI process. This means that the process can safely call any C-function. It should of course save those registers in Cfree that it wants to preserve across the call.

If a PRHI process is swapped in as the result of an interrupt handler calling an ENDISR service, then the INTSET registers are available to the PRHI-process. Before calling a C-function, the process must save the set Cfree-int-set, and it must restore the same registers before it is swapped out (this is always possible, since a PRHI-process is never preempted).

18.10.9. Interrupt handling

In the Virtuoso system model, interrupt handlers are the interface between asynchronous events and the processes that are waiting for then. To be useful, most interrupt handlers will have to interact with the rest of the system at some time. Using flags to be ‘polled’ by the foreground process is usually not an acceptable practice in a real-time system. This method introduces a ‘superloop’ structure into the application, with all its inherent problems.

In a system using the nanokernel, interrupt handlers can communicate with processes using the same channel operations that are available to processes. However, there are some restrictions.

In contrast to a process, an interrupt service routine does not have a private workspace, it executes on the stack of whatever process was interrupted. An ISR on the 21020 can be interrupted by an ISR of higher priority. So, any number of interrupt handlers can be piled on top of each other on the same stack, owned by the current process. This has some important consequences:

1. If an ISR calls a channel service that has a signal action, any process swap that results from this call must be delayed until all interrupt handlers have terminated. These implies that only the PRHI_type of channel operations can be used, as these do not invoke the swapper for a signal action (there is no need to swap, as the caller already has the highest priority). When the last stacked interrupt terminates, the swapper must be called to verify if a swap from the PRLO-process to a PRHI-process is necessary.
2. An ISR must never call any channel service that has a wait action.

Doing so would also block all other interrupt handlers that are stacked below it, as well as the current process. Another way of seeing this is that an ISR cannot wait for something because it doesn't have a separate identity - the producer of the external event (another ISR) has no means of representing who is waiting for it.

Note: The 21020 is provided with a system stack. When an external or a timer interrupt arrives, MODE1 and ASTAT are pushed on that system stack. The pull-operation is performed by the RTI. In our version of the kernel, MODE1 and ASTAT are always pushed on the stack of the task. This is done to be sure that they are saved.

18.10.10. The ISR-level

The 21020 has only one ISR-level. When entering the interrupt handler, global interrupts are enabled. An interrupt of higher priority can interrupt an interrupt of lower priority.

An interrupt ends by the nanokernel function 'ENDISR1'. At that point, the nanokernel will verify if a process swap is required and allowed. The condition tested is the logical AND of:

- NANOK_PRHI = 0; /* The current process is a PRLO-process */
- NANOK_HEAD != 0; /* There is a PRHI-process */
- NANOK_CRIT = 0; /* There are no nested interrupts anymore */

In case of a swap, the interrupt is finished and the PRHI-process is swapped in. If there are nested interrupts, first all interrupts are finished.

See also the chapter about ' writing IRQ's for VIRTUOSO CLASSICO and MICRO'.

18.10.11. Communicating with the microkernel

As mentioned before, the VIRTUOSO microkernel is implemented as a PRHI-process. It uses a single stack based channel to receive commands from the tasks, the network drivers, other PRHI-processes and interrupt handlers. A pointer to this channel is exported in the C-variable K_ArgsP.

Two types of data can be pushed onto this channel:

1. Small integers (0-63) are interpreted as events. Events are simple binary signals that a task can wait for using the KS_EventW()-service. Most events will be generated by interrupt handlers and driver processes. For the 21020 version, event numbers have been assigned as follows:

- 0-31: all interrupts provided by the 21020.
- 48 : timer event.
- rest: are free.

2. All other values pushed onto the microkernel input channel are interpreted as a pointer to a command packet. Command packets are the primary form of communication used within the Virtuoso system. They are used by the tasks to request microkernel services, sent across the Virtuoso network to implement remote kernel calls, and put on waiting lists to represent a task that is waiting for something. It is outside the scope of this manual to present a complete description of the command packet data format. The basic structures and the command codes are defined in K_STRUCT.H

The microkernel maintains a list of free command packets, implemented as a list based channel. A pointer to this channel is exported in the C variable K_ArgsFreeP. Other PRHI-processes can get command packets from this pool, but they must never wait on the channel (i.e. always use PRHI_GET). If the list is empty, correct behavior is to call YIELD and try again later.

In the Virtuoso network, the Srce field of a command packet identifies the sending node, and it is used as a return path for reply messages. The same field also has a secondary function: since all packets sent or received through the network are allocated from the K_ArgsFree list, they should be deallocated after use. The network transmitters always free a packet after it has been sent. The microkernel deallocates a packet if the Scre field is not zero. Consequently, command packets not allocated from the free list must have their Scre field set to zero to prevent deallocation.

Note: we are aware of the fact that this logic is a bit confusing. Future versions of the microkernel will probably use a separate flag to indicate if a packet was dynamically allocated.

Interrupt handlers and PRHI processes can request a microkernel service by building a command packet, and pushing a pointer to it on the microkernel input channel. The only services that can be safely called are the equivalents of KS_Signal and KS_SignalM. Also note that using events will be faster than signals.

The code fragments below show how to perform a KS_Signal() or KS_SignalM() call from within the ISR. In this example the command packet is created and initialized in C, but the same thing could be done entirely in assembly language

The function ‘install_my_isr()‘ is called to initialize the command packet and install the ISR:

```
K_ARGS CP1, *CP1P;
K_SEMA SLIST1 [] = {SEMA1, SEMA2, ..., ENDLIST};
extern void my_isr (void);
void install_my_isr(...)
{
 ...
 /* create a pointer to the command packet */
CP1P = &CP1;
/* initialize CP1 for a KS_Signal (SEMA1) service */
CP1.Srce = 0;
CP1.Comm = SIGNALS;
CP1.Args.s1.sema = SEMA1;
/* or for a KS_SignalM (SLIST1) service */
CP1.Scre = 0;
CP1.Comm = SIGNALM;
CP1.Args.s1.list = SLIST1;
/* install the ISR */
KS_EnableISR (..., my_isr);
...
}
For the ISR, something like the code listed below will be
        required:
.extern _CP1P;
.extern _K_ArgsP;
.global _my_isr
...
_my_isr:
...
i4 = dm(_K_ArgsP); i4 contains pointer to channel
r2 = dm(_CP1P); r2 contains data
PRHI_PSH; signals semaphore
...
```

18.10.12. Virtuoso drivers on the 21020

Drivers are the interface between the processor and peripheral hardware, and the application program. They normally serve two purposes: data-communication and synchronization. As polling is not a recommended practice in a real-time system, most drivers will use interrupts in one way or another.

The virtuoso system does not provide a standard interface to drivers - this

allows the application writer to optimize this important part of their implementation. Some basic services, that will be required for almost all drivers, are provided. Most low-level details have already been described in the previous sections on interrupt handling and communication with the microkernel. At the higher level, a typical driver can usually be divided into three functional parts:

1. The first component is a function to install the driver. This should initialize the hardware and any data structures used, and install interrupt handlers for the driver. A call to this function is usually placed inside a driver statement in the system definition file. The SYSGEN-utility copies this call into a function named init_drivers() it generates in the node#.c files. The init_drivers() subroutine is called by kernel_init() just before it returns.

2. Most drivers will provide one or more subroutines that can be called from the task level, and that implement the actual functionality of the driver. At some point, these functions will call KS_EventW() or KS_Wait() to make the calling task wait for the completion of the driver action.

3. One or more interrupt handlers are required to generate the events or signals waited for by these subroutines.

In the simplest case, the only actions required from the ISR will be to service the hardware and to reschedule a waiting task, and all data handling and protocol implementation can be done at the task level. This method can be used if the interrupt frequency is not too high (< 1000Hz).

For higher data rates, some of the task code should be moved to the ISR, in order to reduce the number of task swaps. In most cases, the actions, required from the interrupt handler will not be the same for each interrupt, and some form of state machine will have to be implemented into the ISR.

If the number of possible states grows, it is often much easier to use one or more PRHI-processes to implement the protocol. Processes can wait for interrupts at any number of places in their code, and each of these points represents a state of the system. As an example, the virtuoso network driver have been designed using this method.

19. Alphabetical List of nanokernel entry points

In the pages to follow, all Virtuoso nanokernel entry points are listed in alphabetical order. Most of these are 21020-call, some are C callable.

- BRIEF . . . . . Brief functional description
- CLASS . . . . . One of the Virtuoso nanokernel service classes of which it is a member.
- SYNOPSIS . . . . . The ANSI C prototype (C callable), or

  Assembly language calling sequence
- RETURN VALUE . . The return value, if any (C callable only).
- ENTRY CONDITIONS Required conditions before call
- EXIT CONDITIONS. Conditions upon return of the call
- DESCRIPTION . . . A description of what the Virtuoso nanokernel service does when invoked and how a desired behavior can be obtained.
- EXAMPLE . . . . . One or more typical Virtuoso nanokernel service uses.
- SEE ALSO. . . . . List of related Virtuoso nanokernel services that could be examined in conjunction with the current Virtuoso nanokernel service.
- SPECIAL NOTES . . Specific notes and technical comments.

19.1. _init_process

• BRIEF . . . . . Initialize a nanokernel process

• CLASS. . . . . Process management

• SYNOPSIS . . . . . void _init_process (void *stack, int stacksize, void entry(void), int i1, int i2);

• DESCRIPTION . . . This C function initializes the process control structure of a process. It must be called before the process is started using start_process (). The entry point, the stacksize, the initial values for i1 and i2 and some internal variables are written to the PCS.

• RETURN VALUE . . none

• EXAMPLE . . . . . In this example, two processes using the same code but different parameters are initialized and started.

```
int adc1[100];          /* stack for first process */
int adc2[100];          /* stack for second process */
extern void adc_proc (void); /* process code */
extern struct adc_pars ADC_Params [2]; /* parameter structs */

_init_process (adc1,100, adc_proc, &ADC_Params [0], 0);
_init_process (adc2,100, adc_proc, &ADC_Params [1], 0);
_start_process (adc1)
_start_process (adc2)
```

• SEE ALSO. . . . . _start_process

• SPECIAL NOTES . .

19.2. _start_process

- BRIEF . . . . . Starts a nanokernel process from the low priority context
- CLASS . . . . . Process management
- SYNOPSIS . . . . . void _start_process (void *stack);
- DESCRIPTION . . . Starts a nanokernel process by making it executable. The process must have been initialized before. The process will start executing immediately. This call returns when the started process deschedules or terminates.
- RETURN VALUE . . none
- EXAMPLE . . . . .

```
int wsp1[100]
int wsp2[100]
extern void proc1 (void);
extern void proc2 (void);
int N = 1000;
_init_process (wsp1,100, proc1, 0, N)
_init_process (wsp2,100, proc2, 0, N)
_start_process (wsp1)
_start_process (wsp2)
```

- SEE ALSO. . . . . _init_process ()
- SPECIAL NOTES . . This function cannot be used from within a high priority nanokernel process. It must be called from the C main () function or by a microkernel task only.

19.3. ENDISR1

• BRIEF . . . . . Terminates an ISR and conditionally invokes the process swapper

• CLASS. . . . . Interrupt service management

• SYNOPSYS ENDISR1

```
ENDISR1 is defined in macro.h
```

• DESCRIPTION . . . This entry point must be called to terminate an ISR running at level 1 (global interrupts enabled). It decrements the level 1 interrupt counter and preforms a nanokernel process swap IFF

- the calling ISR interrupted the PRLO process
- a high priority process is ready to execute

• ENTRY CONDITIONS The ISR should have saved the interrupted context so that the exit sequence listed below would correctly terminate the ISR.

• EXIT CONDITIONS . This call terminates the ISR and does not return.

• EXAMPLE . . . . . This ISR accepts the IIOF0 external interrupt and sends a signal to two hi-priority processes.

```
#include “def21020.h”
#include “macro.h”
.segment /pm seg_pmco;
.global _rx_host_irqh;
.endseg;
.segment /dm seg_dmda;
.var HOST_EVENT = 8;
.extern _K_ArgsP;
.endseg;
.segment /pm seg_pmco;
_rx_host_irqh:
 dm(i7,m7) = MODE1; /* register of the INTSET that must
 dm(i7,m7) = ASTAT; be saved */
 dm(i7,m7) = r0;
 dm(i7,m7) = r1;
 dm(i7,m7) = r2;
 dm(i7,m7) = r4;
 dm(i7,m7) = r8;
 dm(i7,m7) = r12;
 r4 = i4; dm(i7,m7) = r4;
 dm(i7,m7) = i12;
```

```
 i4 = dm(_K_ArgsP);     /* i4 contains the channel */
 r2 = dm(HOST_EVENT); /* r2 contains data */
 PRHI_PSH; /* signals channel */
 ENDISR1; /* ends interrupt*/
.endseg;
```

• SEE ALSO. . . . . .

• SPECIAL NOTES . . A normal interrupt exit (popping saved registers and RETI) is not allowed for an ISR running at level 1.

19.4. K_taskcall

- BRIEF . . . . . Send a command packet to the microkernel process
- CLASS. . . . . Process communication
- SYNOPSYS . . . . . void K_taskcall (K_ARGS *A);
- DESCRIPTION . . . This C-callable function is used by all KS_ services to send command packets to the microkernel process.
- RETURN VALUE . . none
- EXAMPLE . . . . . .
- SEE ALSO. . . . . PRLO_PSH
- SPECIAL NOTES . . This function must be called by microkernel tasks only.

19.5. KS_DisableISR()

- BRIEF . . . . . Remove an ISR from the interrupt vector table
- CLASS . . . . . Interrupt service management
- SYNOPSYS . . . . . void KS_DisableISR (int isrnum);
- DESCRIPTION . . . This C function is equivalent to KS_EnableISR (isrnum, NULL). The interrupt is disabled, and the corresponding entry in the interrupt vector table is cleared.
- RETURN VALUE . . none
- EXAMPLE . . . . . .

```
KS_DisableISR (8) ;     /* remove the host-interrupt */
```

- SEE ALSO. . . . . KS_EnableISR,
- SPECIAL NOTES . . Interrupt numbers are:
  - 0..31 for interrupts enabled in the IRPTL- register

19.6. KS_EnableISR

- BRIEF . . . . . Install an ISR and enable the corresponding interrupt.
- CLASS. . . . . Interrupt service management
- SYNOPSYS . . . . . void KS_EnableISR (int isrnum. void isr (void));
- DESCRIPTION . . . This C function is used to install, remove, or replace an interrupt handler. It takes two parameters: an interrupt number, and a pointer to an ISR. The pointer is entered into the interrupt vector table, and if it is not zero.
- RETURN VALUE . . none
- EXAMPLE . . . . . .

```
extern void _host_irqh(void);
KS_EnableISR (8, _host_irqh);
```

- SEE ALSO. . . . . KS_DisableISR
- SPECIAL NOTES . . Interrupt numbers are:
  - 0..31 for interrupts enabled in the IRPTL register

19.7. PRHI_GET

• BRIEF . . . . . Remove next packet from linked list channel

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . PRHI_GET

```
PRHI_GET is defined in macro.h
```

• DESCRIPTION . . . If the channel is not empty, the first packet is removed from the linked list, the Z flag is reset, and a pointer to the packet is returned. If the channel is empty, the Z flag is set and a NULL pointer is returned. The calling process is never swapped out as a result of calling this service.

• ENTRY CONDITIONS

```
i4 = pointer to linked list channel struct
```

• EXIT CONDITIONS. If the list is not empty:

```
r8 is corrupted
r2 = pointer to removed list element
the Z flag is cleared
```

If the list is empty

```
r8 is corrupted
r2 = 0
the Z flag is set
```

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm (CHANNEL);
PRHI_GET;
```

• SEE ALSO. . . . . PRHI_GETW, PRHI_PUT

• SPECIAL NOTES . . This service must not be called from the low priority context.

19.8. PRHI_GETW

• BRIEF . . . . . Get next packet from linked list channel, or deschedule

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . PRHI_GETW

```
PRHI_GETW is defined in macro.h
```

• DESCRIPTION . . . If the channel is not empty, the first packet is removed from the linked list and a pointer to it is returned. If the channel is empty, the calling process is swapped out and set to wait on the channel. It will be rescheduled by the next call to the PRHI_PUT service on the same channel.

• ENTRY CONDITIONS

```
i4 = pointer to linked list channel struct
i0 = pointer to PCS of calling process
```

• EXIT CONDITIONS.

```
r2 = pointer to list element
r8, r1 are corrupted
```

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm (CHANNEL);
PRHI_GETW;
```

• SEE ALSO. . . . . PRHI_GET, PRHI_PUT

• SPECIAL NOTES . . This service must not be called from the low priority context or by an isr.

19.9. PRHI_POP

• BRIEF . . . . . Remove next element from a stack channel

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . PRHI_POP

```
PRHI_POP is defined in macro.h
```

• DESCRIPTION . . . If the stack is not empty, the top element is removed and returned to the caller. The Z flag is reset. If the stack is empty, the Z flag is set and an undefined value is returned. The calling process is never swapped out as a result of calling this service.

• ENTRY CONDITIONS

```
i4 = pointer to stack channel struct
```

• EXIT CONDITIONS. If the stack is not empty:

```
r8, r0, r1are corrupted
r2 = the element removed from the stack
the Z flag is cleared
```

If the stack is empty:

```
r8,r0,r1 are corrupted
r2 = undefined
the Z flag is set
```

• EXAMPLE . . . . . .

```
#include “traps.inc”
i4 = dm(CHANNEL);
PRHI_POP;
```

• SEE ALSO. . . . . PRHI_POPW, PRHI_PSH

• SPECIAL NOTES . . This service must not be called from the low priority context.

19.10. PRHI_POPW

• BRIEF . . . . . Remove next element from a stack channel, or deschedule

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . PRHI_POPW

```
PRHI_POPW is defined in macro.h
```

• DESCRIPTION . . . If the stack is not empty, the top element is removed and returned to the caller. If the stack is empty, the calling process is swapped out and set to wait on the channel. It will be rescheduled by the next call to the PRHI_PSH service on the same channel.

• ENTRY CONDITIONS

```
i4 = pointer to stack channel struct
i0 = pointer to PCS of calling process
```

• EXIT CONDITIONS.

```
i1 = element removed from the stack
r0, r1, r8 are corrupted
```

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm(CHANNEL);
PRHI_POPW;
```

• SEE ALSO. . . . . PRHI_POP, PRHI_PSH

• SPECIAL NOTES . . This service must not be called from the low priority context or by an isr.

19.11. PRHI_PUT

• BRIEF . . . . . Add a packet to a linked list channel

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . PRHI_PUT
PRHI_PUT is defined in macro.h

• DESCRIPTION . . . If a process is waiting on the channel, the pointer to the packet is passed on, and the waiting process is rescheduled. Otherwise the packet is linked in at the head of the list. In either case, control returns to the caller.

• ENTRY CONDITIONS

```
i4 = pointer to channel
r2 = pointer to packet to add to the list
```

• EXIT CONDITIONS.

```
r0, r1, r8 are corrupted
All other registers are preserved
```

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm (CHANNEL);
r2 = dm (PACKET);
PRHI_PUT
; the packet is added to the list
```

• SEE ALSO. . . . . PRHI_GET, PRHI_GETW

• SPECIAL NOTES . . This entry point must not by called from the low priority context, but it can be used by interrupt handlers.

The first word of the packet is used as a link pointer, and will be overwritten.

19.12. PRHI_PSH

• BRIEF . . . . . Push a word on a stack channel

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . PRHI_PSH

```
PRHI_PSH is defined in macro.h
```

• DESCRIPTION . . . If a process is waiting on the channel, the data word is passed on, and the waiting process is rescheduled. Otherwise the data word is pushed on the stack. In either case, control returns to the caller.

• ENTRY CONDITIONS

```
i4 = pointer to channel
r2 = data word to push
```

• EXIT CONDITIONS.

```
r0, r1, r4 and r8 are corrupted
All other registers are preserved
```

• EXAMPLE . . . . . .

```
#include “macro.h”
.extern _K_ArgsP        ; microkernel input stack
; send a command packet to the microkernel
; assume i0 points to the command packet
i4 = dm (_K_ArgsP);
r2 = dm (0,i0);
PRHI_PSH;
```

• SEE ALSO. . . . . PRHI_POP, PRHI_POPW

• SPECIAL NOTES . . This entry point must not by called from the low priority context, but it can be used by interrupt handlers.

19.13. PRHI_SIG

• BRIEF . . . . . Send an event on a signal channel

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . PRHI_SIG

```
PRHI_SIG is defined in macro.h
```

• DESCRIPTION . . . If a process is waiting on the channel, it is rescheduled (put at the tail of the process FIFO). Otherwise the event count is incremented. In either case, control returns to the caller.

• ENTRY CONDITIONS

```
i4 = pointer to channel
```

• EXIT CONDITIONS.

```
r0, r1, r2 are corrupted
All other registers are preserved
```

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm (SYNC_CHAN);
PRHI_SIG;
```

• SEE ALSO. . . . . PRHI_WAIT

• SPECIAL NOTES . . This entry point must not by called from the low priority context, but it can be used by interrupt handlers.

19.14. PRHI_WAIT

• BRIEF . . . . . Consume an event from a signal channel, or deschedule

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . PRHI_WAIT

```
PRHI_WAIT is defined in macro.h
```

• DESCRIPTION . . . If the event counter is not zero, it is decremented an control returns to the caller. If the event counter is zero, the calling process is swapped out and set to wait on the channel. It will be rescheduled by the next call to the PRHI_SIG service on the same channel.

• ENTRY CONDITIONS

```
i4 = pointer to signal channel struct
i0 = pointer to PCS of calling process
```

• EXIT CONDITIONS.

```
r0, r1, r2 are corrupted
```

• EXAMPLE . . . . . .

```
#include “macro.h”
; wait for event on SYNC_CHAN
i4 = dm(SYNC_CHAN);
PRHI_WAIT;
; the event has happened
```

• SEE ALSO. . . . . PRHI_SIG

• SPECIAL NOTES . . This service must not be called from the low priority context or by an isr.

19.15. PRLO_PSH

• BRIEF . . . . . This call is for internal use only, and is not exactly the equivalent of PRHI_PSH for the PRLO process. This call assumes that a PRHI process is waiting on the channel, and will crash the system if there isn't. PRLO_PUSH is used by the K_taskcall function to send command packets from a task to the microkernel process.

19.16. YIELD

• BRIEF . . . . . Yield CPU to next nanokernel process

• CLASS. . . . . Process management

• SYNOPSIS . . . . . YIELD

```
YIELD is defined in macro.h
```

• DESCRIPTION . . . The calling process is swapped out and added to the tail of the process FIFO. The process at the head of the FIFO is swapped in. This may be the same process, if it was the only one ready to execute.

• ENTRY CONDITIONS

```
i0 = pointer to PCS of calling process
```

• EXIT CONDITIONS.

```
r1, r2, r0 are corrupted
```

• EXAMPLE . . . . . This example shows how to avoid a redundant YIELD operation, by testing the process FIFO

```
#include “nanok.h”
#include “macro.h”
;
r0 = dm(NANOK_HEAD);
r1 = 0;
comp(r0,r1);            ; test head of process FIFO
if eq jump label;
YIELD                   ; yield if there is another process
label: ...
```

• SPECIAL NOTES . . This service must not be called from the low priority context or by an isr.

20. Predefined drivers

Two devices drivers are already added to this release of the Virtuoso kernel. They are:

- the timer device driver
- the host interface device driver
- a communication driver based on shared memory (present in VIRTUOS CLASSICO VSP)

The timer device driver is needed for time-out features of some kernel services and for kernel timer services. The host interface device driver is written to be able to communicate between the host server program and the target board. The shared memory driver was especially written for IXTHOS-21020 boards. The IXD-7232 was provided with two 21020 processors, that can communicate using shared memory. This driver can be extended for other types of communication.

20.0.1. The timer device driver

The timer driver is already installed and enabled in procedure `main()` of the examples that accompany the release of the Virtuoso kernel. If the timer ISR is installed and enabled, the application programmer can read out the timer in high and in low resolution.

The two procedures to read out the timer value are:

- KS_HighTimer ()
- KS_LowTimer ()

In high resolution, the number of timer counts are returned. On the 21020, the count is equal to a period of the clock frequency.

In low resolution, the number of kernel ticks are returned. A kernel tick is a multiple of timer count and defined in the main() function. As this value is a 32-bit wraparound value, it is more intresting to calculate the difference between two values read out consecutively. However, to facilitate this, kernel service KS_Elapse() is written for this purpose.

See the Alphabetical List of Virtuoso kernel services earlier in this manual for a full description of these kernel services.

The timer device driver reserves event signal number 4 or 14 (depending on the priority) for its use. As the host interface uses event number 8, selecting 4 will allow the timer interrupt to interrupt the host interface ISR, while select-

ing 14 can delay the processing of the timer ISR.

Note: In our newest versions of VIRTUOSO CLASSICO and MICRO, the timer is always signalling event number 48.

20.0.2. The host interface device driver

The host interface driver is installed by calling procedure `init_server()`. In the examples that accompany the release of the Virtuoso kernel, the installation of the host interface is done in procedure `main()`.

The host interface driver can be used on two levels. The lowest level needs only one kernel resource, HOSTRES, which secures the use of the low level host interface. This kernel resource must always be locked by the task that wants to make use of the host interface, and unlocked if this task has finished using the host interface. A list of low level procedures are at the disposal of the application programmer to do simple character-oriented I/O:

- server_putch()
- server_pollkey()
- server_terminate()
- server_pollesc()

These procedures will do the locking and unlocking of HOSTRES, so that HOSTRES is transparent to the application programmer, using the low level host interface.

Also installed in the examples is an easy-to-use character-oriented I/O interface, based on two tasks, `conidrv` and `conodrv`, two queues, `CONIQ` and `CONOQ`, two resources, `HOSTRES` and `CONRES`, and a procedure called `printl()`. This higher level interface driver makes use of the low level interface procedures.

It is possible to use an even lower level of the host interface. Doing this, the application programmer can build other host interfaces that do more than character-oriented I/O. The minimum that is needed to make use of the lowest level host interface, is the kernel resource HOSTRES, to secure the use of the interface, and the procedure, named `call_server()`. Note, however, that HOSTRES is not needed if only one task makes use of the lowest level host interface and if the Task Level Debugger is not present. It is not the intention of this manual to lay out the internals of the host interface and the communication protocol between the host server program and the target board(s). Please contact Eonic Systems if more information is wanted on this topic.

For more details on the different levels of the host interface, see “Host server

low level functions” and “Simple terminal oriented I/O” in the chapter of “Runtime libraries”.

The host interface device driver reserves event signal number 8 for its own use.

20.0.3. Shared memory driver

This driver was specific written for IXTHOS boards. These boards are provided with two 21020 processors. Each of these processors, have their own program and data memory space. Beside that, there is a pool of shared memory present. Both processors can access the shared memory. In Virtuoso Classico, the shared memory is used for communication. The customer can install the driver by adding to the SYSDEF-file:

NETLINK NODE1 ‘MemLink()’, NODE2 ‘MemLink()’

20.1. Task Level Timings

Following is a list of task level timings of some of the kernel services provided by Virtuoso. These timings are the result of a timing measurement on a ADSP-21020 board with a clock speed of 25MHz. The kernel used for these timings is the VIRTUOSO Microkernel.

All timings are in microseconds. The C compiler used is the G21k C Compiler v.3.0 from Analog Devices.

```
Minimum Kernel call
Nop (1)                          5
Message transfer
Send/Receive with wait
Header only (2)                  34
16 bytes (2)                     37
128 bytes (2)                    46
1024 bytes (2)                   118
Queue operations
Enqueue 1 byte (1)               9
Dequeue 1 byte (1)               9
Enqueue 4 bytes (1)              9
Dequeue 4 bytes (1)              10
Enqueue/Dequeue (with wait) (2) 35
Semaphore operations
Signal (1)                       6
```



```
Signal/Wait (2)                 28
Signal/WaitTimeout (2)          34
Signal/WaitMany (2)             37
Signal/WaitManyTimeout (2)      43
Resources
Lock or Unlock (1)              7

Note :
(1):  involves no context switch
(2):  involves two context switches. Timing is round-
      trip time.
```

20.2. Application development hints.

The easiest way to start is to copy and modify one of the supplied examples. Some of the necessary files have fixed names, so each application should be put in a separate directory.

The following files will be needed for each application:

SYSDEF:

The VIRTUOSO system definition file. The SYSGEN utility will read this file and generate NODE1.C and NODE1.H.

MAIN1.C:

This contains some more configuration options, and the C ‘main’ function. Copy from one of the examples.

A number of configuration options are defined in this file, so they can be changed without requiring recompilation of all sources (this would be necessary if SYSDEF is modified).

CLCKFREQ : this should be defined to be the clock frequency of the hardware timer used to generate the TICKS time.

TICKTIME : the TICK period in microseconds.

TICKUNIT: the TICK period in CLCKFREQ units.

TICKFREQ: the TICK frequency in Hertz.

The number of available timers, command packets and multiple wait packets are also defined in this file. How much you need of each depends on your

application, but the following guidelines may be followed:

Timers are used to implement time-outs (at most one per task), and can also be allocated by a task.

A command packet will be needed for each timer allocated by a task. Command packets used for calling a kernel service are created on the caller's stack and should not be predefined.

A multiple wait packet will be needed for each semaphore in a KS_WaitM service call (for as long as it remains waiting).

MAIN1.C also defines some variables used by the console driver tasks, the clock system, the debugger task, and the graphics system. These are included automatically if you use the standard names for the required kernel objects.

XXX.ACH: specifies architecture file0

MAKEFILE:

The makefiles supplied in the EXAMPLES directory can easily be modified for your application. They also show how to organize things so you can optionally include the task level debugger. If you want to include the task level debugger, put the corresponding definitions out of comment:

```
VIRTLIB = $(LIBS)\virtosd.lib
DD = -dDEBUG
DDD = -P "DEBUG"
```

and put the other definition in comment:

```
# VIRTLIB = $(LIBS)\virtos.lib
```

whereby # is the comment sign.

There are also two define-statements in the 'mainx.c'-file, that the customer can change in order to personalize the debugger: (only implemented in VIRTUOSO CLASSICO)

```
# define MONITSIZE 1024 /* number of monitor records */
# define MONITMASK MONALL - MONEVENT /* defines the
        quantity of information */
```

Then remake the application, just by doing:

```
MAKE <Enter>.
```


`LINKFILE:` list of the object versions of all source files to be linked along.

YOUR SOURCE FILES : In the examples, this is just test.c

After you have done make-ing your application, you can run the application by typing:

> 21khost -rlsi test

21. Virtuoso on the ADSP 2106x SHARC

21.1. Virtuoso implementations on the 21060

At this moment, both VIRTUOSO MICRO/SP and VIRTUOSO CLASSICO/VSP exist for the ADSP 2106x. The former only includes a microkernel, while the latter uses both a microkernel and nanokernel. Until now, the implementation of VIRTUOSO CLASSICO/VSP only uses link ports to communicate between nodes. This chapter only covers Virtuoso Classico/VSP for SHARC.

21.2. SHARC chip architecture

This section contains a brief description of the SHARC processor architecture. It is not intended to be a replacement of the Processor's User Manual, but as a quick lookup for the application programmer. Detailed information can be found in the "ADSP-2106x SHARC User's Manual" from Analog Devices, Inc.

<SECTION TO BE COMPLETED - PLEASE REFER TO THE SHARC USER MANUAL >

21.3. Relevant documentation

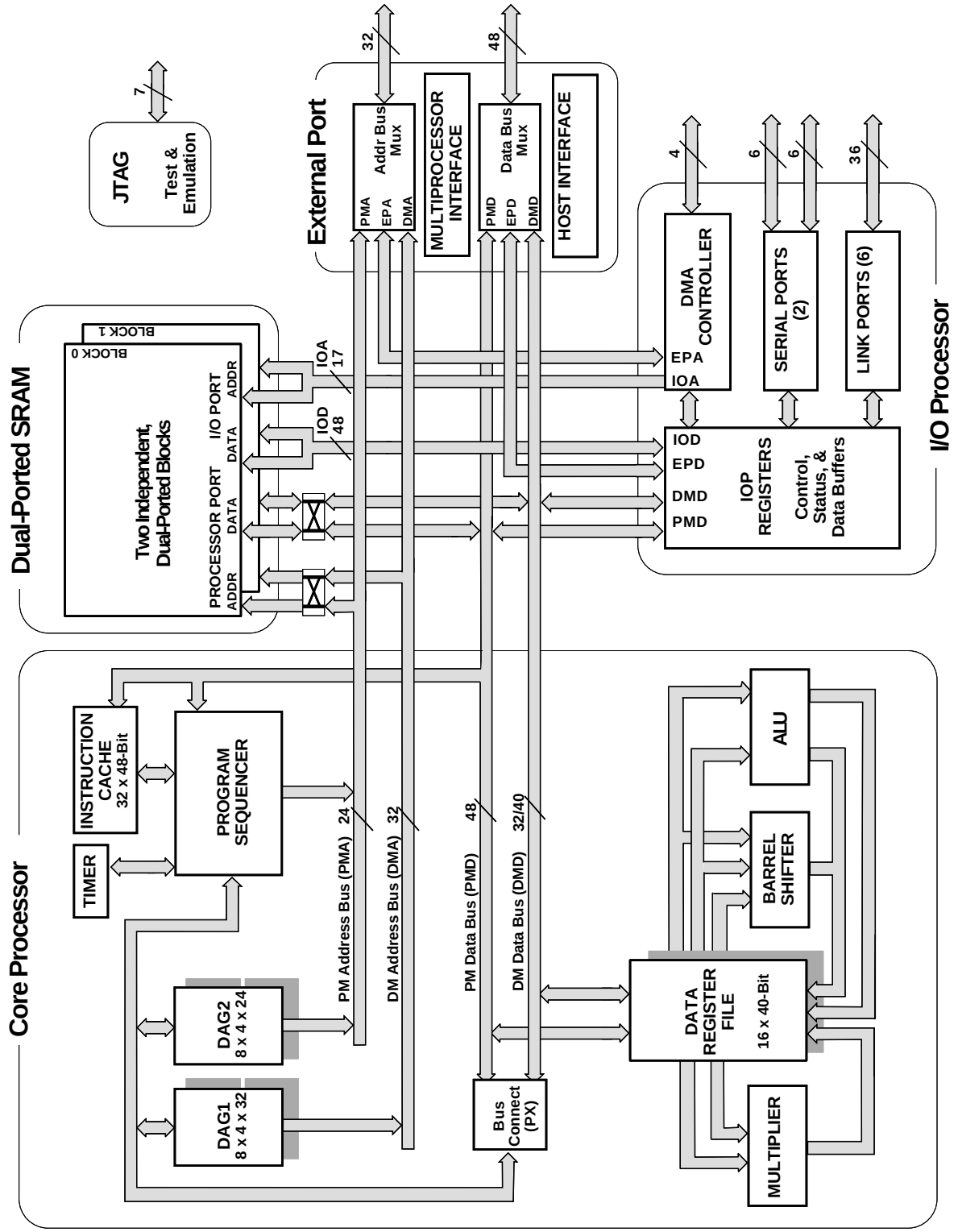
1. "ADSP-2106x SHARC User's Manual", Analog Devices, Inc., 1995
2. "ADSP-21000 Family C Tools Manual", Analog Devices, Inc., 1995
3. "ADSP-21000 Family Assembler Tools & Simulator Manual", Analog Devices, Inc., 1995
4. "ADSP-21000 Family C Runtime Library Manual", Analog Devices, Inc., 1995

21.4. Version of the compiler

Analog Devices, Inc. has different versions of the G21k compiler. In the final release, version 3.2d was used.

21.5. SHARC silicon revisions

We recommend the use of production silicon (rev 2.0 and on) with Virtuoso Classico /VSP. Earlier revisions exhibit anomalies that could cause system failures.



21.6. Developing ISR routines on the SHARC

21.6.1. General principles

When using Virtuoso Classico, there are basically 2 types of ISRs that can be used:

- ISRs that modify the status of a kernel object
- ISRs that don't.

The second type is the easiest to implement, as they do not require an interface to the kernel. The ISR consists of 4 parts:

- Pushing a number of registers on the stack,
- Performing whatever operation that is necessary,
- Restoring the registers from the stack,
- Return from the interrupt.

In this case, any register can be used, as long as it gets restored to its original value before returning from the interrupt.

The first type of ISR requires an interface to the nanokernel, and needs to be much more structured. It consists of 3 parts:

- Saving a number of registers on the stack. What registers and the order in which they need to be saved is fixed.
- Performing whatever operation that is necessary - including the modification of the status of one or more kernel objects.
- Transfer control to the nanokernel, which will decide on the next step.

The list of registers that need saving is called INTSET. It contains the following registers (in order): MODE1, ASTAT, r0, r1, r2, r4, r8, r12, i4, 12. Also see 21.7.8., "Register conventions" on page 12.

The second part of the ISR requires a lot more knowledge of the internals of the nanokernel and microkernel. For more information, check 21.7., "The nanokernel on the 21060" on page 7.

21.6.2. Writing an ISR routine

To illustrate, here are 2 examples, one of each category.

First example: a simple ISR, no kernel entry:

```
#include <def21060.h>

.segment /pm seg_pmco;
.global _simple_isr;
.endseg;

.segment /dm seg_dmda;
.extern _CountP;
.endseg;

.segment /pm seg_pmco;
_simple_isr:
dm(i7,m7) = r0;
r0 = i0;
dm(i7,m7) = r0;
i0 = _CountP;
r0 = dm(m5,i0);
r0 = r0 + 1;
dm(m5,i0) = r0;
r0 = dm(1,i7);
i0 = r0;
r0 = dm(2,i7);
modify(i7,2);
rti;
.endseg;
```

Second example: as an example for an ISR that does enter the kernel, here is the host-interrupt service routine used in Virtuoso Classico/VSP for SHARC:

```
#include <def21060.h>
#include “macro.h”/* macro’s for nanokernel */
#include “event.h”

.segment /pm seg_pmco;
.global _rxhost_irqh;
```



```
.endseg;
.segment /dm seg_dmda;
.extern _K_ArgsP; /* channel for the kernel */
.endseg;

.segment /pm seg_pmco;
_rxhost_irqh:
/*begin - registers that MUST be saved by the IRQ */
dm(i7,m7) = MODE1;
dm(i7,m7) = ASTAT;
dm(i7,m7) = r0;
dm(i7,m7) = r1;
dm(i7,m7) = r2;
dm(i7,m7) = r4;
dm(i7,m7) = r8;
dm(i7,m7) = r12;
r4 = i4;
dm(i7,m7) = r4;
dm(i7,m7) = i12;
/* end - registers that MUST be saved by the IRQ */
i4 = dm(_K_ArgsP);
r2 = HOST_EVENT;
PRHI_PSH; /* pushes event on kernel-channel */
ENDISR1; /* ends the interrupt routine, transfers control to nanokernel*/
.endseg;
```

21.6.3. Installing an ISR routine

Installing an ISR requires a call to KS_EnableISR(). The arguments to this function are the IRQ number, as defined in IMASK, and a pointer to the function which is to serve as the ISR.

ISRs can be installed and enabled in any part of the code of the application. To keep a good overview, however, it is preferable to install/enable them in a central place, like a ‘master’ task, or in the main() function.

It is possible that additional initialisation of registers and/or peripheral I/O has to be done. The best way is to write a C-callable procedure, that does the necessary additional initialisations, and call this procedure before or after the call to KS_EnableISR().

21.6.4. List of ISR related services

1. ENDISR1: Entering the kernel from within an ISR.
2. KS_EventW (int event) : Waits for an event at the task level.
3. KS_EnableISR (int IRQ, void (*ISR) (void))): Installs the ISR.
4. KS_DisableISR (int IRQ): Disables the ISR.

See 21.7., "The nanokernel on the 21060" on page 7 for more details.

21.7. The nanokernel on the 21060

21.7.1. Introduction

The nanokernel provides the lowest level of functionality in the Virtuoso system. It is designed to perform extremely fast communication and context switching for a number of processes. It also provides the entry points necessary to integrate interrupt handlers with the rest of the system. The price to pay for speed is that the nanokernel processes and interrupt handlers must obey very strict rules regarding to their use of CPU registers and the way they interact with each other.

From the point of view of the nanokernel, an application program consists of a collection of three types code modules:

- a single low priority process (PRLO-process).
- any number of high priority processes (PRHI-process).
- any number of interrupt handlers.

It is important to understand what exactly is meant by a process. A process is a thread of execution that has both an identity and a private workspace. These two properties (logically equivalent) make it possible for a process to be swapped out, and wait for an external event while another process is allowed to continue. Interrupt handlers in contrast, do not have a private workspace.

The PRHI processes are scheduled in strict FIFO order, and must observe the special register conventions mentioned above. The PRLO process is assumed to be a C function (using the compiler register conventions), and must always be ready to execute. You can compare it with the IDLE-task of the microkernel.

All communication inside the nanokernel is performed using channels. Several types exit. The simplest type is used for synchronization and corresponds to a counting semaphore. The other types are used for data transfer. The possibility is provided that a user can add his own channel types.

The microkernel, managing the tasks, is built as an application on top of the nanokernel. The main component is a PRHI process that executes commands it receives from a channel. When the channel is empty, the microkernel looks for the next task to run, replaces the nanokernel IDLE-process by that task and performs the additional register swappings required for C tasks.

The nanokernel is not ‘aware’ of the manipulations performed by the microkernel. As far as it concerned, there is only one PRLO-process, which it exe-

cutes whenever no PRHI-process is ready to continue. This makes it possible to use the nanokernel on its own.

21.7.2. Internal data structures

The user does not normally need to access the internal data structures used by the nanokernel. The documentation in this section is provided only for a better understanding of how the nanokernel operates.

A process is represented by a pointer to a Process Control Structure (PCS). For PRHI processes, the PCS occupies the first eight words of its stack. Two entries are placed at the top because of the decrementing stackpointer. A static PCS is used for the Idle-process. More details on the PCS will be introduced in the section on process management.

A channel is represented by a pointer to a Channel Data Structure (CDS). The first word of a CDS is a pointer to the PCS of a process waiting for that channel, or NULL. Other fields depend on the type of the channel and will be introduced in the section on nanokernel communications.

The following static variables are used by the nanokernel to keep track of the state of the system:

NANOK_PRHI: Pointer to the PCS of the current PRHI-process, or NULL if there is none.

NANOK_HEAD: Head pointer for a linked list of PRHI-processes that are ready to run. When the current PRLO-process is swapped out, the PRHI-process at the head of the list is removed, and becomes the current process.

NANOK_TAIL: Tail pointer for a linked list of PRHI-processes that are ready to run. When a process becomes ready to execute, it is added to the tail of the list.

NANOK_PRLO: Pointer to the PCS of the PRLO-process. This is a constant as far as the nanokernel is concerned. The microkernel modifies this pointer.

NANOK_CRIT: This is the number of interrupt handlers running with global interrupts enabled that are not yet terminated. The process swapping is disabled while this value is not zero. On the 21060, this field is not necessary, because IMASKP contains all necessary information on the interrupt nesting state.

Symbolic constants for accessing kernel variables and elements of a PCS are defined in the file ‘nanok.h’

21.7.3. Process management.

The nanokernel variables are initialized as follows:

- NANOK_PRHI = 0;
- NANOK_HEAD = 0;
- NANOK_TAIL = &(NANOK_HEAD)
- NANOK_PRLO = &(PCS for IDLE process)
- NANOK_CRIT = 0;

This means that when an application is started, the idle-process of the nanokernel will start.

In the current version of the nanokernel in Virtuoso Classico/VSP for SHARC, all PRHI-processes must be created and started by the PRLO-process. Two steps are required to create a process:

- create a stack for the process.
- initialise the PCS, and start up the process

The stack can be placed anywhere in memory. It can be a C-array of integers or a memory block allocated by malloc.

The function start_process (*stack, stacksize, entry, i1, i2) is used to initialize the PCS and start the process. It writes the following values to the first 8 words of the stack (see FIGURE 15 on page 10):

PR_LINK:	0	link pointer
PR_SSTP:	(see figure)	saved stack pointer
PR_PI3:	0	saved i3, not used for prhi
PR_PI1:	i1	initial / saved i1
PR_PI2:	i2	initial / saved i2
PR_MODE1:	MODE1	saved MODE1
PR_ASTAT:	ASTAT	saved ASTAT
PR_BITS:	0	user flags, not used for prhi

The following 2 words are written at the top of the stack:

PR_PEND:	NANOK_TRMP	pointer to terminate code
PR_PRUN:	entry	pointer to entry point

After the initialisation, start_process starts the process. As the caller is the PRLO-process, there can be no other PRHI process and the new process will start execution immediately. Control returns to the caller when the new

process terminates or is descheduled by waiting for a channel.

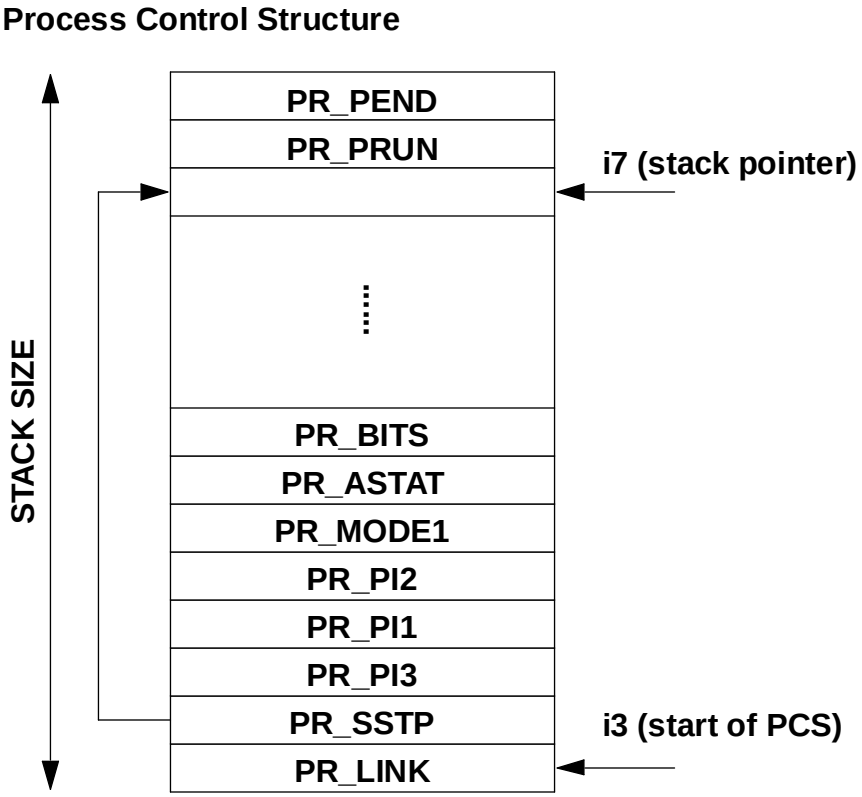


FIGURE 15 Process Control Structure of a process.

When a process is swapped in, the nanokernel simply pops the first entry on the stack, and the process starts at that address. When the process terminates, it should perform some clean-up actions. As the pointer to the termination code is already pushed onto its stack, the process simply needs to pop that last entry from the stack end jump to that address. At that point, it will jump to the process termination code.

To restart a terminated process, repeat the call start_process().

When a PRHI process is swapped in, i3 points to the start of the PCS. A process can create local variables by incrementing its stack pointer by the number of words required.

Note: On the 21060, the stacksize is also a parameter. This is because of the stackpointer is moving from the top to the bottom of the stack.

21.7.4. Nanokernel communications

A channel type is defined by a data structure and a number of nanokernel services that operate on it. Each instance of the data structure is called a channel. Channels provide both process synchronization and data communication.

The nanokernel does not itself use or create channels. However, the services that operate on channels should be considered part of the nanokernel, as they may modify the process FIFO or invoke the nanokernel swapper.

All channels have an internal state. What exactly is represented by the state depends on the type of the channel - typically this will be the occurrence of an event or the availability of data. An operation on a channel can consist of any combination of the following action types:

Wait: The process is said to ‘wait for a channel’

Signal: Signalling action. This action can reschedule a process.

Test and modify: modifies or tests the state of a channel.

Three channel types are predefined in the current nanokernel implementation in Virtuoso Classico/VSP. It is possible to create new channel types if necessary; an example will be given at the end of this chapter. A full description of the nanokernel services for each of these channel types can be found in the alphabetical list in the next chapter.

21.7.5. SEMA_CHAN - counting or semaphore channel

This is an implementation of a counting semaphore. It is typically used by interrupt handlers to reschedule a process that was waiting for the interrupt. The C_CHAN structure has two fields:

CH_PROC: pointer to the PCS of the waiting process or NULL

CH_NSIG: event counter

Two nanokernel services are available for this channel type:

PRHI_WAIT: waiting action

PRHI_SIG: signalling action

21.7.6. LIFO_CHAN - List channel

This type of channel maintains a linked list of memory blocks, using the first word in each block as a link pointer. The microkernel uses this type to implement its free list of command packets, data packets and timers. If used for data communication, it behaves as a LIFO buffer.

The LIFO_CHAN structure has two fields:

CH_PROC: pointer to the PCS of a waiting process or NULL
CH_LIST: pointer to the first element of the linked list or NULL

The nanokernel services that operate on this type are:

PRHI_GETW: wait action
PRHI_GET: test and modify action
PRHI_PUT: signal action

21.7.7. STACK_CHAN - Stack channel

This type of channel uses a memory block as a data stack. The microkernel uses a stack channel to input commands from tasks and the network drivers, and to receive events from interrupt handlers.

The STACK_CHAN structure has three fields:

CH_PROC: pointer to the PCS of a waiting process or NULL
CH_BASE: pointer to the base of the stack
CH_NEXT: pointer to the next free word on the stack

The nanokernel services that operate on this type are:

PRHI_POPW: wait action
PRHI_POP: test and modify
PRHI_PSH: signal action

21.7.8. Register conventions

In order to understand the register conventions adopted by the Virtuoso nanokernel, the following register sets should be introduced:

CSAVE: r3, r5, r6, r7, r9, r10, r11, r13, r14, r15, i0, i1, i2, i3, i5, i8, i9, i10, i11, i14, i15, m0, m1, m2, m3, m8, m9, m10, m11, mrf, mrb, MODE1, MODE2, USTAT1, USTAT2

CFREE: r0, r1, r2, r4, r8, r12, i4, i12, m4, m12

SYSSET: IRPTL, IMASK,...

NSWAP: i1, i2, i3, i7, MODE1, ASTAT

INTSET: MODE1, ASTAT, r0, r1, r2, r4, r8, r12, i4, i12

The CSAVE and CFREE sets are defined by the procedure calling standard

of the C-compiler. CSAVE is the set of registers that is preserved across a subroutine call - if a function uses any of these, it must restore the initial value on return. CFREE is the set of registers that are freely available to all functions - the caller of a subroutine is responsible for preserving them if necessary. The definition of these two sets largely determine what the microkernel is expected to do when swapping tasks. When a task is swapped out as a result of calling a kernel service (which to the task is just a C function), only the CSAVE set need be saved. When a task is preempted (which means that an interrupt handler has woken up the kernel), the CFREE set must be saved as well. Actually, since most of the microkernel is written in C, the CFREE set must be saved before the actual service requested by the interrupt handler is called.

The SYSSET register are used for system and peripheral control only. They are never swapped, and should be regarded as global resources. Only very low level routines (such as hardware drivers) will ever need to access these registers.

The INTSET registers are those that must have been pushed on the stack when an interrupt handler terminates and wakes up the kernel by calling the ENDISR1 service (this is discussed in more detail in the section on interrupt handling below). At that point, the nanokernel needs some registers to work with. It would be a waste of time to pop all registers saved by the ISR, only to have to push them again when entering the kernel.

The registers in NSWAP are saved and restored by the nanokernel when swapping processes. For the PRLO process (assumed to be a C-function, using i3 as its frame pointer) the nanokernel will save and restore i3 in the normal way. When a PRHI-process is swapped in, i3 will be set to point to its process control structure. A PRHI-process can use i3 to access local variables created in its workspace, and should normally not modify this register. If it does, the initial value can always be reloaded from NANOK_PRHI. I3 must point to the PCS whenever the process calls a nanokernel service and when it terminates.

The NSWAP register set is always available, but note the special use of i3.

If a PRHI process is swapped in as the result of a C-function call by the PRLO-process, then the CFREE set is available for use by the PRHI process. This means that the process can safely call any C-function. It should of course save those registers in CFREE that it wants to preserve across the call.

If a PRHI process is swapped in as the result of an interrupt handler calling an ENDISR service, then the INTSET registers are available to the PRHI-process. Before calling a C-function, the process must save the set CFREE-

INTSET, and it must restore the same registers before it is swapped out (this is always possible, since a PRHI-process is never preempted).

21.7.9. Interrupt handling

In the Virtuoso system model, interrupt handlers are the interface between asynchronous events and the processes that are waiting for them. To be useful, most interrupt handlers will have to interact with the rest of the system at some time. Using flags to be 'polled' by the foreground process is usually not an acceptable practice in a real-time system. This method introduces a 'superloop' structure into the application, with all its inherent problems.

In a system using the nanokernel, interrupt handlers can communicate with processes using the same channel operations that are available to processes. However, there are some restrictions.

In contrast to a process, an interrupt service routine does not have a private workspace, it executes on the stack of whatever process was interrupted. An ISR on the 21060 can be interrupted by an ISR of higher priority. So, any number of interrupt handlers can be piled on top of each other on the same stack, owned by the current process. This has some important consequences:

1. If an ISR calls a channel service that has a signal action, any process swap that results from this call must be delayed until all interrupt handlers have terminated. These implies that only the PRHI_type of channel operations can be used, as these do not invoke the swapper for a signal action (there is no need to swap, as the caller already has the highest priority). When the last stacked interrupt terminates, the swapper must be called to verify if a swap from the PRLO-process to a PRHI-process is necessary.
2. An ISR must never call any channel service that has a wait action. Doing so would also block all other interrupt handlers that are stacked below it, as well as the current process. Another way of seeing this is that an ISR cannot wait for something because it doesn't have a separate identity - the producer of the external event (another ISR) has no means of representing who is waiting for it.

Note: The 21060 is provided with a system stack. When an external or a timer interrupt occurs, MODE1 and ASTAT are pushed on that system stack. The pop-operation is performed by the RTI. In our version of the kernel, MODE1 and ASTAT are always pushed on the stack of the interrupted process.

21.7.10. The ISR-level

The 21060 has only one ISR-level. When entering the interrupt handler, global interrupts are enabled. An interrupt of higher priority can interrupt an interrupt of lower priority.

An interrupt ends with a call to the nanokernel function ‘ENDISR1’. At that point, the nanokernel will verify if a process swap is required and allowed. The condition tested is the logical AND of:

- NANOK_PRHI = 0;/* The current process is a PRLO-process */
- NANOK_HEAD != 0; /* There is a PRHI-process */
- NANOK_CRIT = 0; /* There are no more nested interrupts */

In case of a swap, the interrupt is finished and the PRHI-process is swapped in. If there are nested interrupts, first all interrupts are finished.

NOTE: As NANOK_CRIT is not used in the nanokernel on 21060, the last test has been replaced by a check of IMASKP.

21.7.11. Communicating with the microkernel

As mentioned before, the VIRTUOSO microkernel is implemented as a PRHI-process. It uses a single stack based channel to receive commands from the tasks, the network drivers, other PRHI-processes and interrupt handlers. A pointer to this channel is exported in the C-variable K_ArgsP.

Two types of data can be pushed onto this channel:

1. Small integers (0-63) are interpreted as events. Events are simple binary signals that a task can wait for using the KS_EventW()-service. Most events will be generated by interrupt handlers and driver processes. For the 21060 version, event numbers have been assigned as follows:

- 8: host event.
- 10-15: event numbers for KS_LinkOutW
- 16-21: event numbers for KS_LinkInW
- All other events (0-31) are reserved for the kernel
- 48: timer event.
- rest (32-63, except 48): are free.

All event numbers used by the kernel - and reserved for the kernel - are defined in “event.h”.

2. All other values pushed onto the microkernel input channel are interpreted

as a pointer to a command packet. Command packets are the primary form of communication used within the Virtuoso system. They are used by the tasks to request microkernel services, sent across the Virtuoso network to implement remote kernel calls, and put on waiting lists to represent a task that is waiting for something. It is outside the scope of this manual to present a complete description of the command packet data format. The basic structures and the command codes are defined in K_STRUCT.H

The microkernel maintains a list of free command packets, implemented as a list based channel. A pointer to this channel is exported in the C variable K_ArgsFreeP. Other PRHI-processes can get command packets from this pool, but they must never wait for the channel (i.e. always use PRHI_GET). If the list is empty, correct behavior is to call YIELD and try again later.

In the Virtuoso network, the Srce field of a command packet identifies the sending node, and it is used as a return path for reply messages. The same field also has a secondary function: since all packets sent or received through the network are allocated from the K_ArgsFree list, they should be deallocated after use. The network transmitters always free a packet after it has been sent. The microkernel deallocates a packet if the Scre field is not zero. Consequently, command packets not allocated from the free list must have their Scre field set to zero to prevent deallocation.

Note: we are aware of the fact that this logic is a bit confusing. Future versions of the microkernel will probably use a separate flag to indicate if a packet was dynamically allocated.

Interrupt handlers and PRHI processes can request a microkernel service by building a command packet, and pushing a pointer to it on the microkernel input channel. The only services that can be safely called are the equivalents of KS_Signal and KS_SignalM. Also note that using events will be faster than signalling.

The code fragments below show how to perform a KS_Signal() or KS_SignalM() call from within the ISR. In this example the command packet is created and initialized in C, but the same thing could be done entirely in assembly language

The function ‘install_my_isr()‘ is called to initialize the command packet and install the ISR:

```
K_ARGS CP1, *CP1P;
K_SEMA SLIST1 [] = {SEMA1, SEMA2, ..., ENDLIST};
extern void my_isr (void);
void install_my_isr(...)
{
```



```
 ...
 /* create a pointer to the command packet */
CP1P = &CP1;
/* initialize CP1 for a KS_Signal (SEMA1) service */
CP1.Srce = 0;
CP1.Comm = SIGNALS;
CP1.Args.s1.sema = SEMA1;
/* or for a KS_SignalM (SLIST1) service */
CP1.Scre = 0;
CP1.Comm = SIGNALM;
CP1.Args.s1.list = SLIST1;
/* install the ISR */
KS_EnableISR (..., my_isr);
...
}
```

For the ISR, something like the code listed below will be required:

```
.extern _CP1P;
.extern _K_ArgsP;
.global _my_isr
...
_my_isr:
...
i4 = dm(_K_ArgsP); i4 contains pointer to channel
r2 = dm(_CP1P); r2 contains data
PRHI_PSH; signals semaphore
...
```

21.8. Additional microkernel features on the 21060

21.8.1. Use of the PC stack and the counter stack

It is possible to use these stacks at the task level. They are part of the normal task context, and are swapped accordingly.

This implies that it is also allowed to use all optimisation levels provided for by the compiler.

There is one restriction to the use of the PC stack: the nanokernel and microkernel use it internally, so all tasks should leave 2 entries on the PC stack for internal use.

21.8.2. Extended context

It is also possible to use the circular buffer mechanism at the task level. Because this implies adding the m, I and b registers to the task swap set (CSAVE), considerably increasing the time necessary to perform a task swap, this is kept as an option for the programmer. An extra system group has been defined to serve this purpose. This task group is called ‘FPU’. Tasks that require circular buffering, should be defined as members of this group in SYSDEF.

There are some limitations, however:

- i6 and i7 should never be used for circular buffering.
- m6 and m7 are supposed to keep their original values.
- before calling a kernel service in a task using circular buffering, all I registers should be set to 0, and m5, m6, m7, m13, m14 and m15 should be set to their default values. This is required by the C-compiler.

22. Alphabetical List of nanokernel entry points

In the pages to follow, all Virtuoso nanokernel entry points are listed in alphabetical order. While some are C-callable, most of them are not.

- BRIEF . . . . . Brief functional description
- CLASS. . . . . One of the Virtuoso nanokernel service classes of which it is a member.
- SYNOPSIS . . . . . The ANSI C prototype (C-callable), or

  Assembly language calling sequence
- RETURN VALUE . . The return value, if any (C-callable only).
- ENTRY CONDITIONSRequired conditions before call
- EXIT CONDITIONS. Conditions upon return of the call
- DESCRIPTION . . . A description of what the Virtuoso nanokernel service does when invoked and how a desired behavior can be obtained.
- EXAMPLE . . . . . One or more typical Virtuoso nanokernel service uses.
- SEE ALSO. . . . . List of related Virtuoso nanokernel services that could be examined in conjunction with the current Virtuoso nanokernel service.
- SPECIAL NOTES. . Specific notes and technical comments.

22.1. start_process

- BRIEF . . . . . Initialize and start a nanokernel process
- CLASS . . . . . Process management
- SYNOPSIS . . . . . void start_process (void *stack, int stacksize, void entry(void), int i1, int i2);
- DESCRIPTION . . . This C function initializes the process control structure of a process, and subsequently starts it. The entry point, the stacksize, the initial values for i1 and i2 and some internal variables are written to the PCS. This call returns when the started process deschedules or terminates.
- RETURN VALUE . . none
- EXAMPLE . . . . . In this example, two processes using the same code but different parameters are initialized and started.

```
int adc1[100];/* stack for first process */
int adc2[100];/* stack for second process */
extern void adc_proc (void); /* process code */
extern struct adc_pars ADC_Params [2]; /* parameter structs */

start_process (adc1,100, adc_proc, &ADC_Params [0], 0);
start_process (adc2,100, adc_proc, &ADC_Params [1], 0);
```

- SEE ALSO. . . . . .
- SPECIAL NOTES . . This function cannot be used from within a high priority nanokernel process. It must be called from the C main () function or by a microkernel task only.

22.2. ENDISR1

- BRIEF . . . . . Terminates an ISR and conditionally invokes the process swapper
- CLASS. . . . . Interrupt service management
- SYNOPSYS ENDISR1

  ENDISR1 is defined in macro.h
- DESCRIPTION . . . This entry point must be called to terminate an ISR. A nanokernel process swap is performed IFF
  - the calling ISR interrupted the PRLO process,
  - a high priority process is ready to execute,
  - there are no more nested interrupts.
- ENTRY CONDITIONS The ISR should have saved the interrupted context so that the exit sequence listed below would correctly terminate the ISR.
- EXIT CONDITIONS . This call terminates the ISR and does not return.
- EXAMPLE . . . . . .
- SEE ALSO. . . . . .
- SPECIAL NOTES . .

22.3. K_taskcall

- BRIEF . . . . . Send a command packet to the microkernel process
- CLASS . . . . . Process communication
- SYNOPSYS . . . . . void K_taskcall (K_ARGS *A);
- DESCRIPTION . . . This C-callable function is used by all KS_ services to send command packets to the microkernel process.
- RETURN VALUE . . none
- EXAMPLE . . . . . .
- SEE ALSO. . . . . PRLO_PSH
- SPECIAL NOTES . . This function must be called by microkernel tasks only.

22.4. KS_DisableISR

- BRIEF . . . . . Remove an ISR from the interrupt vector table
- CLASS. . . . . Interrupt service management
- SYNOPSYS . . . . . void KS_DisableISR (int isrnum);
- DESCRIPTION . . . Disables the specified interrupt in IMASK, and clears the corresponding entry in the interrupt vector table.
- RETURN VALUE . . none
- EXAMPLE . . . . .

```
KS_DisableISR (15);/* Disable ISR for link buffer 3/DMA Channel 5*/
```

- SEE ALSO. . . . . KS_EnableISR,
- SPECIAL NOTES . . Interrupt numbers are defined by their bit position in IMASK.

22.5. KS_EnableISR

- BRIEF . . . . . Install an ISR and enable the corresponding interrupt.
- CLASS . . . . . Interrupt service management
- SYNOPSYS . . . . . void KS_EnableISR (int isrnum. void isr (void));
- DESCRIPTION . . . This C function is used to install or replace an interrupt handler. It takes two parameters: an interrupt number, and a pointer to an ISR. The pointer is entered into the interrupt vector table.
- RETURN VALUE . . none
- EXAMPLE . . . . . .

```
extern void _host_irqh(void);
KS_EnableISR (8, _host_irqh);
```

- SEE ALSO. . . . . KS_DisableISR
- SPECIAL NOTES . . Interrupt numbers are defined by their bit position in IMASK.

22.6. PRHI_GET

• BRIEF . . . . . Remove next packet from linked list channel

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . PRHI_GET

PRHI_GET is defined in macro.h

• DESCRIPTION . . . If the channel is not empty, the first packet is removed from the linked list, the Z flag is reset, and a pointer to the packet is returned. If the channel is empty, the Z flag is set and a NULL pointer is returned. The calling process is never swapped out as a result of calling this service.

• ENTRY CONDITIONS

i4 = pointer to linked list channel struct

• EXIT CONDITIONS . r0, r2, r8, i4 ASTAT are corrupted

If the list is not empty:

r2 = pointer to removed list element
the Z flag is cleared

If the list is empty:

r2 = 0
the Z flag is set

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm (CHANNEL);
PRHI_GET;
```

• SEE ALSO. . . . . PRHI_GETW, PRHI_PUT

• SPECIAL NOTES . . This service must not be called from the low priority context.

22.7. PRHI_GETW

• BRIEF . . . . . Get next packet from linked list channel, or deschedule

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . PRHI_GETW

PRHI_GETW is defined in macro.h

• DESCRIPTION . . . If the channel is not empty, the first packet is removed from the linked list and a pointer to it is returned. If the channel is empty, the calling process is swapped out and set to wait for the channel. It will be rescheduled by the next call to the PRHI_PUT service on the same channel.

• ENTRY CONDITIONS

i4 = pointer to linked list channel struct
i3 = pointer to PCS of calling process

• EXIT CONDITIONS.

i1 = pointer to list element
r0, r1, r2,r8, ASTAT, i4 are corrupted

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm (CHANNEL);
PRHI_GETW;
```

• SEE ALSO. . . . . PRHI_GET, PRHI_PUT

• SPECIAL NOTES . . This service must not be called from the low priority context or by an ISR.

22.8. PRHI_POP

• BRIEF . . . . . Remove next element from a stack channel

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . PRHI_POP

PRHI_POP is defined in macro.h

• DESCRIPTION . . . If the stack is not empty, the top element is removed and returned to the caller. The Z flag is reset. If the stack is empty, the Z flag is set and an undefined value is returned. The calling process is never swapped out as a result of calling this service.

• ENTRY CONDITIONS

i4 = pointer to stack channel struct

• EXIT CONDITIONS . r8, r0, r1, r2 i4, ASTAT are corrupted

If the stack is not empty:

r2 = the element removed from the stack
the Z flag is cleared

If the stack is empty:

r2 = undefined
the Z flag is set

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm(CHANNEL);
PRHI_POP;
```

• SEE ALSO. . . . . PRHI_POPW, PRHI_PSH

• SPECIAL NOTES . . This service must not be called from the low priority context.

22.9. PRHI_POPW

• BRIEF . . . . . Remove next element from a stack channel, or deschedule

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . PRHI_POPW

PRHI_POPW is defined in macro.h

• DESCRIPTION . . . If the stack is not empty, the top element is removed and returned to the caller. If the stack is empty, the calling process is swapped out and set to wait for the channel. It will be rescheduled by the next call to the PRHI_PSH service on the same channel.

• ENTRY CONDITIONS

i4 = pointer to stack channel struct
i3 = pointer to PCS of calling process

• EXIT CONDITIONS.

i1 = element removed from the stack
r0, r1, r2, r8, ASTAT, i4 are corrupted

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm(CHANNEL);
PRHI_POPW;
```

• SEE ALSO. . . . . PRHI_POP, PRHI_PSH

• SPECIAL NOTES . . This service must not be called from the low priority context or by an isr.

22.10. PRHI_PUT

• BRIEF . . . . . Add a packet to a linked list channel

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . PRHI_PUT
PRHI_PUT is defined in macro.h

• DESCRIPTION . . . If a process is waiting for the channel, the pointer to the packet is passed on, and the waiting process is rescheduled. Otherwise the packet is linked in at the head of the list. In either case, control returns to the caller.

• ENTRY CONDITIONS

i4 = pointer to channel
r2 = pointer to packet to add to the list

• EXIT CONDITIONS.

r0, r1, r8, i4, ASTST are corrupted

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm (CHANNEL);
r2 = dm (PACKET);
PRHI_PUT
; the packet is added to the list
```

• SEE ALSO. . . . . PRHI_GET, PRHI_GETW

• SPECIAL NOTES . . This entry point must not by called from the low priority context, but it can be used by interrupt handlers.

The first word of the packet is used as a link pointer, and will be overwritten.

22.11. PRHI_PSH

• BRIEF . . . . . Push a word on a stack channel

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . PRHI_PSH

PRHI_PSH is defined in macro.h

• DESCRIPTION . . . If a process is waiting for the channel, the data word is passed on, and the waiting process is rescheduled. Otherwise the data word is pushed on the stack. In either case, control returns to the caller.

• ENTRY CONDITIONS

i4 = pointer to channel
r2 = data word to push

• EXIT CONDITIONS.

r0, r1, r4 and r8 are corrupted
All other registers are preserved

• EXAMPLE . . . . . .

```
#include “macro.h”
.extern _K_ArgsP                ; microkernel input stack
; send a command packet to the microkernel
; assume i0 points to the command packet
i4 = dm (_K_ArgsP);
r2 = dm (0,i0);
PRHI_PSH;
```

• SEE ALSO. . . . . PRHI_POP, PRHI_POPW

• SPECIAL NOTES . . This entry point must not by called from the low priority context, but it can be used by interrupt handlers.

22.12. PRHI_SIG

• BRIEF . . . . . Send an event on a signal channel

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . PRHI_SIG

PRHI_SIG is defined in macro.h

• DESCRIPTION . . . If a process is waiting for the channel, it is rescheduled (put at the tail of the process FIFO). Otherwise the event count is incremented. In either case, control returns to the caller.

• ENTRY CONDITIONS

i4 = pointer to channel

• EXIT CONDITIONS.

r0, r1, r2, i4, ASTAT are corrupted
All other registers are preserved

• EXAMPLE . . . . . .

```
#include “macro.h”
i4 = dm (SYNC_CHAN);
PRHI_SIG;
```

• SEE ALSO. . . . . PRHI_WAIT

• SPECIAL NOTES . . This entry point must not by called from the low priority context, but it can be used by interrupt handlers.

22.13. PRHI_WAIT

• BRIEF . . . . . Consume an event from a signal channel, or deschedule

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . PRHI_WAIT

PRHI_WAIT is defined in macro.h

• DESCRIPTION . . . If the event counter is not zero, it is decremented an control returns to the caller. If the event counter is zero, the calling process is swapped out and set to wait for the channel. It will be rescheduled by the next call to the PRHI_SIG service on the same channel.

• ENTRY CONDITIONS

i4 = pointer to signal channel struct
i3 = pointer to PCS of calling process

• EXIT CONDITIONS.

r0, r1, r2 are corrupted

• EXAMPLE . . . . . .

```
#include “macro.h”
; wait for event on SYNC_CHAN
i4 = dm(SYNC_CHAN);
PRHI_WAIT;
; the event has happened
```

• SEE ALSO. . . . . PRHI_SIG

• SPECIAL NOTES . . This service must not be called from the low priority context or by an ISR.

22.14. PRLO_PSH

• BRIEF . . . . . This call is for internal use only, and is not exactly the equivalent of PRHI_PSH for the PRLO process. This call assumes that a PRHI process is waiting for the channel, and will crash the system if there isn't. PRLO_PUSH is used by the K_taskcall function to send command packets from a task to the microkernel process.

22.15. YIELD

• BRIEF . . . . . Yield CPU to next nanokernel process

• CLASS . . . . . Process management

• SYNOPSIS . . . . . YIELD

YIELD is defined in macro.h

• DESCRIPTION . . . The calling process is swapped out and added to the tail of the process FIFO. The process at the head of the FIFO is swapped in. This may be the same process, if it was the only one ready to execute.

• ENTRY CONDITIONS

i3 = pointer to PCS of calling process

• EXIT CONDITIONS.

• EXAMPLE . . . . . This example shows how to avoid a redundant YIELD operation, by testing the process FIFO

```
#include “nanok.h”
#include “macro.h”
;
r0 = dm(NANOK_HEAD);
r1 = 0;
comp(r0,r1);                    ; test head of process FIFO
if eq jump label;
YIELD                           ; yield if there is another process
label: ...
```

• SPECIAL NOTES . . This service must not be called from the low priority context or by an isr.

23. Predefined drivers

23.1. Virtuoso drivers on the 21060

Drivers are the interface between the processor and peripheral hardware, and the application program. They normally serve two purposes: data-communication and synchronization. As polling is not a recommended practice in a real-time system, most drivers will use interrupts in one way or another.

The Virtuoso system does not provide a standard interface to drivers - this allows the application writer to optimize this important part of their implementation. Some basic services, that will be required for almost all drivers, are provided. Most low-level details have already been described in the previous sections on interrupt handling and communication with the microkernel. At the higher level, a typical driver can usually be divided into three functional parts:

1. The first component is a function to install the driver. This should initialize the hardware and any data structures used, and install interrupt handlers for the driver. A call to this function is usually placed inside a driver statement in the system definition file. The SYSGEN-utility copies this call into a function named init_drivers() it generates in the node#.c files. The init_drivers() subroutine is called by kernel_init() just before it returns.

2. Most drivers will provide one or more subroutines that can be called from the task level, and that implement the actual functionality of the driver. At some point, these functions will call KS_EventW() or KS_Wait() to make the calling task wait for the completion of the driver action.

3. One or more interrupt handlers are required to generate the events or signals waited for by these subroutines.

In the simplest case, the only actions required from the ISR will be to service the hardware and to reschedule a waiting task, and all data handling and protocol implementation can be done at the task level. This method can be used if the interrupt frequency is not too high (< 1000Hz).

For higher data rates, some of the task code should be moved to the ISR, in order to reduce the number of task swaps. In most cases, the actions, required from the interrupt handler will not be the same for each interrupt, and some form of state machine will have to be implemented into the ISR.

If the number of possible states grows, it is often much easier to use one or more PRHI-processes to implement the protocol. Processes can wait for

interrupts at any number of places in their code, and each of these points represents a state of the system. As an example, the Virtuoso network driver have been designed using this method.

A number of device drivers are provided with this release of Virtuoso Clas-sico /VSP for SHARC:

- the timer device driver,
- two host interface device drivers,
- unidirectional DMA-based link drivers, both for internal and external memory,
- a bidirectional link driver (internal memory only),
- a unidirectional core-based link driver,
- a unidirectional DMA-based RawLink driver.

23.1.1. The timer device driver

void timer_drv (int timer_prio);

Because of the possibility to use 2 different priorities for the timer IRQ, a parameter had to be added to the driver. The possible values of timer_prio are TMZLI, indicating a low timer priority, or TMZHI, indicating the high timer priority.

The timer device driver is needed for time-out features of some kernel services and for kernel timer services.

The two procedures to read out the timer value are:

- KS_HighTimer ()
- KS_LowTimer ()

In high resolution, the number of timer counts are returned. On the 21060, the count is equal to a period of the clock frequency.

In low resolution, the number of kernel ticks are returned. A kernel tick is a multiple of timer count and defined in the main() function. As this value is a 32-bit wraparound value, it is more interesting to calculate the difference between two values read out consecutively. However, to facilitate this, kernel service KS_Elapse() is written for this purpose.

See the Alphabetical List of Virtuoso kernel services earlier in this manual for a full description of these kernel services.

Event number 48 is reserved exclusively for the timer device driver.

23.1.2. The host interface device driver

Two host interface drivers are currently available:

- void BW21k_host(int irq_num);
- void Alex21k_host(int irq_num);

The former is used for all currently supported Bittware boards (EZ-Lab, Snaggletooth and Blacktip), while the latter is to be used for the ALEX Computer Systems SHARC1000 board

The use of the host interface device driver is required on the root if the host services are to be used.

The parameter passed to the driver is the IRQ number on which the ISR is to be installed. For the Bittware and Alex boards, this value should be 8 by default.

For more details on the use of the host interface, see “Host server low level functions” and “Simple terminal oriented I/O” in the chapter of “Runtime libraries”.

The host interface device driver reserves event signal number 8 for its own use.

23.1.3. Netlink drivers

The NETLINK drivers are used by the kernel to communicate with other SHARCs running Virtuoso Classico/VSP. Several types are available:

1. void NetLink (int link_no, int buffer _no, int direction, int speed);

This is the unidirectional core driver for the link ports. It does not use DMA.

The arguments:

- link_no, buffer_no: link/buffer you wish to use.
- direction: RX for receive, TX for transmit.
- speed: SSPEED for single speed, DSPEED for double speed.

2. void NetLinkDMA (int link_no, int buffer_no, int direction, int speed);

This is the unidirectional DMA driver for the link ports. It can only be used for transfers from or to the internal SHARC memory.

The arguments:

- link_no, buffer_no: link/buffer you wish to use.
- direction: RX for receive, TX for transmit.
- speed: SSPEED for single speed, DSPEED for double speed.

3. void NetLinkDMAExt (int link_no, int buffer_no, int direction, int speed);

This is the unidirectional DMA driver for the link ports. This driver can be used for transfers from or to the internal and external SHARC memory.

The transfers to external memory are handled in 2 steps. First, data is transferred to/from internal memory (using DMA). In the second step, the data is transferred to/from internal from/to external memory, also using DMA.

The arguments:

- link_no, buffer_no: link/buffer you wish to use.
- direction: RX for receive, TX for transmit.
- speed: SSPEED for single speed, DSPEED for double speed.

4. void NetLinkDMAExtC (int link_no, int buffer_no, int direction, int speed);

This is the unidirectional DMA driver for the link ports. This driver can be used for transfers from or to the internal and external SHARC memory.

The transfers to external memory are handled in 2 steps. First, data is transferred to/from internal memory (using DMA). In the second step, the data is transferred to/from internal from/to external memory. This last step is a core transfer.

The arguments:

- link_no, buffer_no: link/buffer you wish to use.
- direction: RX for receive, TX for transmit.

speed: SSPEED for single speed, DSPEED for double speed.

5. void NetLinkDMA2 (int link_no, int buffer_no, int token, int speed);

This is the bidirectional driver for the link ports. You may experience problems with this drivers if you are using pre rev 2.0 silicon.

The arguments:

- link_no, buffer_no: link/buffer you wish to use.
- token: initial location of the token. Token exchange is performed automatically, but the driver needs the initial position. Values: TOKEN / NOTOKEN.

- speed: SSPEED for single speed, DSPEED for double speed.

For updates on the status of the bidirectional link driver, check the readme file included with your release.

23.1.4. Raw Link drivers

void RawLinkDMA (int link_no, int buffer_no, int speed);

This is the RAW DMA link driver for the link ports.

The arguments:

- link_no, buffer_no: link/buffer you wish to use.
- speed: SSPEED for single speed, DSPEED for double speed.

This driver does not implement any protocol, it allows the programmer to transfer (receive and transmit) specified amounts of data over the link port. Also, it is not allowed to use a RawLinkDMA on a link that is used as a NETLINK.

The services which use this driver are:

- KS_Linkin(W) (int buffer, int length, void *addr);
- KS_Linkout(W) (int buffer, int length, void *addr);

The specified lengths are in WORDS, not bytes.

23.1.5. Common remark for all link drivers

It is currently not possible to use DSPEED drivers on rev 1.x silicon. This will work on rev 0.6 and probably also on later (rev 2.0 and on) revisions.

It is possible to assign any link port to any link buffer. The driver will check if the requested buffer is already assigned to any other link. If so, it will simply return, and the driver is not loaded. This could prevent your application from running correctly, so please check the buffer assignments carefully.

It is now required that a section called ‘packets’ is allocated in internal memory to hold all the packets used by the kernel. This is done in the architecture file used to link the application. This section holds command and datapackets used to communicate between the kernel and nanokernel processes. The size of a command packet is fixed at 16 words, and the size of a data packet is defined in ‘mainx.c’ (in bytes). If there is not enough room in this section, the kernel will not operate correctly. Also see the provided examples.

24. Task Level Timings

Following is a list of task level timings of some of the kernel services provided by Virtuoso. These timings are the result of a timing measurement on a rev 1.2 ADSP-21062 board with a clock speed of 40MHz. The kernel used for these timings is the VIRTUOSO Microkernel.

All timings are in microseconds. The C compiler used is the G21k C Compiler v.3.2d from Analog Devices.

minimum VIRTUOSO call time	5
SEND/RECEIVE WITH WAIT (MAILBOX SERVICES)	
header only :**	39
8 bytes : **	40
16 bytes : **	40
32 bytes :**	40
64 bytes :**	41
128 bytes :**	41
256 bytes : **	43
512 bytes : **	46
1024 bytes :**	53
2048 bytes : **	65
4096 bytes : **	91
QUEUE OPERATIONS	
enqueue 1 byte *	8
dequeue 1 byte *	8
enqueue 4 bytes *	8
dequeue 4 bytes *	8
enqueue 1 byte to a waiting higher priority task **	26
enqueue 4 bytes to a waiting higher priority task **	26
SEMAPHORE OPERATIONS	
signal semaphore *	7
signal to waiting high pri task **	25
signal to waiting high pri task, with timeout **	33
signal to waitm (2) **	58
signal to waitm (2), with timeout **	65
signal to waitm (3) **	72

signal to waitm (3), with timeout **	78
signal to waitm (4) **	85
signal to waitm (4), with timeout **	92
RESOURCE OPERATIONS	
average lock and unlock resource *	6
MEMORY MAP OPERATIONS	
average alloc and dealloc memory page *	6

Note :

*: involves no context switch

**: involves two context switches. Timing is round-trip time.

25. Application development hints.

The easiest way to start is to copy and modify one of the supplied examples. Some of the necessary files have fixed names, so each application should be put in a separate directory.

The following files will be needed for each application:

SYSDEF:

The VIRTUOSO system definition file. The SYSGEN utility will read this file and generate NODE1.C and NODE1.H.

MAIN1.C:

This contains some more configuration options, and the C ‘main’ function. Copy from one of the examples.

A number of configuration options are defined in this file, so they can be changed without requiring recompilation of all sources (this would be necessary if SYSDEF is modified).

CLCKFREQ : this should be defined to be the clock frequency of the hardware timer used to generate the TICKS time.

TICKTIME : the TICK period in microseconds.

TICKUNIT: the TICK period in CLCKFREQ units.

TICKFREQ: the TICK frequency in Hertz.

The number of available timers, command packets and data packets are also defined in this file. How much you need of each depends on your application, but the following guidelines may be followed:

Timers are used to implement time-outs, and can also be allocated by a task.

A command packet will be needed for each timer allocated by a task. Command packets used for calling a kernel service are created on the caller’s stack and should not be predefined.

MAIN1.C also defines some variables used by the console driver tasks, the clock system, the debugger task, and the graphics system. These are included automatically if you use the standard names for the required kernel

objects.

XXX.ACH: specifies architecture file.

MAKEFILE:

The makefiles supplied in the EXAMPLES directory can easily be modified for your application. They also show how to organize things so you can optionally include the task level debugger. If you want to include the task level debugger, put the corresponding definitions out of comment:

VIRTLIB = $(LIBS)\virtosdr.lib
DD = -dDEBUG
DDD = -P “DEBUG”

and put the other definition in comment:

VIRTLIB = $(LIBS)\virtosr.lib

whereby # is the comment sign.

There are also two define-statements in the ‘mainx.c’-file, that the customer can change in order to ‘personalise’ the debugger:

define MONITSIZE 1024 /* number of monitor records */
define MONITMASK MONALL - MONEVENT /* what will be monitored */

Then remake the application, just by doing:

MAKE <Enter>.

LINKFILE: list of the object versions of all source files to be linked in the executables.

After you have done make-ing your application, you can run the application by typing:

> 21khost -rlsi test

26. Virtuoso on the Intel 80x86

26.1. Notes over PC interrupt drivers

ISR_C.C : code that installs the ISR's (Interrupt Service Routines) coded in the ISR_A.ASM file.

ISR_A.ASM : code for the interrupt routines that handle the DOS timer.

Virtuoso takes over the 18.2 Hz timer from DOS (it effectively doesn't run anymore) and upscales it to 1000 Hz for internal use. For some purposes DOS however needs this timer (e.g. when you use the floppy disk inside Virtuoso, the disk spinning motor would run forever because the time-out value is never reached as the associated timer doesn't run). For this reason the ISR adjusts from time to time the correct time in the 18.2 Hz timer.

26.2. Warning when using Virtuoso on a PC

When you use Virtuoso on a PC, you must be aware that DOS is still there to provide access to the host resources, such as the disk. The problem is that whenever you issue a service request to DOS, you enter a critical section and all kernel activity is suspended. For example when you write a file, DOS initiates an access to the disk. As long as this operation is not finished, DOS waits. So, while you can use Virtuoso to develop the logic of a real-time application, you can't really use it to build a hard real-time application on a PC. This can only be achieved on boards that don't require DOS. Note that none of the real-time kernels on the market achieve this. Whatever solution is chosen, it can't solve the fundamental problem that DOS is sitting there. At best you can use a real-time kernel on a PC to achieve a soft real-time application, unless you don't make any DOS call.

In Virtuoso, DOS services (stdio and graphics) are accessed through a DOS server task. This task is normally defined with the highest priority.

27. Virtuoso on the Motorola 56K DSP

27.1. Virtuoso versions on 56K

Virtuoso Micro/SP is available for the 56K. A port of the nanokernel on 56K is available in a single-processor version, Virtuoso Nano/SP.

27.2. DSP 56001 Chip Architecture

This section contains a brief description of the DSP5600x processor architecture. It is not intended to be a replacement of the Processor's User Manual, but as a quick lookup for the application programmer. Detailed information can be found in the "DSP56000/DSP 56001 User's Manual" from Motorola.

The DSP56001 has a Harvard architecture (separated program and data addressing) with multiple internal buses.

Interfacing to the outside world is done via one 47-pin expansion port (port A) and 24 additional I/O pins, which are divided between ports B and C. Of the 24 pins, 15 are assigned to port B. These pins can be used as general-purpose I/O pins or as host MPU/DMA interface pins. Port C consists of 9 pins, also usable as general I/O pins or as SCI (Serial Communications Interface) and SSI (Synchronous Serial Interface) pins.

The heart of the processor consists of 3 execution units operating in parallel: the data arithmetic logic unit (ALU), the address generation unit (AGU) and the program controller. In addition to these 3 execution units, the 56K has also six on-chip memories, three on-chip MCU-style peripherals (SCI, SSI, host interface), a clock generator, and seven buses.

The main components of the DSP56001 are:

- Data and Address Buses
- Data ALU
- Address Generation Unit
- X Data Memory
- Y Data Memory
- Program Controller
- Program Memory
- I/O:

  Memory Expansion (Port A)
  General Purpose I/O (Ports B and C)

Host Interface
Serial Communication Interface
Synchronous Serial Interface

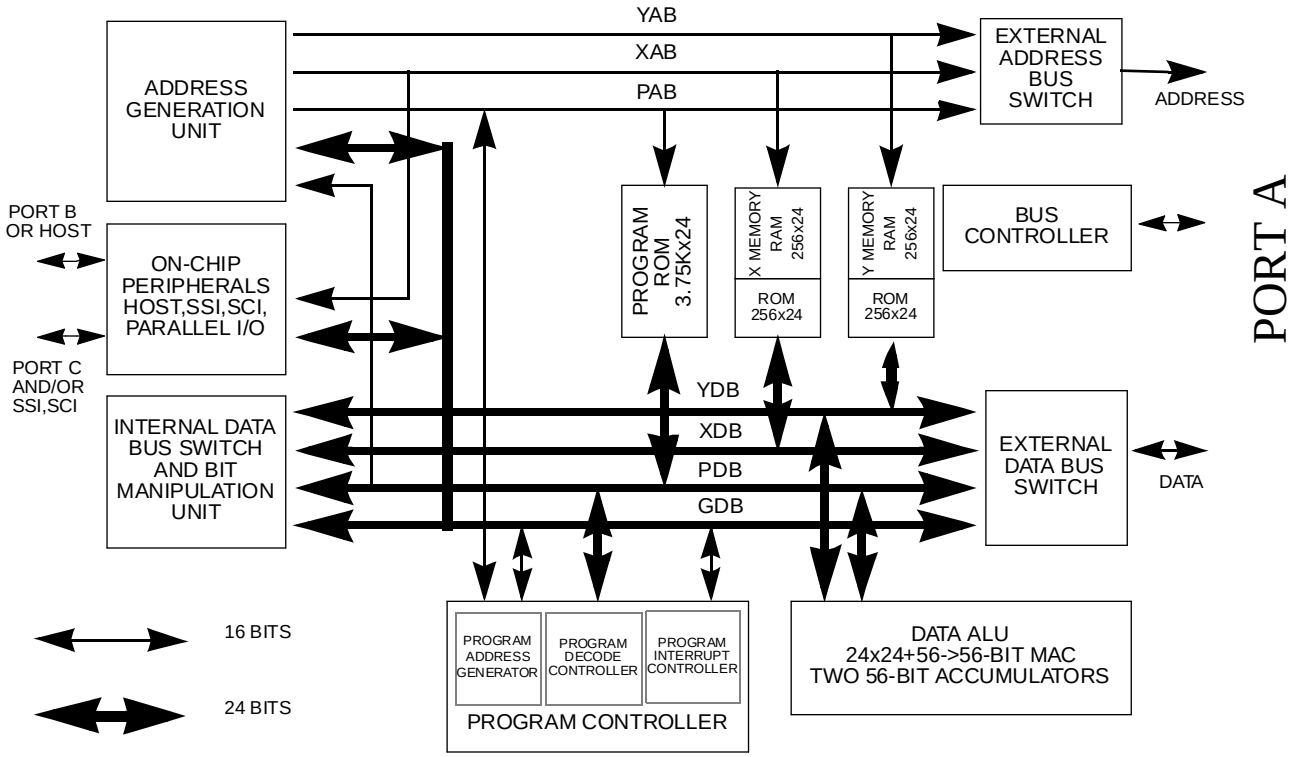
The major components of the Data ALU are as follows:

- Four 24-bit input registers
- A parallel, single-cycle, nonpipelined multiply-accumulator/logic unit (MAC)
- Two 48-bit accumulator registers
- Two 8-bit accumulator extension registers
- Two data bus shifters/limiter circuits

The major components of the Address Generation Unit are:

- Address Register Files
- Offset Register Files
- Modifier Register Files
- Address ALU
- Address Output Multiplexer

DSP56000 Block Diagram:



27.3. DSP56001 software architecture

The programming model indicates that the DSP56001processor architecture can be viewed as three functional units operating in parallel, i.e. the Data ALU, the AGU, and program controller. The goal of the instrucion set is to keep each of the units busy each instruction cycle, achieving maximum speed and minimum program size.

The eight main data ALU registers (X0..1, Y0..1, A0..1,B0..1) are 24 bits wide. The A and B registers have extension registers, which are 8 bits wide each. When these extensions are used as source operands, they occupy the low-order portion of the word, and the high-order portion is sign extended. When used as destination operands, only the low-order portion of the word is received, while the high order portion is not used.

The 24 AGU registers are 16 bits wide, and may be accessed as word operands for address, address modifier and data storage. When used as a source operand, the registers occupy the low-order portion of the word, and the high-order portion is zero-filled. When used as destination operand, the registers only receive the low-order part of the word, the high-order part is not used.

The program control registers:

- the 8-bit OMR (Operating Mode Register) may be accessed as a word operand, but not all 8 bits are defined.
- the LC (Loop Counter), LA (Loop Address Register), SSH (System Stack High) and SSL (System Stack Low) registers are 16 bits wide and may be accessed as word operands. When used as a source operand, these registers occupy the low-order portion of the 24-bit word, the high-order portion is zero. When used as a destination operand, they receive the low-order portion of the 24-bit word, and the high-order portion is not used.

The Loop Counter Register is a special 16-bit counter used to specify the number of times a program loop is to be repeated. The Loop Address register indicates the location of the last instruction word in a hardware program loop.

The System Stack is a separate internal memory, divided into 2 16-bit wide parts, each 15 locations deep. The SSH stores the PC contents, and the SSL stores the stores the SR contents for subroutine calls and long interrupts. In order to be able to use this hardware stack and the associated commands (hardware DO loops, ...) in a multitasking environment, it is saved, along with other registers, to the user stack of the calling task when this is swapped out. This way, no conflicts can occur if the next task also makes use of the hardware stack.

The SP (Stack Pointer) register is a 6-bit register that may be accessed as a word operand. It indicates the location of the top of the SS and the status of the stack (empty, full, underflow and overflow conditions).

- the PC (Program Counter) is a special 16-bit program control register is always referenced implicitly as a short-word operand. It contains the address of the next location to be fetched from program memory space.
- the 16-bit SR (Status Register) has the MR (System Mode Register) occupying the high-order 8 bits and the CCR (Condition Code Register) occupying the lower 8 bits. The SR may be accessed as a word operand. The MR and CCR may be accessed individually as word operands.

The MR is a special-purpose register defining the current system state of the processor. Special attention is given to the Interrupt Mask Bits. These reflect the current priority level of the processor and indicate the IPL (Interrupt Priority Level) needed for an interrupt source to interrupt the processor. These bits can be modified under software control.

The CCR is a special-purpose register that defines the current user state of the processor.

Contents of the Status Register:

Bit Nr.	Code	Meaning	Register
15	LF	Loop Flag	MR
14	*	Reserved	MR
13	T	Trace Mode	MR
12	*	Reserved	MR
11	S1	Scaling Mode	MR
10	S2	Scaling Mode	MR
9	I1	Interrupt Mask	MR
8	I0	Interrupt Mask	MR
7	*	Reserved	CCR
6	L	Limit	CCR
5	E	Extension	CCR
4	U	Unnormalized	CCR
3	N	Negative	CCR
2	Z	Zero	CCR
1	V	Overflow	CCR
0	C	Carry	CCR

27.3.1. Addressing Modes

The DSP56K provides three different addressing modes:

- Register Direct
- Address Register Indirect
- Special

Register Direct

- Data or Control Register Direct
- Address Register Direct

Address Register Indirect

■ No Update	(Rn)
■ Postincrement by 1	(Rn)+
■ Postdecrement by 1	(Rn)-
■ Postincrement by Offset Nn	(Rn)+Nn
■ Postdecrement by Offset Nn	(Rn)-Nn
■ Indexed by Offset Nn	(Rn+Nn)
■ Predecrement by 1	-(Rn)

Special Addressing

- Immediate Data
- Absolute Address
- Immediate Short
- Short Jump Address
- Absolute Short
- I/O Short
- Implicit Reference

The DSP56K address ALU supports linear, modulo, and reverse-carry arithmetic types for all address register indirect modes. These arithmetic types easily allow the creation of data structures in memory for FIFOs, delay lines, circular buffers, stacks and bit-reversed FFT buffers. Each address register Rn has its own modifier register Mn associated with it. The contents of this modifier register determines the type of arithmetic to be performed for addressing mode calculations.
The following modifier classes are supported:

- Linear Modifier
- Modulo Modifier
- Reverse Carry Modifier

27.3.2. I/O Memory

The on-chip peripheral registers occupy the top 64 locations of the X data memory ($FFC0-$FFFF).

The off-chip peripheral registers should be mapped into the top 64 locations to take advantage of the move peripheral data instruction (MOVEP).

27.3.2.1. PORT A

Port A is the memory expansion port that can be used for either memory expansion or for memory-mapped I/O. An internal wait-state generator can can be programmed to insert up to 15 wait states (BCR register) if access to slower memory or I/O devices is required. The Bus Wait signal allows an external device to control the number of wait states inserted in a bus access operation. Bus arbitraion signals allow an external device to use the bus while internal operations continue using internal memory.

The expansion bus timing is controlled by the Bus Control Register (BCR), located at X:$FFFE.

This BCR consists of 4 subregisters:

Subregister	Bits	Controls	Location of Memory
External X Memory	12-15	External X Data Memory	X:$200 - X:$FFC0
External Y Memory	8-11	External Y Data Memory	Y:$200 - Y:$FFC0
External P Memory	4-7	External Program Memory	P:$F00 - P:$FFFF
External I/O Memory	0-3	External Peripherals	Y:$FFC0 - $FFFF

27.3.2.2. PORT B

General-Purpose I/O

Port B can be viewed as 3 memory-mapped registers that control 15 I/O pins. After a reset, port B is configured as general-purpose I/O with all 15 pins as input by clearing all 3 registers.

The 3 registers are:

	Location	Bits	Use
Port B Control Register (PBC)	X:$FFE0	0	0: Parallel I/O
			1: Host Interface
Port B Data Direction Register	X:$FFE2	0-14	0: Input
(PBDDR)			1: Output
Port B Data Register	X:$FFE4	0-14	Data

Host Interface

The HI is a byte-wide, full-duplex, double-buffered, parallel port which may be directly connected to the data bus of a host processor. It is asynchronous and consists of 2 banks of registers - one bank accessible to the host processor, and a second bank accessible to the DSP CPU.

The DSP CPU views the HI as a memory-mapped peripheral occupying 3 24-bit words in data memory space. It may be used as a normal memory-mapped peripheral, using polling or interrupt techniques. The memory-mapping allows the DSP CPU to communicate with the HI registers using standard instructions and addressing modes. The MOVEP instruction allows HI-to-memory and memory-to-HI transfers without using an intermediate register.

The 3 registers are the following:

	Location	Bits	Use
Host Control Register (HCR)	X:$FFE8	3-4	DSP CPU HI flags
		2	Host Command Interrupt
		1	Host Transmit Interrupt
		0	Host Receive Interrupt
Host Status Register (HSR)	X:$FFE9	7	DMA
		3-4	HOST HI flags
		2	Host Command Pending
		1	Host Transmit Data Empty
		0	Host Receive Data Full
Host Receive Data Register	X:$FFEB		
Host Transmit Data Register	X:$FFEB		

27.3.2.3. PORT C

Port C is a triple-function I/O port with nine pins. Three of these nine pins can be configured as general-purpose I/O or as the serial communications interface (SCI). The other six pins can be configured as general-purpose I/O or as the synchronous serial interface.

Port C can be viewed as 3 memory-mapped registers that control 9 I/O pins. After a reset, port C is configured as general-purpose I/O with all 9 pins as input by clearing all 3 registers.

The 3 registers are:

	Location	Bits	Use
Port C Control Register (PCC)	X:$FFE1	0-2	0: Parallel I/O
			1: SCI
		3-8	0: Parallel I/O
			1: SSI
Port C Data Direction Register	X:$FFE3	0-9	0: Input
(PCDDR)			1: Output
Port C Data Register	X:$FFE5	0-8	Data

27.3.3. Exceptions

Exceptions may originate from any of the 32 vector addresses listed in the following table. The corresponding interrupt starting address for each interrupt source is shown also.

Interrupt Starting Address	IPL	Interrupt Source
$0000	3	Hardware RESET
$0002	3	Stack Error
$0004	3	Trace
$0006	3	SWI
$0008	0-2	IRQA
$000A	0-2	IRQB
$000C	0-2	SSI Receive Data

$000E	0-2	SSI Receive Data with Exception Status
$0010	0-2	SSI Transmit Data
$0012	0-2	SSI Transmit Data with Exception Status
$0014	0-2	SCI Receive Data
$0016	0-2	SCI Receive Data with Exception Status
$0018	0-2	SCI Transmit Data
$001A	0-2	SCI Idle Line
$001C	0-2	SCI Timer
$001E	3	NMI - Reserved for Hardware Development
$0020	0-2	Host Receive Data
$0022	0-2	Host Transmit Data
$0024	0-2	Host Command (Default)
$0026	0-2	Available for Host Command
$003C	0-2	Available for Host Command
$003E	3	Illegal Instruction

The 32 interrupts are prioritized into 4 levels. Level 3, the highest priority level is not maskable. Levels 2-0 are maskable. The priority level of an interrupt can be programmed to 0,1,2 or disabled.

Interrupts are processed in the following way:

- a hardware interrupt is synchronized with the DSP clock, and the interrupt pending flag for that particular interrupt is set. An interrupt source can have only 1 interrupt pending at any given time.
- all pending interrupts are arbitrated to select the interrupt which will be processed. The arbiter automatically ignores any interrupts with an IPL lower than the interrupt mask level in the SR and selects the remaining interrupt with the highest IPL.
- the interrupt controller freezes the PC and fetches 2 instructions at the 2 interrupt vector addresses associated with the selected interrupt.
- the interrupt controller puts the 2 instructions into the instruction stream and releases the PC. The next interrupt arbitration is then begun.

Two types of interrupt may be used: fast and long. The fast routine consists of the 2 automatically inserted interrupt instruction words. In this case, the state of the machine is not saved on the stack.

A JSR within a fast interrupt routine forms a long interrupt. A long interrupt routine terminates with an RTI instruction to restore the PC and SR from the System Stack and return to normal program execution.

27.4. Relevant documentation

1. “DSP56000/DSP56001 Digital Signal Processor User’s Manual”, Motorola Inc., 1990.

2. “G56KCC - Morotola DSP56000/DSP56001 Optimizing C compiler User’s Manual”, Motorola Inc., 1991.

27.5. C calling conventions and use of registers

This section contains the following topics:

- Storage Allocation
- Register Usage
- Subroutine Linkage
- Procedure rologue and Epilogue
- Stack Layout

27.5.1. Storage Allocation

The Basic C data types are implemented as follows:

char	24 bits, signed
unsigned char	24 bits, unsigned
short	24 bits, signed
unsigned short	24 bits, unsigned
int	24 bits, signed
unsigned int	24 bits, unsigned
long	48 bits, signed
unsigned long	48 bits, unsigned
float	48 bits
double	48 bits
pointer (address)	24 bits, max value 0xFFFF

27.5.2. Register Usage

The compiler register reserves the following machine registers for particular uses:

Register	Use
R0	Frame Pointer
R6	Stack Pointer
R7	Structure Return Address
R1-R5, R7	Register promotion by optimiser
N0-N7	Code Generator Temporary
M0-M7	Unused by compiler, dangerous side effects
A	48-bit function return value, float, double or long
A1	24-bit and 16-bit return value, integer or pointer
B,X,Y	48-bit register promotion by optimiser
X1,X0,Y1,Y0	24-bit register promotion by optimiser

27.5.3. Subroutine Linkage

27.5.3.1. Preserved Registers

Every register in the set performs a specific function, thus requiring the programmer to preserve any register that is to be directly used in in-line and out-line assembly language code.

27.5.3.2. Register Return Values

A	48-bit function return value, float, double or long
A1	24-bit and 16-bit return value, integer or pointer

27.5.3.3. Parameter Passing

Information passed to C subroutines is stored in a parameter data space which is similar to the local data space. However, the data is in reverse order and each parameter is referenced via a negative offset from the frame pointer. Actual parameters are pushed onto the activation record in reverse order by the calling subroutine. (see 4.6.1 in “G56KCC - Motorola DSP56000/ DSP56001 Optimizing C Compiler User’s Manual”)

27.5.3.4. Subroutine Call sequence

Every time a C language subroutine is called, a strict calling convention is followed. The subroutine calling sequence is broken down into 3 sub-sequences that are strictly defined.

- Caller Sequence

The caller portion of the subroutine calling sequence is responsible for:

3. pushing arguments onto the activation record - in reverse order,
4. the actual subroutine call,
5. stack pointer adjustment.

Additional caller sequence when the subroutine called will return a structure:

6. allocate space in the caller's activation record to store the return structure,
7. pass the return value address in accumulator A.

- Callee Sequence

During the initial portion of the subroutine calling sequence, the callee is responsible for:

8. saving the return address (ssh) and the old frame pointer (R0),
9. updating frame and stack pointers,
10. saving the following registers, as required: B1, B0, X1, X0, Y1, Y0, R1-R5 and R7.

- Return Sequence

During the final portion of the subroutine calling sequence, the callee is responsible for:

11. placing the return value in accumulator A,
12. testing the return value. This optimises the case where function calls are arguments to conditional operators.

Additional callee sequence when the subroutine called will return a structure:

13. the return value is not passed in accumulator A. A copy of the return structure is placed into the space allocated in the caller's activation record and pointed to by accumulator A.

27.5.4. Procedure Prologue and Epilogue

A leaf routine is a subroutine that makes no further subroutine calls. When the compiler identifies such routines, the prologue and epilogue code are optimized (no save and restore of the ssh).

For routines which have local variables packed in registers, a move instruction will be generated to save the register upon entry and restore it before exiting.

For all non-leaf routines, a move must be emitted to save ssh on the stack. When local variables exist that couldn't be packed in registers, code must be emitted to save and restore the frame pointer and the stack pointer

27.5.5. Stack Layout

Interfacing C and Assembly allows the user to utilize the benefits of both languages in programming tasks. When interfacing C and Assembly, it is essential that Assembly language, that is called from C, must match the compiler's conventions. Although not strictly required, it is recommended that all assembly language routines use the standard stack usage conventions, so that these routines can be called from within C code or vice versa.

Here is an example of a C program calling an assembly language routine:

```
extern int asmsub ();
main ()
        {
        int i, arg1, arg2, arg3;
        i = asmsub (arg1, arg2, arg3);
        ...
        }
```

The assembly language routine is declared as an ordinary external C routine. According to the compiler's naming conventions, the C code will contain a call to a global symbol named `Fasmsub`. That is, the compiler prepends an upper case letter F to each C procedure name. Therefore the assembly language must define a global symbol of this name, that is:

```
XDEF Fasmsub
Fasmsub:
< entry code (prologue) >
< body of routine >
< exit code (epilogue) >
```

When a subroutine is called, a new copy of the so-called subroutine activation record is put on the run-time stack, and returning from the subroutine removes the activation record. An activation record is the run-time representation of a C subroutine. Typically, such a record consists of the following elements:

- Parameter data space: Information passed to C subroutines is stored in a parameter data space which is similar to the local data space.

However, the data is in reverse order and each parameter is referenced via a negative offset from the nframe pointer. Actual parameters are pushed onto the activation record in reverse order by the calling subroutine.

- Old frame pointer. The old frame pointer provides a dynamic link to the calling subroutine's activation record. Once the called subroutine has completed execution, the frame pointer will be updated with this value
- Return address - which is pushed on the DSP's system stack high register. This is the return address to the calling subroutine. The return address is not saved for leaf subroutines.
- Local data space. The location of C variables that have a lifetime that extends only as long as the subroutine is active and that could not be explicitly promoted to register storage class by the optimiser.
- Register spill and compiler temporary space. This area is utilised by the compiler to store intermediate results and preserve registers.
  Note: The frame pointer (R0) points to the first element in the local data space.
  Note 2: The stack pointer (R6) points to the next available data memory location.

By default, global and static data elements are located below the run-time stack and each element is referenced by a unique label that is known at compile-time.

27.6. Interrupt Service Routines (ISR)

The two ISR levels that are normally supported by Virtuoso (ISR0 and ISR1) are not present in the version for the DSP56000 processor, since the DSP56000 supports multiple interrupt levels on its own. For more details on the different interrupts supported by the DSP56000, see section 27.3.3. of this manual, or, even more in detail, the DSP56000 User's Guide from Motorola.

27.6.1. ISR Conventions

When using self-written ISRs in conjunction with the Virtuoso kernel, there are certain conventions to follow:

- Registers that must be preserved in an ISR.
- Interrupts that must be disabled at certain sections in the ISR code.

Saving or preserving registers

There are two sorts of ISRs that can be written:

1. ISRs that stand on their own and do not make use of the kernel to give signals
2. ISRs giving signals to the kernel

In both cases, the Stack Pointer (R6) must first be incremented, prior to saving any register at the start of an ISR. This is because a critical section exists in the epilogue code of a procedure:

```
move (R6)-
move y:(R6),<reg>
```

This instruction sequence restores a previously saved register. If an interrupt occurs in between these two instructions, the Stack Pointer points to the address where the value of the preserved register is written and the first move in the ISR to save a register will overwrite that value, if no prior increment of R6 is done. The same goes for the preservation of SSH.

Keeping this potential danger in mind, the following prologue code must be used for an ISR of the first class (no kernel interaction):

```
move (R6)+              ; prior increment of the SP
move <reg>,y:(R6)+      ; repeat this instruction for
                        ; every register that must be
                        ; saved
```

At the end of the ISR, right before the RTI instruction, following epilogue code must be used:

```
move (R6)-
move y:(R6)-,<reg>      ; repeat this instruction for
                        ; every register that is saved
                        ; in the prologue code of the
                        ; ISR
```

Note, that this epilogue code is not critical anymore, provided the prologue of all ISRs start with an increment of the Stack Pointer (R6).

Which registers have to be preserved by an ISR depends on the class of ISR an on which registers are used in the ISR. If the ISR stands on its own (no signal is made to the kernel), only those registers must be preserved that are used by the ISR. In this case, the prologue and epilogue code just described are to be used. In the case the ISR gives a signal to the kernel, all registers that are used by the ISR must be preserved, except the registers R1, R2, B and X: these registers must always be saved at the start of a signalling ISR, regardless if they are used by the ISR or not, because the kernel is relying

on the fact that they are saved. The kernel expects in register B1 the event signal number, which can range from 0 to 63 inclusive. So, for a signalling ISR, next conventions must be followed:

1. First increment the Stack Pointer R6.
2. Save registers X, B, R1 and R2 in this sequence.
3. Save all other registers used by the ISR.
4. Do whatever has to be done (body of ISR).
5. Restore all registers except R2, R1, B and X. Note, however, that the last register restore may NOT contain a decrement of the Stack Pointer, because this decrement would be one too much.
6. Load register B1 with the event signal number (value 0 - 63).
7. Jump to label `Fkernel_sign_entry`, to give the signal.

An example is given for each class of ISR.

Example 1: a non-signalling ISR uses registers N0, N2, R0 and R1.

```
move (R6)+              ; prior increment of R6
move N0,y:(R6)+         ; save N0
move N2,y:(R6)+         ; save N2
move R0,y:(R6)+         ; save R0
move R1,y:(R6)+         ; save R1
<body of ISR>
move (R6)-              ; post-decrements are faster
move y:(R6)-,R1         ; restore R1
move y:(R6)-,R0         ; restore R0
move y:(R6)-,N2         ; restore N2
move y:(R6)-,N0         ; restore N0
rti                     ; finito
```

Example 2: a signalling ISR using R4 and M4 as extra registers.

```
move (R6)+              ; prior increment of R6
move X,l:(R6)+          ; save X
move B10,l:(R6)+        ; save B
move B2,y:(R6)+         ;
move R1,x:(R6)          ; save R1
move R2,y:(R6)+         ; save R2
move R4,y:(R6)+         ; save R4
move M4,y:(R6)+         ; save M4
<body of ISR>
move (R6)-              ; post-decrements are faster
```



```
move y:(R6)-,M4         ; restore M4
move y:(R6),R4          ; restore R4 - NO DECREMENT!
move #SIGNUM,B1         ; load B1 with event signal
                        ; number
jmp Fkernel_sign_entry ; signal the kernel
```

If a C procedure is called from an ISR, all registers that are not preserved across a procedure call (see paragraph 27.5.3. for a list of preserved registers), have to be saved. However, for a signalling ISR, it is not advised to make a subroutine jump to a C function from within the ISR as this would introduce needless overhead of context saving. The kernel, when jumped to by label `Fkernel_sign_entry`, will perform a context save for all non-preserved registers. In this case, it is advised to make a task that waits for an event, with kernel service `KS_EventW(n)`, and that calls this C function after it is waked up by a signal to event number `n`.

27.7. Alphabetical list of ISR related services

```
Fkernel_sign_entry
/* for entering the kernel from within an ISR */
/* single processor version only */
KS_EventW()
/* for waiting for an interrupt at the task level */
KS_EnableISR()
/* for installing an ISR */
KS_DisableISR()
/* for removing an ISR */
```


27.7.1. Fkernel_sign_entry

• SYNOPSIS . . . . . Label jumped to when entering the kernel from within an ISR

• BRIEF . . . . . This service gives a signal to the kernel with an event code numbered between 0 and 63 inclusive. A task can wait for the occurrence of such a signal by using kernel service `KS_EventW(n)`.

• EXAMPLE . . . . .

• SEE ALSO . . . . . KS_EventW

• SPECIAL NOTES . . The kernel signalling service assumes that certain conventions are followed by the ISR:

1. Stack Pointer R6 must be incremented at the very start of the ISR
2. Registers X, B, R1 and R2 have to be saved at the start of the ISR, after the prior increment of R6, with the sequence as indicated (see also previous paragraph)
3. Prior to jumping to the entry `Fkernel_sign_entry`, register B1 must be loaded with the event number (between 0 and 63 inclusive)
4. A JMP instruction must be used to jump to the entry `Fkernel_sign_entry`, not a JSR instruction. The System Stack of the processor will be managed by the kernel, so that, when returning from interrupt, the correct program address will be loaded in the Program Counter

This kernel service is only callable from an ISR written in assembly when used with the single processor version (with no nanokernel).

27.7.2. KS_DisableISR

• BRIEF . . . . . Disables to ISR to be triggered by interrupt

• SYNOPSYS . . . . .

```
void KS_DisableISR (int IRQ);
```

• DESCRIPTION . . . This C-callable service disables an ISR by writing an ILLEGAL instruction at the appropriate place in the interrupt vector table. Also, for the following interrupts, the corresponding bits in the IPR register of the processor will be changed accordingly:

- IRQA
- IRQB
- Host Command

Other interrupts can also be disabled by this service, but only in the sense that the JSR instruction at the corresponding place in the interrupt vector table will be overwritten by an ILLEGAL instruction.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
KS_DisableISR (9);
```

• SEE ALSO. . . . . KS_EnableISR

• SPECIAL NOTES . .

27.7.3. KS_EnableISR

• BRIEF . . . . . Enables an ISR to be triggered by an interrupt

• SYNOPSYS . . . . .

```
void KS_EnableISR (int IRQ,
    void (*ISR)(void),
    int PrioLevel,
    int Mode);
```

• DESCRIPTION . . . This C-callable kernel service installs an ISR by writing a JSR instruction at the appropriate place in the interrupt vector table and setting the IPR register of the processor with the correct bit-values for the actual interrupt. This service may be used to install following interrupts, together with their priority level and interrupt mode (if appropriate):

- IRQA
- IRQB
- Host Command

Other interrupts can also be installed by this service, but for them the priority level and interrupt mode is not applicable and the arguments PrioLevel and Mode are not used.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
extern void ISRDMACh1(void);
KS_EnableISR (9, ISRDMACh1, 2, 0);
```

27.7.4. KS_EventW

• BRIEF . . . . . Waits for event associated with ISR

• SYNOPSIS . . . . .

```
KS_EventW(int IRQ)
```

• DESCRIPTION . . . This C-callable kernel service can be used by a an application task to wait for a signal, given by an ISR. It forms a pair with kernel service Fkernel_sign_entry.

• EXAMPLE. . . . . .

• SEE ALSO. . . . . Fkernel_sign_entry

• SPECIAL NOTES . .

27.8. Developing ISR routines

When developing Interrupt Service Routines, the ISR conventions, described in paragraph 27.6.1. have to be followed.

The best place to install and enable an ISR, is in procedure `main()`, where predefined drivers, like the driver for the timer interrupt, are installed and enabled.

It is possible that additional initialization of registers and/or peripheral I/O has to be done. The best way to do this, is writing a C-callable procedure, that does the necessary additional initializations, and call this procedure after the call to `KS_EnableISR()`. An example of this method is the installation of the timer ISR in procedure `main()`:

```
#include “iface.h”
extern void timer0_irqh (void);
extern void timer0_init (void);
...
int main (void)
{
...
KS_EnableISR (4, timer0_irqh, IPLEVEL2, IPMNEDGE);
timer0_init();
...
}
note: When the timer is implemented by means of the
        internal timer (DSP56002), the first argument
        of the KS_EnableISR() and KS_DisableISR() has
        to be 30. See section ‘The timer device driver’.
```

27.9. The nanokernel on the 56002

Virtuoso Nano/SP is available on the E-tools Minikit. Documentation is provided separately with this product.

27.10. Predefined drivers

Two devices drivers are already added to this release of the Virtuoso kernel. They are:

- the timer device driver
- the host interface device driver

The timer device driver is needed for time-out features of some kernel services and for kernel timer services. The host interface device driver is written to be able to communicate between the host server program and the DSP56000 target board.

27.10.1. The timer device driver

Only the DSP56002 processors with versions from D41G have an internal timer. For the older 5600x processors, the implementation of the timing functions depends on what is provided by the boards. For example, the 56001 LSI board provides a hardware timer which is connected with the IRQB interrupt line.

This difference of timer implementation results in a few consequences for the microkernel services, as will be explained later in this section.

The timer driver is always installed and enabled in procedure `main()`by means of the KS_EnableISR() service. If the internal timer is used, the first parameter of this function is 30, because the timer interrupt vector address is 30. On the other hand, if the timer is implemented by means of an external counter and IRQB, then the first argument has to be 4. If the timer ISR is installed and enabled, the application programmer can read out the timer in high and in low resolution.

- In low resolution, the number of kernel ticks (default milli seconds) are returned. As this value is a 48 bit wraparound value, it is more interesting to calculate the difference between two values which were read out consecutively. However, to facilitate this, kernel service KS_Elapse() is written for this purpose. The services KS_Sleep() and KS_LowTimer() also use this low resolution service.
- The KS_HighTimer() service provides a high resolution timer by returning the number of timer counts. The units of this value depends of the clock speed of the counter. This service is not provided when the internal timer is used since this counter can not be read. In this case the high resolution timer value will be equal to the low resolution value.

See the Alphabetical List of Virtuoso kernel services earlier in this manual for a full description of these kernel services.

The timer device driver reserves event signal number 48 for its use.

27.10.2. The host interface device driver

The host interface driver is installed by calling procedure `init_server()`. In the examples that accompany the release of the Virtuoso kernel, the

installation of the host interface is done in procedure `main()`.

The host interface driver can be used on two levels. The lowest level needs only one kernel resource, HOSTRES, which secures the use of the low level host interface. This kernel resource must always be locked by the task that wants to make use of the host interface, and unlocked if this task has finished using the host interface. A list of low level procedures are at the disposal of the application programmer to do simple character-oriented I/O:

- server_putchar()
- server_pollkey()
- server_terminate()
- server_pollesc()

These procedures will do the locking and unlocking of HOSTRES, so that HOSTRES is transparent to the application programmer, using the low level host interface.

Also installed in the examples is an easy-to-use character-oriented I/O interface, based on two tasks, `conidrv` and `conodrv`, two queues, `CONIQ` and `CONOQ`, two resources, `HOSTRES` and `CONRES`, and a procedure called `printl()`. This higher level interface driver makes use of the low level interface procedures.

It is possible to use an even lower level of the host interface. Doing this, the application programmer can build other host interfaces that do more than character-oriented I/O. The minimum that is needed to make use of the lowest level host interface, is the kernel resource HOSTRES, to secure the use of the interface, and the procedure, named `call_server()`. Note, however, that HOSTRES is not needed if only one task makes use of the lowest level host interface and if the Task Level Debugger is not present. It is not the intention of this manual to lay out the internals of the host interface and the communication protocol between the host server program and the target board(s). Please contact ISI if more information is wanted on this topic.

For more details on the different levels of the host interface, see “Host server low level functions” and “Simple terminal oriented I/O” in the chapter of “Runtime libraries”.

The host interface device driver reserves event signal number 16 for its own use.

27.11. Task Level Timings

Following is a list of task level timings of some of the kernel services provided by Virtuoso. These timings are the result of timing measurement on a

DSP56002 board with a clock speed of 40 MHz and zero wait state program- and data-memory.

All timings are in microseconds. The C compiler used for the DSP56002 environment, is the G56KCC from Motorola.

```
Minimum Kernel call
Nop (1)                          10
Message transfer
Send/Receive with wait
Header only (2)                  71
16 bytes (2)                     75
128 bytes (2)                    80
1024 bytes (2)                   126
Queue operations
Enqueue 1 byte (1)               22
Dequeue 1 byte (1)               23
Enqueue 4 bytes (1)              26
Dequeue 4 bytes (1)              26
Enqueue/Dequeue (with wait) (2) 64
Semaphore operations
Signal (1)                       16
Signal/Wait (2)                  52
Signal/WaitTimeout (2)           70
Signal/WaitMany (2)(3)           74
Signal/WaitManyTimeout (2)(3)    92
Resources
Lock or Unlock (1)               16

Note :
One byte is one 24-bit word on the DSP56000.
(1): involves no context switch
(2): involves two context switches. Timing is round-
        trip time.
(3): Length of semaphore list is 2.
```

27.12. Application development hints.

The easiest way to start is to copy and modify one of the supplied examples. Some of the necessary files have fixed names, so each application should be put in a separate directory.

The following files will be needed for each application:

SYSDEF:

The VIRTUOSO system definition file. The SYSGEN utility will read this file and generate NODE1.C and NODE1.H.

MAIN1.C:

This contains some more configuration options, and the C ‘main’ function. Copy from one of the examples.

A number of configuration options are defined in this file, so they can be changed without requiring recompilation of all sources (this would be necessary if SYSDEF is modified).

CLCKFREQ : this should be defined to be the clock frequency of the hardware timer used to generate the TICKS time.

TIICKTIME : the TICK period in microseconds.

TIICKUNIT:the TICK period in CLCKFREQ units.

TICKFREQ:the TICK frequency in Hertz.

The number of available timers, command packets and multiple wait packets are also defined in this file. How much you need of each depends on your application, but the following guidelines may be followed:

Timers are used to implement time-outs (at most one per task), and can also be allocated by a task.

A command packet will be needed for each timer allocated by a task. Command packets used for calling a kernel service are created on the caller’s stack and should not be predefined.

A multiple wait packet will be needed for each semaphore in a KS_WaitM service call (for as long as it remains waiting).

MAIN1.C also defines some variables used by the console driver tasks, the clock system, the debugger task, and the graphics system. These are included automatically if you use the standard names for the required kernel objects.

CRT056l.ASM:

start-up assembly code

MAKEFILE:

The makefiles supplied in the EXAMPLES directory can easily be modified for your application. They also show how to organize things so you can optionally include the task level debugger. If you want to include the task level debugger, put the corresponding definitions out of comment:

```
VIRTLIB = $(LIBS)\virtosd.lib
DD = -dDEBUG
DDD = -P "DEBUG"
```

and put the other definition in comment:

```
# VIRTLIB = $(LIBS)\virtos.lib
```

whereby # is the comment sign. Then remake the application, just by doing:

```
MAKE <Enter>.
```

LINKFILE:

List of the object versions of all source files to be linked along.

YOUR SOURCE FILES :

In the examples, this is just test.c

28. Virtuoso on the Motorola 68030 systems

Virtuoso has been ported to a 68030 based system of CompControl hosted by OS/9. The operation is similar except that common memory regions are used to communicate between the different processor boards.

This document file contains additional information concerning the Virtuoso kernel.

28.1. Source files of the Virtuoso kernel

The source files of the Virtuoso kernel are compiled with the GNU C-compiler. Following source files have to be present in order to be able to make the relocatables:

.a files:

```
kernel.a mbint2.a starter2.a timer1.a
```

.c files:

```
charconv.c condrv.c dllist.c event.c hint.c iface.c
        mail.c mmap.c nodeinit.c printf.c printl.c
        queue.c res.c rtxcmain.c signal.c task.c
        ticks.c tldebug.c tlmonit.c tstdio.c
```

.h files:

```
dllist.h iface.h k_struct.h k_types.h siotags.h
        stdarg.h tlmonit.h tstate.h tstdio.h
```

makefiles:

```
Virtuoso.mak Virtuoso_d.mak
```

listfiles:

```
relocs.lst relsdbg.lst
```

Two makefiles are present. Each of them makes relocatables from the source files. “Virtuoso.mak” builds relocatables for a version without the Virtuoso task level debugger; “Virtuoso_d.mak” builds relocatables for a version with the debugger.

Each makefile will build a library from the relocatables. The list of relocatables is found in files “relocs.lst” (for the version without debugger) and “relsdbg.lst” (for the version with debugger). At the present time the libraries

cannot be used yet to build executable applications, since the order of the relocatables in the libraries is not yet optimal.

The makefiles presume following directory structure:

```
/h0
/
VIRTUOSO
/
GNUC
/
RELS ---------- RELS_NODBG
```

The source files and makefiles reside in subdirectory GNU C. Relocatables compiled without debugger option go into subdirectory RELS_NODBG; those compiled with debugger option go into subdirectory RELS.

28.2. Building an application executable

Two demos on the disk can serve as a guide to build an application. Each of the demos can be built with or without the task level debugger incorporated. The makefiles of these demos presume the following directory structure:

```
/h0
/
VIRTUOSO
/
TEST ---------TEST_SIO
RELS ---------RELS_NODBG -------RELS -------RELS_NODBG
```

Following source files must be present in the source directories TEST and TEST_SIO:

.c files:

```
demo.c driv1.c node1.c
```

.h files:

```
dllist.h iface.h k_struct.h k_types.h node1.h siotags.h
        tstdio.h
```

makefiles:

```
makefile nodbg.mak
```


link list files:

```
demo.lnf demo_nodbg.lnf
```

The include files “siotags.h” and “tstdio.h” are only needed in directory TEST_SIO.

Source file “demo.c” contains the source code for the application. “node1.c” and “node1.h” are files that are created by the system generation utility. At this moment however, the system generation utility is not yet ready, so that the present version of “node1.c” and “node1.h” are customized for the demo-applications. In “node1.c” one can find the definitions of all kernel objects used in the application: the task control blocks, queues, memory maps, mail-boxes, semaphores, resources and the names of the kernel objects for debugging purposes.

Since no OS-9 kernel is present on the target processor board, the application, with the Virtuoso kernel, will have to be able to run on its own. Therefore, the link option -r is used in the makefiles to build a raw binary file for a non-OS-9 target system.

When building an application, care must be taken during linking phase. The order in which the relocatables are linked into one application executable is not important, except for one module: “starter2.r”. This module must ALWAYS be the FIRST MODULE in the link list, because this module contains the startup code of the kernel and calls to initialization routines.

More on this later.

28.3. Configuration of the processor boards CC-112 of CompControl

In order to use the capabilities of the CC-112 processor boards for applications with the Virtuoso kernel, following configuration is needed:

On the main processor board running the server program and the OS-9 kernel:

1. watchdog jumper: disabled (connected)
2. EPROM jumpers : configured for 1 Mbit EPROM
3. SCSI jumper : connected
4. cache jumper : disabled (connected)
5. Boot-ROM initialization parameters:
6. Processor Module Name : CC112
7. Display NVDev contents on Reset/Poweron : y
8. Boot from floppy disk SCSI port : y
9. Boot from hard disk SCSI port : y
10. Boot from EPROM : n
11. Boot from VMEnet port : n
12. Host SCSI ID : 7
13. Hard disk controller SCSI ID : 0
14. Floppy disk controller SCSI ID : 1
15. Reset SCSIbus on Reset/PowerOn : y
16. Reset VMEbus on Reset/PowerOn : y
17. DPR Supervisory Access : y
18. DPR Non-privileged Access : y
19. DPR Extended Access : y
20. DPR Standard Access : y
21. VMEbus DPR Base Address : 08000000
22. Bus Request Level : 3
23. Bus Request/Release Mode :
24. Request Direct - Release On Request
25. Interrupt Handler VMEbus mask :
26. Level(s) 7,6,5,4,3,2,1 enabled
27. Interrupt Handler Local mask : Level(s) 7,1 enabled
28. VMEnet port : 08000080

On the target processor board running the application with Virtuoso:

1. watchdog jumper: disabled (connected)
2. EPROM jumpers : configured for 1 Mbit EPROM
3. SCSI jumper : connected
4. cache jumper : enabled (not connected)
5. Boot-ROM initialization parameters:
6. Processor Module Name : CC112
7. Display NVDev contents on Reset/Poweron : y
8. Boot from floppy disk SCSI port : n
9. Boot from hard disk SCSI port : n
10. Boot from EPROM : n
11. Boot from VMEnet port : n
12. Host SCSI ID : 7
13. Hard disk controller SCSI ID : 0
14. Floppy disk controller SCSI ID : 1
15. Reset SCSIbus on Reset/PowerOn : n
16. Reset VMEbus on Reset/PowerOn : n
17. DPR Supervisory Access : y
18. DPR Non-privileged Access : y
19. DPR Extended Access : y
20. DPR Standard Access : y
21. VMEbus DPR Base Address : 08400000
22. Bus Request Level : 2
23. Bus Request/Release Mode :
24. Request Direct - Release On Request
25. Interrupt Handler VMEbus mask : All levels disabled
26. Interrupt Handler Local mask : Level(s) 7,5,1 enabled
27. VMEnet port : 08000080

28.4. Additional information about the modules

`starter2.a`

As already mentioned earlier, this module must be the first one linked into an executable. This is because it contains the initialization that is normally done by the start routine for an application that is built to run on OS-9 (cfr. the module cstart.r).

Initialization of the data area, uninitialized as well as initialized, must now be done by the application itself (OS-9 isn't there anymore to do this job). This module will calculate and fill-in the global data pointer in register a6 of the

processor.

When an application is built to run on a non-OS-9 target system, the block of initialized data will follow immediately after the object code of the application:

```
---> increasing address --->
 |------------------| size of -------| size of -------| initialized |
 | application code | un-initialized | initialized----| data------- |
 |----------------- | data block---- | data block-----| block------ |
 |----------------- | (1 longword) --| (1 longword) --|------------ |
```

To find the block of initialized data, the starter routine will search for the first initialized data value. This is a value defined in this module (the hex. value FEEDC0DE). This is another reason why this module must be linked first in the executable. Once this value is found, its address is used as the global data pointer and filled in register a6.

The data block of the initialized data however, is not yet at the right place. The application expects that the global data pointer in register a6 will point to the first byte of the un-initialized data block. The ordering of the data blocks expected by the application is such, that the block of un-initialized data comes first, then the block of initialized data follows.

Therefore, the block of initialized data will be moved in a higher address space with the amount of bytes equal to the size of the un-initialized data block. Then the address space between the application code and the moved block of initialized data will serve as the block of un-initialized data. This address space is then filled with zeros.

The start routine is entered via an exception of the VME-mailbox memory. At the end of the start routine a ‘return-from-exception’, or RTE, causes the processor to return to its normal processing state. This mechanism will now be used to go to the C-routine ‘main()’, by overwriting the normal RTE- return address with the address of ‘main()’.

Further initializations performed by the start routine are:

1. - initialization of the VME-mailbox;
2. - correction of all the pointer values in the initialized data block

(see additional information concerning the module “nodeinit.c”).

`mbint2.a`

The VME-mailbox address space is only used as a trigger, to initiate a mailbox exception. The actual mailbox is fixed at address 0x08000100 and has a

maximum length of 64 32-bit words. In the mailbox exception routine the exception is translated to a kernel-signal (signal code 1), so that an application task can wait on it by using kernel service KS_Event() with parameter value 1. After being awaken from an event-signal 1, the application task can get the information with kernel service KS_Linkin(). The information that is read, is formatted in the Eonic Systems-proprietary protocol.

During a mailbox exception handle, the exception handler will lock the mailbox, to prevent accidentally overwriting by processes on other boards. This locking is done by writing a non-zero value at a fixed address 0x08000080. Other processor boards, or the server program on the main processor board, willing to use the mailbox, will therefore have to check the value at this address, prior to write information in the mailbox. Only if the value is zero, the mailbox is free to be written in.

`timer1.a`

The timer interrupt routine works on interrupts coming from the DUART on the processor board. The DUART is programmed as a timer to give regular interrupts each 1 millisecond.

The system level debugger, present in the EPROM's of the processor board, can be consulted by pushing the "abort" button on the front of the processor board. This however, can be done only once, because the system level debugger will stop the timer actions of the DUART. It is not possible to return from the system level debugger to the application, by entering the command 'g' of the debugger, as the application is completely paralyzed. The processor board must then be reset and the server program must reload and restart the application from the beginning.

The timer interrupt routine has a special feature built-in for visual inspection of its healthiness. Somewhere in the code of the timer interrupt routine a subroutine jump is done to a routine called "heartbeat". This routine changes a LED in the front of the processor board each half second. If this feature is not desired, the BSR-instruction to the 'heartbeat'- routine is to be put in comment.

`kernel.a`

This is the very core of the Virtuoso kernel. All interrupt routines (at this moment only the mailbox- and the timer-interrupt routine) and Virtuoso kernel service requests enter this module. The kernel service requests are translated to a processor TRAP, so that all kernel services are processed in the supervisory state of the processor, together with the interrupt processing.

The kernel initialization routine also contains initialization of the cache con-

trol register of the processor. This will only go for the 68030 processor type. If cache operation is not desired, two instructions within the kernel initialization routine must be put in comment:

```
move.l #$3919,d0
movec d0,CACR
```

The last instruction of the kernel initialization routine transfers the processor state to User Mode. This instruction must always be the last instruction before the RTS-instruction. This module also contains a copy-routine, called “fastcopy”. It copies blocks of memory as fast as possible, by copying 4 bytes at a time, if the block to be copied is more than 3 bytes long. This routine is used several times in the kernel to copy for instance a kernel message body. It is a C-callable routine, so that applications can easily use it too.

`charconv.c, printf.c`

Because several standard C-functions written for the OS-9 environment use OS-9 features, some functions used by the application are re-written:

`atoi()`
`printf()`

The idea behind it is to have the greatest possible independency of OS-9 libraries for an application written to run on Virtuoso on the target processor board.

`condrv.c`

Console input and output is queue-driven in Virtuoso. Two tasks, one for console input, the other for console output, run at a high priority to handle console I/O. Communication with these two tasks is done via a Input- resp. Output-queue. If console input/output is not needed in the application, the console tasks and queues can be removed from the system description files (node1.c and node1.h). In this case this module is not needed too.

`hint.c`

Kernel service KS_Linkin is serially oriented: reading out of the VME-mailbox message can be done in more than one step, by calling KS_Linkin several times. This is particularly interesting to process the Eonic Systems-proprietary protocol, because the length of an Eonic Systems-proprietary protocol message is given in the first word of the message itself. Therefore a readout of such a message has to be done in two steps.

After reading out the last piece of the message, kernel service KS_FreeMailbox() must be called in order to reset an internal read-pointer to an initial value. This service will also unlock the mailbox, so that a new message can be entered in the mailbox. So, KS-FreeMailbox() must not be for-

gotten.

`nodeinit.c`

This module contains part of the initialization sequence of the system. It will correct all pointers within the block of initialized data. Because the executable of an application is built from relocatables, initialized pointers will not have an absolute address value. A correction factor must be added to the pointers.

The correction factor is not equal for all initialized pointers: For pointers to data structures, the global data pointer must be added; for pointers to functions and to constant strings, the application start-address must be added. Both correction factors are calculated by the start routine in module “starter2.a” and are called “GlobDataPtr” and “StartOfCode”.

Constant strings however, can always be used, without the need to add a correction factor, as these strings are incorporated into the modules themselves. So for example:

```
sprintf (Buffer, “Hello World\n”);
```

can be written in the application without any problem.

`rtxcmain.c`

This module contains the main-function “main()”. Here, all initialization routines for the kernel structures are called. In “main()”, the DUART is also initialized and programmed to give tick-interrupts every 1 millisecond. Finally, all tasks of the EXE-group will be started.

As “main()” is part of the idle-task of the kernel, “main()” will never be exit-ed. Instead an endless loop is called that performs some statistics.

`tldebug.c, tlmonit.c`

These modules are only needed when the task level debugger of Virtuoso is wanted.

`tstdio.c`

This module contains all standard Input/Output services supported by Virtuoso. If no standard I/O is wanted, this module must not be linked with the application.

28.5. Server program for CompControl VME system board, running on OS-9

This document file contains additional information concerning the structure

and internals of the server program for Virtuoso.

28.5.1. Purpose of the server program

The server program resides on the first VME card in a VME node and has several tasks. It is written to run on the host operating system OS-9. In this way, one can view the server program as a bridge between Virtuoso, running on the other processor boards in the node, and the file system of OS-9, the text-output on the console screen and the keyboard-input.

Following list of features are supported by the server program:

1. loading an application, including the Virtuoso kernel, from disk and putting it on another processor board via the VME-bus. At startup time, the target processor board has no operating system running on it and has only a system level debugger in its EPROM's;
2. starting a target processor board that has just received its application software from the server program. This can be achieved by use of the interrupt vector of the VME mailbox that is present on a processor board of CompControl;
3. on request of an application running on another processor board, putting characters on the console screen;
4. catching key-presses from the keyboard and sending the key-code to the target processor board;
5. on request of an application running on another processor board, performing standard Input/Output functions, hereby eventually accessing the file system.

Communication between the server program and target processor boards will be done via the VME mailboxes on the processor boards by the use of an Eonic Systems-proprietary communication protocol.

28.5.2. Source files for the server program

Following source files have to be present in the source directory of the server program, in order to be able to make an executable:

`server.c` : C-source file of the server program

`hstdio.c` : C-source file of the standard Input/Output interface rtxh.h : include file with additional definitions

`siotags.h` : include file with the standard Input/Output tags

`makefile` : makefile for the server program server.lnf: link list file for the server program. The makefile presumes that the directory structure for the server program is as follows:


```
/h0/SERVER/CMDS
/h0/SERVER/SOURCE/RELS
```

The makefile also presumes that the mailbox-driver routines reside in directory /h0/MBDRIVER and that OS-9 libraries are to be found in directory /h0/LIB.

28.5.3. Use of the server program

The server program can be started just by typing “server”. If one starts the server this way, the name of the executable for the target processor board will be prompted for. Alternatively, the name of the executable can be given as a parameter, for example: server demo .

If the target processor board has to get a fresh executable, it always has to be reset first.

Some options can be used on the server program:

/c : the server presumes that the target processor board has already running an application on it, and will not load an executable on the target processor board. The server will signal the target board that it has started again and resumes operation. With this option, no executable file name is to be given.

/pn : before triggering the target board, the server will pause for n seconds.

/axxxxxxxx: specifies the absolute address at which the application will be loaded. xxxxxxxx is the address in hexadecimal format. If this parameter is not given, a default absolute address will be taken: 08003000.

The server can be interrupted or stopped by entering ^C or ^E. When one does this, the server will prompt “x to exit, c to continue”, so that the operator can still change his mind here. If after all the server is stopped, it can always be restarted to resume its duties with the option /c .

29. Virtuoso on the Motorola 96K DSP

29.1. Virtuoso versions on 96K

At this moment, two versions exist for the 96K. Both contain the microkernel. The first one is dedicated to single processor systems and does not contain the nanokernel level. This is the version described.

The second version contains the nanokernel and is suited for multiprocessor targets as well (with fully distributed semantics). The section on this version is under preparation.

29.2. DSP 96002 chip architecture

This section contains a brief description of the DSP9600x processor architecture. It is not intended to be a replacement of the Processor's User Manual, but as a quick lookup for the application programmer. Detailed information can be found in the "DSP96002 User's Manual" from Motorola.

The DSP96002 has a Harvard architecture (separated program- and data-addressing) with multiple internal buses. The interface to the outside world is done via two programmable ports. The concept of the processor family, to which this processor belongs, defines as its core a Data ALU, Address Generation Unit (AGU), Program Controller and associated Instruction Set. The On-Chip Program Memory, Data Memories and Peripherals support many numerically intensive applications, however they are not considered part of the core.

The DSP96002 supports IEEE 754 Single Precision (8 bit Exponent and 24 bit Mantissa) and Single Extended Precision (11 bit Exponent and 32 bit Mantissa) Floating-Point and 32 bit signed and unsigned fixed point arithmetic, coupled with two identical external memory expansion ports.

The major components of the DSP96002 are:

- Data Buses and Address Buses
- Data ALU
- Address Generation Unit (AGU)
- X Data Memory
- Y Data Memory
- Program Control and System Stack
- Program Memory
- Port A and Port B External Bus Interfaces

- Internal Bus Switch and Bit Manipulation Unit
- I/O Interfaces

FIGURE 16 does not give all details of the DSP96002 Block Diagram. See figure 3-1 in the DSP96002 User's Manual for full details

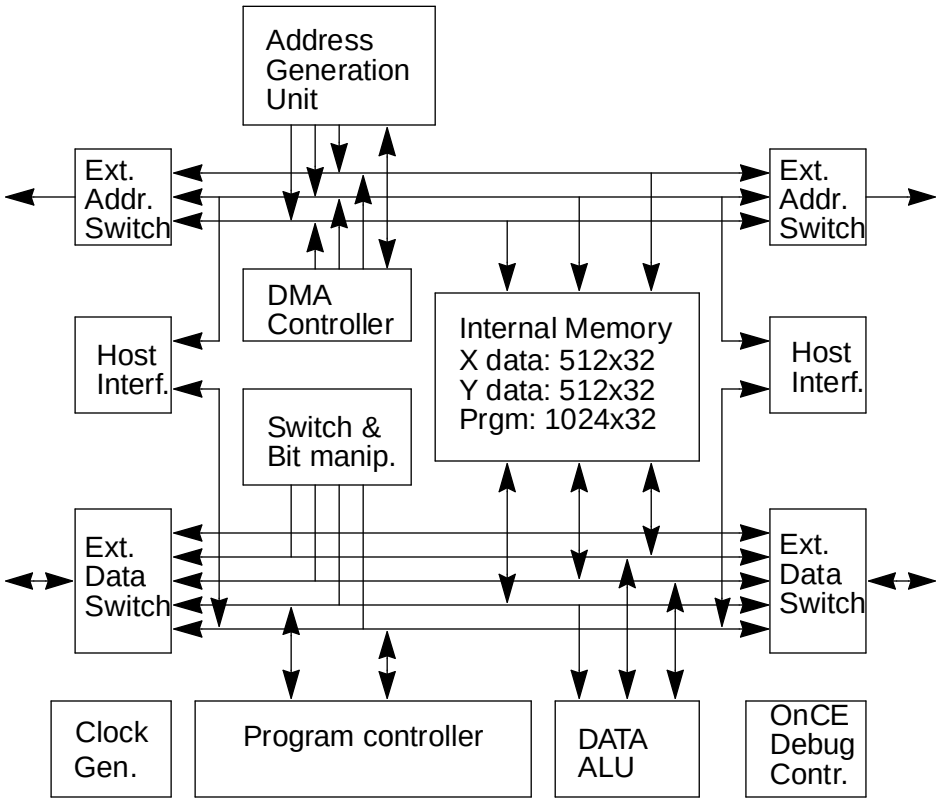


FIGURE 16 DS96002 simplified Block Diagram.

The major components of the DATA ALU unit are:

- Data ALU Register File
- Multiply Unit
- Adder Unit
- Logic Unit
- Format Converter
- Divide and Square Root Unit
- Controller and Arbitrator

The Adder Unit has also a Barrel Shifter. The major components of the

Address Generation Unit (AGU) are:

- Address Register Files
- Offset Register Files
- Modifier Register Files
- Temporary Address Registers
- Modulo Arithmetic Units
- Address Output Multiplexers

29.3. DSP 96002 software architecture

The programmer can view the DSP 96002 architecture as 3 execution units operating in parallel. The 3 execution units are the

- Data ALU
- Address Generation Unit
- Program Controller

The DSP 96002 instruction set has been designed to allow flexible control of these parallel processing resources. Many instructions allow the programmer to keep each unit busy, thus enhancing program execution speed. The programming model is shown in FIGURE 17 and FIGURE 18.

The ten Data ALU registers, D0-D9, are 96-bits wide and may be treated as 30 independent 32-bit registers or as ten 96-bit floating point registers. Each 96-bit register is divided into 3 sub-registers: high, middle and low.

The eight address registers, R0-R7, are 32-bits wide and may contain addresses or general purpose data. The 32-bit address in a selected address register is used in the calculation of the effective address of an operand.

The eight offset registers, N0-N7, are 32-bits wide and may contain offset values used to increment and decrement the corresponding address registers in address register update calculations or they may be used for general purpose storage.

The eight modifier registers, M0-M7, are 32-bits wide and may contain values which specify address arithmetic types used in address register update calculations (i.e., linear, reverse carry, and modulo) or they may be used for general purpose storage. When specifying modulo arithmetic, a modifier register will also specify the modulo value to be used.

The Status Register is a 32-bit register consisting of an 8-bit Mode register (MR), an 8-bit IEEE Exception register (IER), an 8-bit Exception register

(ER) and an 8-bit Condition Code register (CCR). Special attention is given here to the Interrupt Mask bits, I1 and I0, in the Mode register (MR). These bits reflect the current priority level of the processor and indicate the interrupt priority level (IPL) needed for an interrupt source to interrupt the processor. The current priority level of the processor may be changed under software control. The interrupt mask bits are set during processor reset.

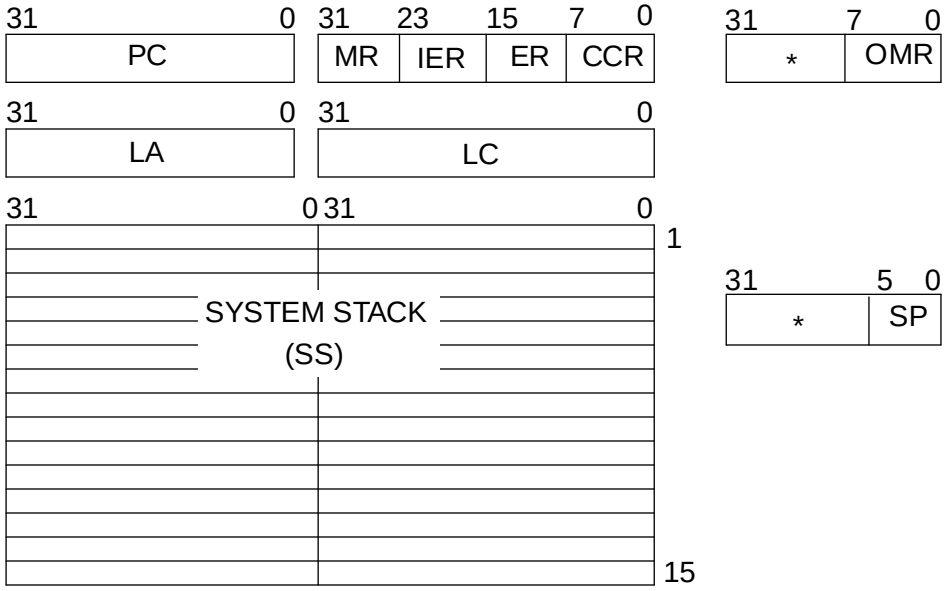


FIGURE 17 DSP96002 Programming Model - Program Controller

The Loop Counter (LC) is a 32-bit special purpose register used to specify the number of times a hardware program loop is to be repeated. The Loop Address Register (LA) is a 32-bit register that indicates the location of the last instruction word in a hardware program loop.

The System Stack is a separate internal memory which stores the PC and SR for subroutine calls and long interrupts. The stack will also store the LC and LA registers in addition to the PC and SR registers for program looping. The System Stack memory is 64 bits wide and 15 locations deep. The Stack Pointer (SP) is a 32-bit register that indicates the location of the top of the system stack and the status of the stack (underflow, empty, full, and overflow conditions).

FIGURE 18 DSP 96002 Programming Model - Data ALU and Address Generation Unit

DATA ALU

95		0	
D9.H	D9.M	D9.L	D9
D8.H	D8.M	D8.L	D8
D7.H	D7.M	D7.L	D7
D6.H	D6.M	D6.L	D6
D5.H	D5.M	D5.L	D5
D4.H	D4.M	D4.L	D4
D3.H	D3.M	D3.L	D3
D2.H	D2.M	D2.L	D2
D1.H	D1.M	D1.L	D1
D9.H	D0.M	D0.L	D0
31 0	31 0	31 0	

ADDRESS GENERATION UNIT

M7	N7	R7
M6	N6	R6
M5	N5	R5
M4	N4	R4
M3	N3	R3
M2	N2	R2
M1	N1	R1
M0	N0	R0
31 0	31 0	31 0

I1 I0	Exceptions permitted	Exceptions masked
0 0	IPL 0,1,2,3	None
0 1	IPL 1,2,3	IPL 0
1 0	IPL 2,3	IPL 0,1
1 1	IPL 3	IPL 0,1,2

FIGURE 19 Interrupt Mask bits I1 and I0

The SR format is shown hereunder.

Bit Nr. Code Meaning

31	LF	Loop Flag
30	*	Reserved
29	I1	Interrupt Mask
28	I0	Interrupt Mask
27	FZ	Flush to Zero
26	MP	Multiply
25	*	Reserved
24	*	Reserved
23	*	Reserved
22	R1	Rounding Mode
21	R0	Rounding Mode
20	SIOP	IEEE Invalid Operation
19	SOVF	IEEE Overflow
18	SUNF	IEEE Underflow
17	SDZ	IEEE Divide-by Zero
16	SINX	IEEE Inexact
15	UN CC	Unordered Condition
14	NAN	Not-A-Number
13	S NAN	Signaling NaN
12	OP ERR	Operand Error
11	OVF	Overflow
10	UNF	Underflow
9	DZ	Divide-by Zero
8	INX	Inexact
7	A	Accept
6	$\overline{R}$	Reject
5	LR	Local Reject
4	I	Infinity
3	N	Negative
2	Z	Zero
1	V	Overflow
0	C	Carry

Bit 0-7: CCR	Bit 8-15: ER
Bit 16-23: IER	Bit 24-31: MR

29.3.1. DSP 96002 addressing modes

The DSP96002 instruction set contains a full set of operand addressing modes. All address calculations are performed in the Address Generation Unit to minimize execution time and loop overhead.

Address register indirect modes require an offset and a modifier register for use in address calculations. These registers are implied by the address register specified in an effective address in the instruction word. Each offset register Nn and each modifier register Mn is assigned to an address register Rn having the same register number n.

The addressing modes are grouped into three categories:

- Register Direct
- Address Register Indirect
- PC Relative and Special

The Register Direct addressing modes are:

- Data or Control Register Direct
- Address Register Direct

The Address Register Indirect modes are:

- No Update : (Rn)
- Postincrement by 1 : (Rn) +
- Postdecrement by 1 : (Rn) -
- Postincrement by Offset Nn : (Rn) + Nn
- Postdecrement by Offset Nn : (Rn) - Nn
- Indexed by Offset Nn : (Rn + Nn)
- Predecrement by 1 : - (Rn)
- Long Displacement : (Rn+Label)

The PC Relative modes are:

- Long Displacement PC Relative
- Short Displacement PC Relative
- Address Register PC Relative

Special Addressing modes are:

- Immediate Data
- Immediate Short Data
- Absolute Address

- Absolute Short Address
- Short Jump Address
- I/O Short Address
- Implicit Reference

The DSP96002 Address Generation Unit supports linear, modulo and bit-reversed address arithmetic for all address register indirect modes. Address modifiers determine the type of arithmetic used to update addresses. Address modifiers allow the creation of data structures in memory for FIFOs (queues), delay lines, circular buffers, stacks and bit-reversed FFT buffers. Each address register Rn has its own modifier register Mn associated with it.

Following modifier classes are supported by the DSP96002:

- Linear Modifier
- Reverse Carry Modifier
- Modulo Modifier
- Multiple Wrap-Around Modulo Modifier

29.3.2. I/O memory and special registers

Internal I/O peripherals occupy the top 128 locations in X memory space. External I/O peripherals occupy the top 128 locations in Y memory space.

Register IPR in the X DATA memory space is used to program the Interrupt Priority for the DMA channels, the Host interfaces and the external interrupts IRQA, IRQB and IRQC. IPR is located on address X:$FFFFFFFF.

Register PSR is the Port Select Register, and is located on address X:$FFFFFFFC. Every memory space (X, Y and P) is divided into 8 equal portions of each 0.5 gigawords in length. PSR is used to map each of those portions onto the two ports A and B.

29.3.3. Expansion ports control

The DSP 96002 has two external expansion ports (Port A and Port B). Each port has a bus control register (BCRA and BCRB) where memory wait states may be specified. BCRA and BCRB are located on addresses X:$FFFFFFFE and X:$FFFFFFFD respectively.

29.3.4. Exceptions

Exceptions and interrupts are prioritized: a higher priority interrupt can suspend the execution of the interrupt service routine of a lower priority inter-

rupt. The priority level of an interrupt can be programmed to be at level 0, 1, 2 or to be disabled. Level 3 has the highest priority level and is unmaskable. This level is used by the RESET interrupt and by several exceptions, like Stack Error, Illegal Instruction and (F)TRAPcc. Each interrupt or exception starts at its own vector address. Following is a list of all interrupts and their starting address:

Starting Address	Interrupt Source
$FFFFFFFE	Hardware RESET
$00000000	Hardware RESET
$00000002	Stack Error
$00000004	Illegal Instruction
$00000006	(F)TRAPcc
$00000008	IRQA
$0000000A	IRQB
$0000000C	IRQC
$0000000E	Reserved
$00000010	DMA Channel 1
$00000012	DMA Channel 2
$00000014	Reserved:
$0000001A	Reserved
$0000001C	Host Interrupts:
$0000003E	Host Interrupts
$00000040	Reserved
$000000FE	Reserved
$00000100	User Interrupt Vector
$000001FE	User Interrupt Vector

During an interrupt instruction fetch, instruction words are fetched from the interrupt starting address and interrupt starting address +1 locations. While these two interrupt instructions are being fetched, the Program Counter is inhibited from being updated and so, the interrupt instructions are just inserted in the normal instruction stream.

Two types of interrupt routines may be used: fast and long. The fast routine consists of only the two automatically inserted interrupt instruction words. A jump to subroutine within a fast interrupt routine forms a long interrupt. A long interrupt routine is terminated with an RTI instruction to restore the PC and SR from the System Stack and return to normal program execution.

29.4. Relevant documentation

1. “DSP96002 IEEE Floating-Point Dual-Port Processor User’s Manual”, Motorola Inc., 1989
2. “Intertools Toolkit User’s Manual 96002 Release 1.1 for the PC”, Intermetrics Inc., 1991, Document Version 3.7, C Compiler / Assembler Version 1.1

29.5. C calling conventions and use of registers

This section contains following topics:

- Storage Allocation
- Segmentation Model
- Register Usage
- Subroutine Linkage
- Stack Layout

29.5.1. Storage Allocation

The basic C data types are implemented as follows:

`char`	32 bits, unsigned
`short`	32 bits, signed
`int`	32 bits, signed
`unsigned`	32 bits, unsigned
`long`	32 bits, signed
`float`	32 bits
`double`	64 bits in L memory
pointer (address)	32 bits (absolute address)

29.5.2. Segmentation model

User variables are allocated storage in one of the following places:

1. The run time stack.
2. The D3, D4, D5, D6, R1, R2, R3 and R5 registers.
3. The global data area (idata_y, ildata_L, udata_Y and uldata_L segments) referenced using absolute addressing.
4. Separate segments.

29.5.3. Register usage

The compiler reserves the following machine registers for particular uses:

Register	Use
D0	Integer and float return values, first parameter value
D1	Second parameter value
D3.l-D6.l	Integer and character register variables
D3.m-D6.m	
D3.h-D6.h	
D3-D6	Floating point register variables
R1-R3 and R5	Pointer register variables
R6	Frame pointer register (FP)
R7	Memory stack pointer (MSP)

29.5.4. Subroutine linkage

Preserved registers

Every procedure is responsible for preserving the following registers:D3, D4, D5, D6, R1, R2, R3, R5, R6 and R7. This rule also applies to any assembly language routines called from compiled code.

Register Return Values

The compiler expects function return values in registers under the following circumstances:

- Pointer values are returned in D0.
- All non-structure variables are returned in D0.

Parameter Passing

The first two parameters are passed in D0 and D1 and the remaining parameters are pushed on the stack unless the called routine accepts a variable number of arguments. In this case, the variable length portion of the parameter list is always pushed on the stack. Other parameters are pushed as one word.

Double parameters are pushed as one word in L memory. If the first parameter is a structure, all parameters get pushed on the stack. If the first parameter is not a structure but the second one is, the first parameter will be passed in D0 and the remaining parameters will be pushed.

Calling Sequence

The generated code for a procedure call has the following form:

1. Determine if the function return value will be returned in a register. If not, allocate space for a function return on the stack.
2. Load the first two arguments into registers D0 and D1.
3. Push the remaining arguments onto the stack. The arguments are pushed as words in reverse order, i.e., the last argument is pushed first.
4. If a function return temporary was allocated, push its address.
5. Call the function.
6. Pop off any stack arguments.
7. If a function return temporary was allocated, deallocate it after it is used.

Procedure Prologue and Epilogue

No prologue and epilogue code is generated for leaf routines that have no local variables. A leaf routine is one which makes no procedure calls.

For routines which have local variables packed in registers, a move instruction will be generated to save the register upon entry and restore it before exiting.

For all non-leaf routines, a move must be emitted to save SSH on the stack. When local variables exist that couldn't be packed in registers, code must be emitted to save and restore the frame pointer (R6) and the stack pointer (R7).

Example of a leaf procedure with all locals packed to registers:

- Prologue:

```
move <reg>,y:(R7)+      ; 1 move for each preserved
                        ; register word
```

- Epilogue:

```
move (R7)-              ; 2 moves for first preserved
                        ; register
move y:(R7),<reg>       ; note, however, that if there
                        ; are any other registers to be
                        ; restored, the decrement of R7
                        ; may be coalesced onto the
```



```
                                ; previous move
```

Example of a leaf procedure with stack labels:

- Prologue:

```
move #n,N7              ; n is the size of the new frame
move R6,x:(R7)          ; save FP in X side of stack
move R7,R6              ; set up new FP from SP
move (R7)+N7            ; update SP to restore space for
                                ; the new frame
move <reg>,y:(R7)+      ; 1 move for each preserved
                        ; register to be saved
```

- Epilogue:

```
move (R7)-              ; moves to restore preserved
move y:(R7),<reg>       ; registers
move R6,R7              ; restore old stack pointer
move x:(R6),R6          ; restore old frame pointer
```

Example of a non-leaf procedure with stack labels:

This is the same as leaf prologue and epilogue with 2 extra moves to save and restore the SSH.

Extra prologue instruction:

```
movec SSH,y:(R7)+N7
```

Extra epilogue instruction:

```
move y:(R6),SSH
```

29.5.5. Stack layout

Interfacing C and Assembly allows the user to utilize the benefits of both languages in programming tasks. When interfacing C and Assembly, it is essential that Assembly language, that is called from C, must match the compiler's conventions. Although not strictly required, it is recommended that all assembly language routines used the standard stack usage conventions, so that these routines can be called from within C code or vice versa.

Here is an example of a C program calling an assembly language routine:

```
extern int asmsub ();
main () {
```

```
                    int i, arg1, arg2, arg3;
                    i = asmsub (arg1, arg2, arg3);
                    ...
}
```

The assembly language routine is declared as an ordinary external C routine. According to the compiler's naming conventions, the C code will contain a call to a global symbol named `Fasmsub`. That is, the compiler prepends an upper case letter `F` to each C procedure name. Therefore the assembly language must define a global symbol of this name, that is:

```
XDEF Fasmsub
Fasmsub:
< entry code (prologue) >
< body of routine >
< exit code (epilogue) >
```

Before the call the C compiler processes the parameters according to C conventions. In particular, the first two arguments are placed in registers D0 and D1, and the remaining argument is placed on the stack.

After the entry code (or prologue) in the assembly routine is executed, the stack configuration is as shown below:

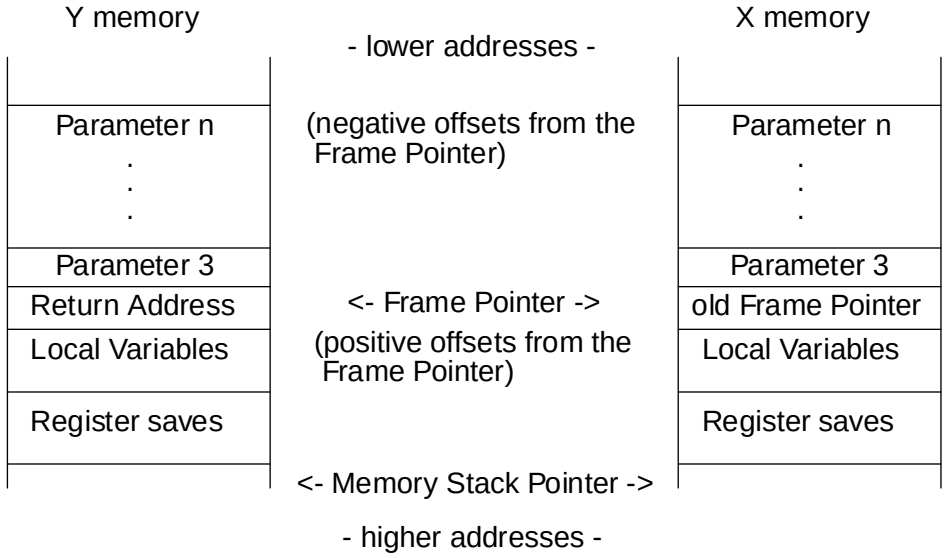


FIGURE 20 Stack usage

29.6. Interrupt Service Routines (ISR)

The two ISR levels that are normally supported by Virtuoso (ISR0 and ISR1) are not present in the version for the DSP96002 processor, since the DSP96002 supports multiple interrupt levels on its own. For more details on the different interrupts supported by the DSP96002, see section 29.3.4. of this manual, or, even more in detail, the DSP96002 User's Guide from Motorola.

29.6.1. ISR conventions

When using self-written ISRs in conjunction with the Virtuoso kernel, there are certain conventions to follow:

- Registers that must be preserved in an ISR.
- Interrupts that must be disabled at certain sections in the ISR code.

Saving or preserving registers

There are two sorts of ISRs that can be written:

1. ISRs that stand on their own and do not make use of the kernel to give signals
2. ISRs giving signals to the kernel

In both cases, the Stack Pointer (R7) must first be incremented, prior to saving any register at the start of an ISR. This is because a critical section exists in the epilogue code of a procedure:

```
move (R7)-
move y:(R7),<reg>
```

This instruction sequence restores a previously saved register. If an interrupt occurs in between these two instructions, the Stack Pointer points to the address where the value of the preserved register is written and the first move in the ISR to save a register will overwrite that value, if no prior increment of R7 is done. The same goes for the preservation of SSH.

Keeping this potential danger in mind, the following prologue code must be used for an ISR of the first class (no kernel interaction):

```
move (R7)+              ; prior increment of the SP
move <reg>,y:(R7)+      ; repeat this instruction for
                        ; every register that must be
                        ; saved
```

At the end of the ISR, right before the RTI instruction, following epilogue

code must be used:

```
move (R7)-
move y:(R7)-,<reg>     ; repeat this instruction for
                       ; every register that is saved
                       ; in the prologue code of the
                       ; ISR
```

Note, that this epilogue code is not critical anymore, provided the prologue of all ISRs start with an increment of the Stack Pointer (R7).

Which registers have to be preserved by an ISR depends on the class of ISR an on which registers are used in the ISR. If the ISR stands on its own (no signal is made to the kernel), only those registers must be preserved that are used by the ISR. In this case, the prologue and epilogue code just described are to be used. In the case the ISR gives a signal to the kernel, all registers that are used by the ISR must be preserved, except the registers D0.L, D1.L and R0: these registers must always be saved at the start of a signalling ISR, regardless if they are used by the ISR or not, because the kernel is relying on the fact that they are saved. The kernel expects in register D0.L the event signal number, which can range from 0 to 63 inclusive. So, for a signalling ISR, next conventions must be followed:

1. First increment the Stack Pointer R7.
2. Save registers D0.L, D1.L and R0 in this sequence.
3. Save all other registers used by the ISR.
4. Do whatever has to be done (body of ISR).
5. Restore all registers except R0, D1.L and D0.L. Note, however, that the last register restore may NOT contain a decrement of the Stack Pointer, because this decrement would be one too much.
6. Load register D0.L with the event signal number (value 0 - 63).
7. Jump to label `Fkernel_sign_entry`, to give the signal.

An example is given for each class of ISR.

Example 1: a non-signalling ISR uses registers D2.L, D4.L, R0 and R1.

```
move (R7)+             ; prior increment of R7
move D2.L,y:(R7)+      ; save D2.L
move D4.L,y:(R7)+      ; save D4.L
move R0,y:(R7)+        ; save R0
move R1,y:(R7)+        ; save R1
<body of ISR>
move (R7)-             ; post-decrements are faster
move y:(R7)-,R1        ; restore R1
```



```
move y:(R7)-,R0         ; restore R0
move y:(R7)-,D4.L       ; restore D4.L
move y:(R7)-,D2.L       ; restore D2.L
rti                     ; finito
```

Example 2: a signalling ISR using R4 and M4 as extra registers.

```
move (R7)+              ; prior increment of R7
move D0.L,y:(R7)+       ; save D0.L
move D1.L,y:(R7)+       ; save D1.L
move R0,y:(R7)+         ; save R0
move R4,y:(R7)+         ; save R4
move M4,y:(R7)+         ; save M4
<body of ISR>
move (R7)-              ; post-decrements are faster
move y:(R7)-,M4         ; restore M4
move y:(R7),R4          ; restore R4 - NO DECREMENT!
move #SIGNUM,D0.L       ; load D0.L with event signal
                        ; number
jmp Fkernel_sign_entry  ; signal the kernel
```

If a C procedure is called from an ISR, all registers that are not preserved across a procedure call (see paragraph 29.5.4. Subroutine Linkage for a list of preserved registers), have to be saved. However, for a signalling ISR, it is not advised to make a subroutine jump to a C function from within the ISR as this would introduce needless overhead of context saving. The kernel, when jumped to by label `Fkernel_sign_entry`, will perform a context save for all non-preserved registers. In this case, it is advised to make a task that waits on an event, with kernel service `KS_EventW(n)`, and that calls this C function after it is waked up by a signal to event number `n`.

Interrupt disabling times.

For the release of Virtuoso for the DSP96002 processor, it is not needed to disable interrupts in the code of an ISR. Because it is advised to disable interrupts in a period as short as possible, no interrupt disabling should be done in an ISR.

There are, however, certain sections in the kernel code where interrupts must be disabled, because these sections are critical and may not be interrupted. The longest period in the kernel code that interrupts are disabled is:

52 + 29*wp + 7*wy clock cycles

whereby wp and wy are the memory wait states for external program and Y-data memory respectively. Taking a clock frequency of 33 MHz and zero wait state external memory into account, the longest interrupt disabling time in the kernel is 52 clock cycles, or 1.58 microseconds.

29.7. Alphabetical list of ISR related services

```
Fkernel_sign_entry
/* for entering the kernel from within an ISR */
/* single processor version only */
KS_EventW()
/* for waiting on an interrupt at the task level */
KS_EnableISR()
/* for installing an ISR */
KS_DisableISR()
/* for removing an ISR */
```

29.8.1. Fkernel_sign_entry

• Synopsis . . . . . Label jumped to when entering the kernel from within an ISR

• Brief . . . . . .

This service gives a signal to the kernel with an event code numbered between 0 and 63 inclusive. A task can wait on the occurrence of such a signal by using kernel service `KS_EventW(n)`.

• Example . . . . . .

• See also . . . . . KS_EventW

• Special Notes . . . . The kernel signalling service assumes that certain conventions are followed by the ISR:

1. Stack Pointer R7 must be incremented at the very start of the ISR
2. Registers D0.L, D1.L and R0 have to be saved at the start of the ISR, after the prior increment of R7, with the sequence as indicated (see also previous paragraph)
3. Prior to jumping to the entry `Fkernel_sign_entry`, register D0.L must be loaded with the event number (between 0 and 63 inclusive)
4. A JMP instruction must be used to jump to the entry `Fkernel_sign_entry`, not a JSR instruction. The System Stack of the processor will be managed by the kernel, so that, when returning from interrupt, the correct program address will be loaded in the Program Counter

This kernel service is only callable from an ISR written in assembly when used with the single processor version (with no nanokernel).

29.9.2. KS_DisableISR

• BRIEF . . . . . Disables to ISR to be triggered by interrupt

• SYNOPSYS . . . . .

```
void KS_DisableISR (int IRQ);
```

• DESCRIPTION . . .

This C-callable service disables an ISR by writing an ILLEGAL instruction at the appropriate place in the interrupt vector table. Also, for the following interrupts, the corresponding bits in the IPR register of the processor will be changed accordingly:

- IRQA
- IRQB
- IRQC
- DMA Channel 1
- DMA Channel 2
- Host A Command
- Host B Command

Other interrupts can also be disabled by this service, but only in the sense that the JSR instruction at the corresponding place in the interrupt vector table will be overwritten by an ILLEGAL instruction.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
KS_DisableISR (9);
```

• See also . . . . . .

```
KS_EnableISR
```

• Special notes . . . .

29.10.3. KS_EnableISR

• BRIEF . . . . . Enables an ISR to be triggered by an interrupt

• SYNOPSYS . . . . .

```
void KS_EnableISR (int IRQ,
    void (*ISR)(void),
    int PrioLevel,
    int Mode);
```

• DESCRIPTION . . . This C-callable kernel service installs an ISR by writing a JSR instruction at the appropriate place in the interrupt vector table and setting the IPR register of the processor with the correct bit-values for the actual interrupt. This service may be used to install following interrupts, together with their priority level and interrupt mode (if appropriate):

- IRQA
- IRQB
- IRQC
- DMA Channel 1
- DMA Channel 2
- Host A Command
- Host B Command

Other interrupts can also be installed by this service, but for them the priority level and interrupt mode is not applicable and the arguments PrioLevel and Mode are not used.

• RETURN VALUE . . NONE

• EXAMPLE . . . . . .

```
extern void ISRDMACh1(void);
KS_EnableISR (9, ISRDMACh1, 2, 0);
```

29.11. The Nanokernel

No description, since the Nanokernel is not yet completed for the DSP96002 release of Virtuoso.

29.12.1. KS_EventW

- Brief . . . . . Waits on event associated with ISR
- Synopsis . . . . .

```
KS_EventW(int IRQ)
```

- Description . . . . . This C-callable kernel service can be used by a an application task to wait for a signal, given by an ISR. It forms a pair with kernel service Fkernel_sign_entry.
- Example . . . . .
- See also . . . . . Fkernel_sign_entry
- Special Notes . . . .

29.13. Developing ISR routines

When developing Interrupt Service Routines, the ISR conventions, described in paragraph 29.6.1., have to be followed.

The best place to install and enable an ISR, is in procedure `main()`, where predefined drivers, like the driver for the timer interrupt, are installed and enabled.

It is possible that additional initialization of registers and/or peripheral I/O has to be done. The best way to do this, is writing a C-callable procedure, that does the necessary additional initializations, and call this procedure after the call to `KS_EnableISR()`. An example of this method is the installation of the timer ISR in procedure `main()`:

```
#include “iface.h”
extern void timer0_irqh (void);
extern void timer0_init (void);
...
int main (void)
{
...
KS_EnableISR (4, timer0_irqh, IPLEVEL2, IPMNEDGE);
timer0_init();
...
}
```

29.14. The nanokernel on the 96002

Section in preparation.

29.15. Predefined drivers

Two devices drivers are already added to this release of the Virtuoso kernel. They are:

- the timer device driver
- the host interface device driver

The timer device driver is needed for time-out features of some kernel services and for kernel timer services. The host interface device driver is written to be able to communicate between the host server program and the DSP96002 target board.

29.15.1. The timer device driver

The timer driver is already installed and enabled in procedure `main()` of the examples that accompany the release of the Virtuoso kernel. If the timer ISR is installed and enabled, the application programmer can read out the timer in high and in low resolution.

In low resolution, the number of kernel ticks are returned. As this value is a 32 bit wraparound value, it is more interesting to calculate the difference between two values read out consecutively. However, to facilitate this, kernel service KS_Elapse() is written for this purpose.

In high resolution, the number of timer counts are returned. However, if no timer device is present on the DSP96002 application board, the timer interrupts will be generated by the host server program and there is no possibility of reading out the timer counter in high resolution. In this case the high resolution timer value will be equal to the low resolution value.

The two procedures to read out the timer value are:

- KS_LowTimer ()
- KS_HighTimer ()

See the Alphabetical List of Virtuoso kernel services earlier in this manual for a full description of these kernel services.

The timer device driver reserves event signal number 4 for its use.

29.15.2. The host interface device driver

The host interface driver is installed by calling procedure `init_server()`. In the examples that accompany the release of the Virtuoso kernel, the installation of the host interface is done in procedure `main()`.

The host interface driver can be used on two levels. The lowest level needs only one kernel resource, HOSTRES, which secures the use of the low level host interface. This kernel resource must always be locked by the task that wants to make use of the host interface, and unlocked if this task has finished using the host interface. A list of low level procedures are at the disposal of the application programmer to do simple character-oriented I/O:

- server_putch()
- server_pollkey()
- server_terminate()
- server_pollesc()

These procedures will do the locking and unlocking of HOSTRES, so that HOSTRES is transparent to the application programmer, using the low level host interface.

Also installed in the examples is an easy-to-use character-oriented I/O interface, based on two tasks, `conidrv` and `conodrv`, two queues, `CONIQ` and `CONOQ`, two resources, `HOSTRES` and `CONRES`, and a procedure called `printl()`. This higher level interface driver makes use of the low level interface procedures.

It is possible to use an even lower level of the host interface. Doing this, the application programmer can build other host interfaces that do more than character-oriented I/O. The minimum that is needed to make use of the lowest level host interface, is the kernel resource HOSTRES, to secure the use of the interface, and the procedure, named `call_server()`. Note, however, that HOSTRES is not needed if only one task makes use of the lowest level host interface and if the Task Level Debugger is not present. It is not the intention of this manual to lay out the internals of the host interface and the communication protocol between the host server program and the target board(s). Please contact Eonic Systems if more information is wanted on this topic.

For more details on the different levels of the host interface, see “Host server low level functions” and “Simple terminal oriented I/O” in the chapter of “Runtime libraries”.

The host interface device driver reserves event signal number 6 for its own use.

29.16. Task Level Timings

Following is a list of task level timings of some of the kernel services provided by Virtuoso. These timings are the result of timing measurement on a DSP96002 board with a clock speed of 33MHz and zero wait state program- and data-memory.

All timings are in microseconds. The C compiler used for the DSP96002 environment, is the Intertools C Compiler v.1.1 from Intermetrics.

```
Minimum Kernel call
Nop (1)                          9
Message transfer
Send/Receive with wait
Header only (2)                 59
16 bytes (2)                    62
```



```
128 bytes (2)                    68
1024 bytes (2)                   123
Queue operations
Enqueue 1 byte (1)               17
Dequeue 1 byte (1)               17
Enqueue 4 bytes (1)              18
Dequeue 4 bytes (1)              18
Enqueue/Dequeue (with wait) (2) 56
Semaphore operations
Signal (1)                       12
Signal/Wait (2)                  46
Signal/WaitTimeout (2)           56
Signal/WaitMany (2)              64
Signal/WaitManyTimeout (2)       73
Resources
Lock or Unlock (1)               12

Note :
One byte is one 32-bit word on the DSP96002.
(1): involves no context switch
(2): involves two context switches. Timing is round-
       trip time.
```

29.17. Application development hints.

The easiest way to start is to copy and modify one of the supplied examples. Some of the necessary files have fixed names, so each application should be put in a separate directory.

The following files will be needed for each application:

SYSDEF:

The VIRTUOSO system definition file. The SYSGEN utility will read this file and generate NODE1.C and NODE1.H.

MAIN1.C:

This contains some more configuration options, and the C ‘main’ function. Copy from one of the examples.

A number of configuration options are defined in this file, so they can be changed without requiring recompilation of all sources (this would be neces-

sary if SYSDEF is modified).

CLCKFREQ : this should be defined to be the clock frequency of the hardware timer used to generate the TICKS time.

TIICKTIME : the TICK period in microseconds.

TIICKUNIT:the TICK period in CLCKFREQ units.

TICKFREQ:the TICK frequency in Hertz.

The number of available timers, command packets and multiple wait packets are also defined in this file. How much you need of each depends on your application, but the following guidelines may be followed:

Timers are used to implement time-outs (at most one per task), and can also be allocated by a task.

A command packet will be needed for each timer allocated by a task. Command packets used for calling a kernel service are created on the caller's stack and should not be predefined.

A multiple wait packet will be needed for each semaphore in a KS_WaitM service call (for as long as it remains waiting).

MAIN1.C also defines some variables used by the console driver tasks, the clock system, the debugger task, and the graphics system. These are included automatically if you use the standard names for the required kernel objects.

PMAIN.ASM:

start-up assembly code

MAKEFILE:

The makefiles supplied in the EXAMPLES directory can easily be modified for your application. They also show how to organize things so you can optionally include the task level debugger. If you want to include the task level debugger, put the corresponding definitions out of comment:

```
VIRTLIB = $(LIBS)\virtosd.lib
DD = -dDEBUG
DDD = -P "DEBUG"
```

and put the other definition in comment:

```
# VIRTLIB = $(LIBS)\virtos.lib
```


whereby # is the comment sign. Then remake the application, just by doing:

```
MAKE <Enter>.
LINKFILE:
```

List of the object versions of all source files to be linked along.

```
LOCAT.CMD:
```

Locator command file. Change the memory reservations in this file, according to your memory needs. If the locator pass of the linker finds it has not enough Y: memory to place the data sections, it starts locating sections in the YR: memory space from address 7ff on (or right after the reservation for the ROM space - #400 to #7ff). This very annoying locator problem results in data sections being placed in nonexisting memory. The only known way to circumvent this problem is to change the other reservations so that every section can be placed in existing memory spaces.

YOUR SOURCE FILES :

In the examples, this is just test.c

After you have done make-ing your application, you should run the batch file MAKESYM.BAT, contained in the BIN subdirectory, from the directory where your application is built. This batch file makes the necessary symbol files .ADR and .SYM, starting from the .MAP file. The server program will not start your application if you omit this step.

30. Virtuoso on the Motorola 68HC11.

Chapter in preparation

31. Virtuoso on the Motorola 68HC16 microcontroller.

Chapter in preparation

32. Virtuoso on the Mips R3000 systems.

Chapter in preparation

33. Virtuoso on the INMOS T2xx, T4xx, T8xx.

33.1. Introduction

This section contains many parts taken from the manual of one of the first real-time kernels Eonic Systems developed. This provided for the first time preemptive scheduling on the INMOS transputer. For this reason, this chapter was more or less kept in its original form and contains the unusual part that explains why one needs a real-time kernel on the transputer. At that time, this was almost a religious issue. Many low level features you need to know on traditional processors are not needed on the transputer because they are solved in hardware, but the drawback is that one only has FIFO-based scheduling.

33.2. The transputer : an example component for distributed processing

When INMOS launched the transputer on the market, the goal of it was manyfold but primarily to provide cost-effective parallel processing. A second, but certainly as important aspect was the software methodology behind the transputer. People at INMOS were aware of the immense problems posed by large sequential programs as they are used for more and more complex systems, especially as for reasons of performance more than one processor is needed. Therefore, the software concept was developed first and the transputer afterwards.

The idea was simple : in order to manage complexity, one has to divide the whole up in smaller manageable components with well defined interfaces. They derived this idea from the so called CSP computing paradigm. CSP stands for Communicating Sequential Processes and describes programs as consisting of a number of processes that interact exclusively through communication. The point is that a lot of the problems programmers need to solve by writing a computer program reflect this paradigm very well. Examples range from a dataprocessing application (pipelined transformation of generated data) to finite element programs where each subspace can be calculated upon independently while interchanging intermediate values with the other subspaces. In process control, the parallel nature is even more apparent. For example, a data-sampler reads some data in and passes it on for further processing, while still another process processes the results for acting upon the controlled system.

The use of processes is not new for people who are familiar with operating systems or real-time applications. Their use is the only way to manage the

complexity. The main reason normal application programmers were not “allowed” to use this mechanism is that most processors have no provisions to support processes and data-communication. The only way out is a software based implementation, resulting in important performance losses. In addition, no languages were available that supported real parallel constructs in the language itself.

INMOS changed all that by designing in a first step a simple parallel language, called occam, and in a second step by building the transputer as a virtual occam machine. What this means in hardware terms is that the transputer is a classic 32bit processor (10 Mips at 20 MHz) with some novel features :

1. Instructions for process creation and termination;
2. Instructions for data-communication;
3. A microcoded FIFO-scheduler;
4. Two priority levels supported in microcode;

Four high speed communication links also enable to spread very easily processes over different processors for more performance.

In a nutshell, the transputer is a fast single chip computer with networking and parallel processing capabilities, requiring very simple interface logic to the outside world. Using the transputer is using computers as components.

33.3. Process control with transputers

The transputer as a virtual occam machine exhibits very well the CSP model on which its design was based. This CSP model is often found in the architecture of process control applications. As a result, the transputer has intrinsic features making it an attractive building block for process control applications, especially if these are distributed.

If no hard real-time constraints are to be satisfied, the overhead due to the use of communicating processes is relatively low as the transputer provides support in microcode for process scheduling and for communication. If however hard real-time constraints are to be satisfied, the designer is faced with a major difficulty, since the microcoded scheduler is a FIFO scheduler. Whereas this scheme was chosen for simplicity and performance in terms of throughput, a FIFO scheduler cannot guarantee the scheduling of a process within a known time interval. With the FIFO scheduler, a process can be worst case delayed for a time-interval equal to :

(2*N - 2)*TS + TSCH. TS being the timeslice (1 ms), N the number of processes in the low priority queue and TSCH being the time interval to the next descheduling point. [2],[3],[9]. The net result is that the transputer FIFO

scheduler enables fast throughput but results in unpredictable interrupt service response intervals. This problem has been identified by various authors. See [3],[4],[10]. In addition, even if we were able to start a task upon demand, we are still faced with the problem of timely execution. Indeed, once a critical (higher priority) task has started, we need to be sure that it will run until completion. In figure 2 this problem is made explicit by graphically representing two instances of the process queue.

Table 1 represents typical lower and upper limits of the interrupt service response interval (ISRI) in microseconds when 5 respectively 10 processes are in the FIFO queue. For TSCH, it was assumed that a normal distribution was valid with an upper bound of 100 microseconds. These figures are to be compared with the results obtained when using Virtuoso. A second problem concerns the timely termination of a interrupt service request. Even if the programmer were able to start up a critical process within a known (and sufficiently short) interval, the time slicing mechanism will intervene to allow the other processes to continue.

	Without Virtuoso (FIFO-queue)	**With Virtuoso (multiple priority queue)**
ISRI limits	**lower/upper**	**lower/ upper**
5 processes in queue	**1 / 8100**	**20 / 140**
10 processes in queue	**1 / 18100**	**20 / 140**
direct context switch : 6		

TABLE 1 Typical ISRI limit values (in microseconds)

33.4. A solution based on process priority

To be correct, the transputer has features that enable it to meet hard real-time constraints, provided one is willing to give up most of the benefits of the CSP model. The transputer knows two priority levels, each with its own process queue. Whereas a low priority process will be descheduled by a higher priority process at the next instruction, a high priority process cannot be interrupted. A feasible design methodology is then to let a high priority process accept the interrupt (“interrupt handling”) and forward the actual handling to a low priority process for “interrupt servicing”. However to achieve

timely execution, the resulting program will consist of a short high priority process with at most one low priority process. The unpredictable timing of the multi-tasking program is avoided by implementing a single-task program. Hence, the designer is back at sequential programming and processor cycles will be wasted. More complex programming techniques are possible, such as the artificially shortening of the hardware fixed timeslice using a cyclic timer. The net result is once again more complex programs. The worst side-effect however is that the problem is solved in an application dependent manner. In addition, this results in a high overhead. See [12], [13] for an example.

A general solution consists in using multiple priorities. This was demonstrated by various authors. See [4], [5], [6], [7]. The use of multiple priorities and the use of an appropriate scheduling algorithm can guarantee, under the right conditions, the timely execution of all processes. As such, the determination of the priority of each process is application dependent. On a single processor a rate monotonic scheduler where the priorities are inversely related to the periodicity of the processes, will result in a feasible scheduling order if the workload is under 70 %. An algorithm that better deals with aperiodic events is the earliest deadline. This can guarantee the scheduling of all processes even if the workload is close to 100 %. [7]. In general, the problem in known to be NP-complete, especially if all factors such as aperiodic events and common resources are taken into account. In these cases, a priority inheritance mechanism is advisable [5],[11]. Fortunately, the general case is often an exception so that most applications can be implemented using simpler algorithms.

33.5. Modifying the FIFO scheduler on the transputer

For the processor dependent low level routines, written in assembler, two major obstacles had to be overcome. The first was to find the algorithm that converted the FIFO scheduling mechanism into a preemptive scheduling mechanism. The second was how to implement the kernel in such a way that user tasks could safely interact with the special I/O hardware of the transputer. Both problems were solved in a satisfactory way using occam and C with a minor amount of assembler inserts. In addition we were happily surprised to find that the implementation on the transputer is one of the fastest available when compared with other processors running at the same speed.

In order to understand the importance of this key issue, it is worthwhile to have a closer look at the FIFO transputer scheduler mechanism.

On the transputer, the compiler will generate for each user defined process a “startp” instruction. In order to start, a process needs to be initialized with its Workspace Pointer (Wptr) and its Instruction pointer (Iptr). Whenever a new process is started (normally from within the first started process), a FIFO

process queue, implemented as a linked list, is build up. This is achieved by adding the new process at the back of the queue.

Once started, a process can be descheduled for the following reasons :

1. Because it has to wait on a communication with another process via an external link, via a soft channel in memory, or via the event pin
2. Because it has to wait on a timer event
3. Because of an interrupt from a high priority process which is ready to execute
4. Because its timeslice has expired and a descheduling point has been reached

In both queues all processes are executed in the order they were when placed in the queue. Parallel processing on a single transputer is then emulated through the time-slicing mechanism. Note that not all processes are present in the queue at all times. Only those processes that are ready to execute will be.

Hence to completely know the state of an executable process on the transputer, one needs to know the following elements [2]:

1. Wptr
2. Iptr
3. Areg, Breg, Creg
4. Ereg and the process status, if appropriate
5. FAreg, FBreg, FCreg
6. the place in the queue

Fortunately, these elements can be known by exploiting the fact that when a low priority process is interrupted by a high priority process, its elements are saved in the lower memory locations. Once these elements are known, it is possible to rearrange the linked list of the process queue such that, using a look-up table, the highest priority process is always in front of the queue. Care has to be taken that the linked list structure is not broken at any moment, especially as the transputer may start to manipulate the queue, independently of the currently executing instruction [8].

The final result is the capability of preemptive scheduling, meaning that the process with the highest priority, when in a runable state is always scheduled first and will be executing until it no longer can.

33.6. The Virtuoso implementation

In practice, Virtuoso is programmed as a set of high priority processes. Note that the kernel is the only process allowed to run at the transputer high prior-

ity and it is the only process allowed to talk directly to the external hardware, such as the links and the event pin. This means that all drivers and component processes of the microkernel are in fact Virtuoso nanokernel processes. The difference with the implementation on other classical processors is that the nanokernel is implemented by hardware and that only the channel primitives are available for communication and synchronization.

The application tasks run at transputer low priority, with a user defined (lower) priority.

The Virtuoso kernel will start up the tasks in order as defined by the user. Afterwards, the normal preemptive scheduling algorithm takes over.

Within each task the user can still use the normal FIFO scheduling mechanism as long as his actions are restricted to the boundaries of the task. So multiple subtasks can be run at the same priority with hardware support from the transputer.

When an application task needs to access the external hardware, the kernel will execute this as a service, using a small driver process. Memory mapped peripherals can be accessed directly by the application tasks.

Some of the communication links are not available to the application tasks. For example links that are purely used for building the network. The routing layer of the Virtuoso kernel uses these links to transfer data as well as remote service calls from one processor to another.

33.7. Requirements for embedded real-time systems

Various real-time kernels are on the market that provide preemptive scheduling services. In addition the application tasks need to be able to communicate in a synchronous and asynchronous way. They need to be able to synchronize and the kernel needs to be able to allocate memory, protect common resources and provide timer related services. In an era where processing technology is changing at a very rapid pace, a portable kernel, written in a high-level language, is essential. The result was RTXC/MP, the first true real-time kernel for the transputer. The second generation of this development was called virtuoso, because it is more than a real-time kernel, it is a programming system. It is the basis of a series of real-time dedicated products.

Embedded real-time systems have a number of characteristics that enable the designer to relax some of the conditions to be met :

1. Often the application will be static
2. Input and controlling actions are known in great detail, including their dura-

tion as well as their timing

3. Most of the code will be cyclic
4. The application is often stand-alone

As such, most of the program will be spent in a periodic loop, while aperiodic events (such as alarm conditions) are relatively rare. however when the latter occur, the designer needs to be sure that these will be handled immediately and without being interrupted.

Hence, such a control program will consist of a number of tasks, each with its own static priority. The designer will assign highest priority to the critical aperiodic tasks, while the periodic tasks receive an application dependent priority, eventually started by a master timer process. Note that by using a preemptive task scheduler, tasks that require no attention, will not consume any CPU time, resulting in an overall efficient use of processor resources.

These requirements are illustrated in Figure 2 that shows how a critical task will start immediately and stay running until termination when using a preemptive scheduler.

33.8. Small grain versus coarse grain parallelism

In essence there is no difference between a user task communicating with the outside world or with a process having a different priority. Both will result in a process being removed from the current process queue, hence breaking the linked list continuity as known by the kernel. Therefore, all tasks are implemented as low priority processes while these tasks can communicate among each other via the kernel running at high priority.

For this reason, Virtuoso contains some transputer specific calls, such as `KS_linkinW()` and `KS_LinkoutW()`. On the other hand, if within a user defined task of a given priority several processes that communicate with each other are started, the transputer hardware still takes care of the linked list continuity. The final result is that at each priority level, the normal interprocess communication facilities are available. This is primarily of interest as it simplifies the software design. The kernel itself is not resident but forms an integral part of the application program that is linked with the task code. As such a system designed with Virtuoso exhibits a two-layer structure. At the highest level, each task has an unique priority, while at the lowest level the basic hardware supported transputer mechanisms are still available. Figure 3 gives a schematic overview of Virtuoso running on a single transputer. Note that during program development, there is little difference for the programmer as to writing normal applications on the transputer. This is achieved by way of a system generation utility that generates automatically all kernel tables based on the information provided by the programmer.

33.9. Additional benefits from Virtuoso on the transputer

As was already indicated, the user can still use the normal transputer mechanisms within each priority level. We exercise complete control over the transputer from within the Virtuoso kernel. As such, we are able to provide programming services usually not available for the transputer programmer. First of all, we are able to monitor each scheduling event, so that the programmer knows exactly what has happened during the execution of his program. In addition, we are now in a position to manipulate the execution of the tasks at the instruction level and at the task level. This enables us to single step through the user tasks while providing a direct link with the original source code. At the time of writing, the latter development under the form of a single step debugger was not finished yet.

Nevertheless, the capability of preemptive scheduling provides a lot of real benefits for the transputer programmer. First of all, the FIFO queue latency is eliminated. This means that communication processes now can start as soon as a message arrives on the links. The net result is more throughput for short as well as for long messages. Secondly, the error flag is now task specific and the kernel will detect it. This simply means more security for the application. Thirdly, if a link is now disconnected or too noisy so that the normal synchronization protocol will result in a hanging communication, the kernel can detect this as well. While most transputer would then simply display all symptoms of classical deadlock, the kernel is able to continue to execute all other tasks and eventually reset and reinitiate the communication.

33.10. Device drivers with Virtuoso on the INMOS transputer

Although on the transputer, every communication acts as an interrupt by activating a process, the transputer itself does not know the traditional interrupt mechanism. Nevertheless, all interrupts can be handled, be it with a different hardware and software methodology. In general you will see that interfacing to peripheral devices can be simpler than on most traditional processors.

Only remember that the links, timers, and the event pin must be accessed through the kernel services if Virtuoso is running on the processor. The driver itself can be written at the Virtuoso microkernel task level using the kernel provided interfacing routines or the user can write a driver as a high priority process. Be aware that the driver is scheduled in FIFO order and must be kept short in order not to monopolize the CPU too long.

Below, we outline various schemes :

a. Use a separate transputer

(For example the 16 bit T225) to interface to the peripheral device and com-

municate with it through a link.

This solution is fast and simple. It assumes that you will run only one single I/O process on the separate processor and that you pass along the commands or data as soon as possible to minimize the communication delay. In addition as the T225 has a demultiplexed bus interfacing with it through memory is very simple. Some people have build a prototype A/D card with it in a single day and using only a minimum of components.

b. Interface the peripheral device through a link adaptor

A link adaptor converts between a serial link protocol and a parallel 8 bit system. This enables to read and to write to the peripheral device as a normal channel. Various transputer modules (such as DACs, ADCs and graphics TRAMs) that exploit this scheme are available from transputer board vendors.

c. Memory map the device

This means that you set up a dataregister at a certain memory location (preferably outside the normal memory space) and that you read from it or write to it as a normal memory location. In order to activate the handling process and to avoid expensive polling loops, you will probably need an ‘interrupt’, provided for example by the event pin. The equivalent of vectorising interrupts can be achieved by using an additional memory mapped register that holds some “status” or “address” related to the peripheral device. You can safely read or write to memory mapped registers from within your Virtuoso tasks.

33.11. Performance results

33.11.1. Single processor version. (v.3.0.)

The native transputer interrupt response can be fast (typically 1 to 3 microseconds) [9], but in reality this is the lower limit as to the actual scheduling of a specific process, one has to take account of the queue “latency”, often resulting in tens of milliseconds of reaction times. In the occam version and in the C version, we obtained basic switching times of 6 microseconds on a transputer running at 25 MHz and using the internal RAM. This means that the actual penalty is less than 5 microseconds, while we gain the certainty that the process will be scheduled within a known interval. The actual Virtuoso call takes longer since the kernel has to verify all pending messages, timers and the priority. Typically, a kernel service will take a minimum of 20 microseconds on a 30 MHz transputer using 3 cycle external memory and in absence of floating point code. When the FPU is in use, the times obtained

are increased with 16 microseconds (64 bit reals). Below, a summarizing table taken from the provided demo program, gives performance details for the version 3.0. of Virtuoso. These compare favorably with the figures obtained on other processors. Note that the times for the buffered communication calls (enqueue, dequeue) are essentially set-up times, which compare very favorably with other systems, available for the transputer. All times were measured in the kernel service originating task, which entails sometimes an extra context switch. Timings were made with the Logical Systems compiler and with the monitor disabled.

Minimum Virtuoso kernel call	**13 us**
Average allocate or deallocate	**17 us**
enqueue 1 byte	**22 us**
dequeue 1 byte	**29 us**
enqueue 4 bytes	**23 us**
dequeue 4 bytes	**23 us**
signal semaphore	**17 us**
average lock and unlock of a resource	**17 us**
enqueue 1 byte to waiting higher priority task	
+ dequeue in waiting task	**83 us**
send message to higher priority task	
+ wait for acknowledgment	**83 us**
send message to lower priority task	
+ wait for acknowledgment	**85 us**
signal/wait and wait/signal handshake	
between two tasks	**71 us**

TABLE 2 Table 2.: Performance Figures for Virtuoso v.3.0. calls (at 30 MHz)

33.11.2. The distributed version

The benchmarks indicate that the interrupt response times are only slightly increased by the fact that a service is located on a remote processor, even if the call is forwarded using an intermediate node. As such each intermediate node adds less than 50 microseconds to the interrupt response times. However different dynamic effects can have an importance, such as :

1. - communication latency
2. - time-varying workloads
3. - lack of buffers to store suspended calls

In the design of the distributed version, which mainly consisted in adding an embedded router to the kernel, special care was taking to minimize above mentioned effects. For this reason all kernel resources (tasks, semaphores,

queues, etc) are defined system wide. Hence in the whole system, each of these resources is uniquely identified. For example, there is only one task with the highest priority, although on each node in the network, the task with the highest local priority will be scheduled first by the kernel. The main advantage of this system lies in the fact that, by making the datastructures more dynamic, it has become possible to handle the messages in order of priority of the task that generated it. Hence the communication latency is reduced to a minimum. Below, the key performance figures for Virtuoso are summarized :

1. hop-delay : 50 us
2. task-to-task throughput rate : 1.5 MB/s (one link) to 4.6 MB/s (3 links)
3. message set-up time : < 25 us
4. synchronization delay for two tasks in a cluster of 1000 processors : around 600 us.

The main conclusion is that small clusters of transputers using Virtuoso, act as a single processing unit but with increased performance as compared to the same program running on a single transputer. In addition, it is now feasible to have several high priority tasks running at the same time. This is important if several high priority tasks are needed and when timely termination is a design criterium.

33.12. Using the compiler libraries with Virtuoso

The use of a real-time kernel brings you many advantages, compared to the standard operating environment. However, to support real-time operations, the user must refrain from using certain runtime functions. In general it can be stated that all operations which influence the scheduling should be performed through the provided kernel functions. If the user performs actions such as for example link I/O or timer_wait(), the proper working of the kernel can not be guaranteed. It is the users responsibility to assure that the rules below are followed.

33.13. Specific Parallel C routines not to be used by the tasks

The programmer must not use any 3L call that can activate the hardware scheduler from outside the task's boundary. In particular, this means any call that directly accesses the links, the event pin or the timer. Also, starting up threads at high priority is not permitted. Use the equivalent Virtuoso calls.

a. Timer functions which can cause a descheduling operation:

```
timer_delay();
timer_wait();
alt_wait();
```

```
alt_wait_vec();
```

The other functions timer_now() and timer_after() can be freely used.

Equivalent Virtuoso functions are provided.

b. Thread functions interfering with Virtuoso

```
thread_create();
     /* Uses par_malloc which is not protected under
     Virtuoso*/
thread_deschedule();
     /* Uses the timer functions */
```

Within each Virtuoso task it is however perfectly save to start up new threads using thread_start().

The other thread functions, thread_priority(), thread_restart(), thread_stop() are allowed.

Equivalent Virtuoso functions are provided.

c. Channel I/O to threads which are not generated from the same master task.

For external links use KS_LinkinW() and KS_LinkoutW(). For reading the event pin use KS_EventW(). For intertask communication you can use message calls.

d. Semaphore functions within threads which are not generated from the same master task

For intertask synchronization you can use the Virtuoso semaphore functions.

All standard file I/O routines like printf(), scanf(), etc. In some cases this will work, but the use of it is not advised. Use the supplied I/O library functions instead.

33.14. Specific routines of the INMOS C Toolset not to be used by the tasks.

a. Timer functions which can cause a descheduling operation.

```
ProcTimerAlt ()
ProcTimerAltList ()
ProcAfter ()
```



```
ProcWait ()
```

Equivalent Virtuoso functions are provided.

b. Thread functions interfering with Virtuoso.

```
ProcReschedule ()
ProcRunHigh ()
ProcPriPar ()
ProcAlloc ()
ProcAllocClean ()
```

Within each Virtuoso task it is however perfectly save to start up new threads although no tasks should be started up in the transputer high priority queue.

c. Channel I/O

to threads which are not generated from the same master task.

```
ChanAlloc ()
ChanOutTimeFail ()
ChanInTimeFail ()
```

For external links use KS_LinkinW() and KS_LinkoutW().

For reading the event pin use KS_EventW(). For intertask communication you can use message calls.

d. Semaphore functions within threads which are not generated from the same master task.

For intertask synchronization you can use the Virtuoso semaphore functions. Note the differences between the types.

```
SemAlloc ()
```

e. Memory allocation.

```
malloc ()
calloc ()
realloc ()
free ()
```

e. All standard file I/O routines like printf(), scanf(), etc. They don’t work as they require that the server program that comes with the compiler is used. Use the supplied I/O library functions.

33.15. Specific routines of the Logical Systems compiler not to be used by the tasks.

a. LSC Timer functions which can cause a descheduling operation.

```
ProcTimerAlt ()
ProcTimerAltList ()
ProcAfter ()
ProcWait ()
```

Equivalent Virtuoso functions are provided.

b. Thread functions interfering with Virtuoso.

```
ProcReschedule ()
ProcRunHigh ()
ProcPriPar ()
PForkHigh ()
ProcAlloc ()
ProcFree ()
SetHiPriQ ()
SetLoPriQ ()
ProcToHigh()
ProcToLow()
```

Within each Virtuoso task it is however perfectly save to start up new threads although no tasks should be started up in the transputer high priority queue.

c. LSC channel I/OChannel I/O to threads which are not generated from the same master task.

```
ChanOutTimeFail ()
ChanInTimeFail ()
ChanAlloc ()
ChannelFree ()
```

For external links use KS_LinkinW() and KS_LinkoutW(). For reading the event pin use KS_EventW(). For intertask communication you can use message calls.

```
malloc ()
calloc ()
realloc ()
free ()
```


d. LSC semaphore functions.

Semaphore functions within threads which are not generated from the same master task.

```
SemAlloc ()
SemFree ()
```

For intertask synchronization you can use the Virtuoso semaphore functions. Note the differences between the types.

e. All standard file I/O routines like printf(), scanf(), etc. Use the supplied I/O library functions instead.

34. Virtuoso on the INMOS T9000 transputer

In the mean time the T9000, the transputer of the second generation is coming to the market (announced end of 1992). It features a tenfold increase in performance in computing and in communication performance as compared to the T800 series. For process control and fault tolerant applications the T9000 has a lot of options. In addition there is now a protected mode and T9000 local mode as well, including traphandlers.

Besides these interesting features the T9000 has 16 Kb on chip RAM, still has two priority FIFO based queues and has an on chip Virtual Channel Processor. The latter enables, in conjunction with a link switch very fast communication between any processor of the network.

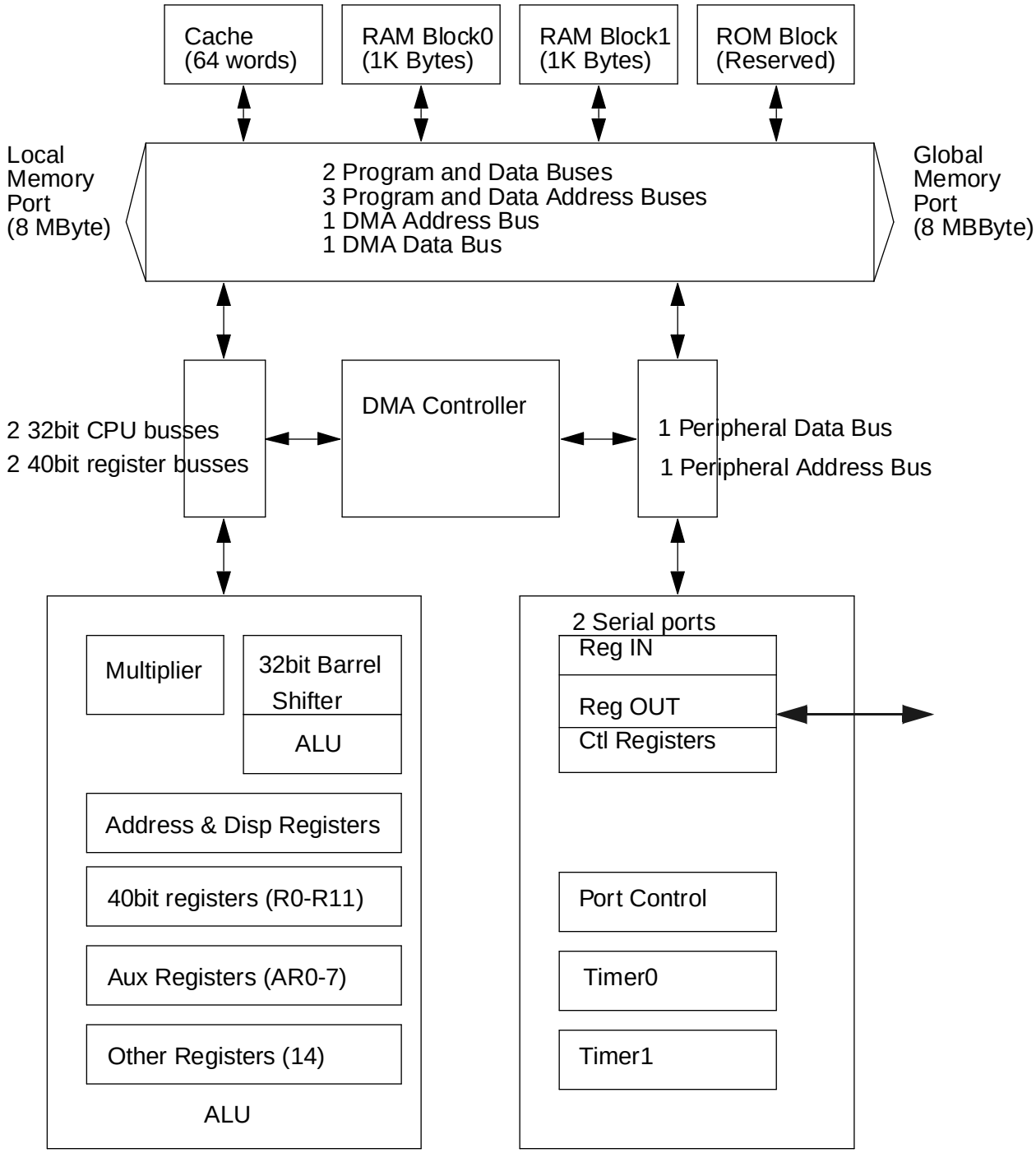
As you can see, a lot of the features of Virtuoso will even be faster on the T9000 while we will be able to implement the critical part of the kernel with more on chip support. The main thing is that the programmers interface to Virtuoso will be almost identical, delivering on the promise of portability across different technologies.

35. Virtuoso on the Texas Instruments TMS320C30 & C31

35.1. Virtuoso versions on TMS320C30/C31

At this moment, we support the microkernel level and one ISR level on the TMS320C30/C31. A port of the nanokernel as well as of the two level ISRs is undertaken and wil be released shortly.

35.2. TMS320C30 Chip Architecture



35.3. TMS320C30 Software Architecture

The TMS320C30 has a register based CPU architecture. The CPU contains following components :

- A Floating Point & Integer multiplier;
- ALU for floating point, integer and logical operators;
- 32bit barrel shifter
- Internal buses (CPU1/CPU2 and REG1/REG2)
- Auxiliary Register Arithmetic Units (ARAU)
- CPU register file

We will only list the CPU registers as these are the most important ones for the applications programmer.

Asm Symbol	Assigned Function Name
R0 .. R7	Extended Precision Registers
AR0 .. AR7	Auxiliary Registers
DP	Data Page Register
IR0	Index Register 0
IR1	Index Register 1
BK	Block Size Register
SP	System Stack Pointer
ST	Status Register
DIE	DMA Coprocessor Interrup Enable Register
IIE	Internal Interrupt Enable Register
IIF	IIOF Flag Register
RS	Repeat Address Register
RE	Repeat End Address Register
RC	Repeat Counter
PC	Program Counter
IVTP	Interrupt Vector Table Pointerr
TVTP	Trap Vector Table Pointer

35.3.1. Addressing Modes

The TMS320C40 addressing modes can be partioned into 4 groups. In each group two or more addressing types are provided.

1. General Addressing modes
   - Register Operand is register
   - Immediate Operand is 16bit immediate value
   - Direct Operand is 24bit address
   - Indirect Addres is in 24bit Auxilairy Register
2. Three Operand Addressing Mode
   - Register (see above)r
   - Indirect (see above)
   - Immediate (see above)
3. Parallel Addressing Modes
   - Register Operand is extended precision register
   - Indirect (see above)
4. Long-immediate adressing mode
   - Operand is 24-bit immediate value
5. Branch Adddressing Modes
   - Register (see above)
   - PC-relative A signed 16bit displacement is added to the PC

35.4. Relevant documentation

It is highly recommended to read carefully the following documentation available from Texas Instruments. In this manual we only outline the main

TMS320C3x User's Guide (Texas Instruments, 1991 Edition)

TMS320 Floating Point DSP Assembly Language Tools (Texas Instruments, 1991)

TMS320 Floating Point DSP Optimizing C Compiler (Texas Instruments, 1991

35.5. Application development hints

The easiest way to start is to copy and modify one of the supplied examples. Some of the necessary files have fixed names, so each application should be put in a separate directory.

The following files will be needed for each application :

SYSDEF : the Virtuoso system definition file. The SYSGEN utility will read this file and generate NODE1.C and NODE1.H.

MAIN1.C : this contains some more configuration options, and the ‘main’ function.

A number of configuration options are defined in this file, so they can be changed without requiring recompilation of all sources (this would be necessary if SYSDEF is modified).

CLCKFREQ : this should be defined to be the clock frequency of the hardware timer used to generate the TICKS time. For a C30, this is 1/4 of the CPU clock frequency.

TICKTIME : the TICK period in microseconds.

TICKUNIT : the TICK period in CLCKFREQ units.

TICKFREQ : the TICK frequency.

The number of available timers, command packets and multiple wait packets are also defined in this file. How much you need of each depends on your application, but the following guidelines may be followed :

- Timers are used to implement timeouts (at most one per task), and can also be allocated by a task.

- A command packet will be needed for each timer allocated by a task.

Command packets used for calling a kernel service are allocated on the caller’s stack and should not be predefined.

- A multiple wait packet will be needed for each semaphore in a KS_WaitMservice call (for as long as it remains waiting).

MAIN1.C also defines some variables used by the console driver tasks, the debugger task, and the graphics system. These are included automatically if you use the standard names for the required kernel objects. Finally, the main function is the obvious place to install ISR’s. This should be done AFTER the kernel_init () call. Note that at this point interrupts are already enabled, so you should disable them while initializing the hardware used by your ISR’s (use TRAP 0 in assembly, or the DISABLE macro in C code). KS_EnableISR0 disables interrupts while it is executing, and enables them (unconditionally) on return.

BOOTLSI.OBJ : can be copied from the LIB directory. It replaces the boot-module in the compiler library. The processor memory bus configuration is

also defined in this file.

MAKEFILE : the makefiles supplied in the EXAMPLES directory can easily be modified for your application. They also show how to organize things so you can optionally include the task level debugger.

*.CMD FILES : copy from the examples and modify as necessary. In the examples, LSIC30.CMD defines the memory layout and section placement, and TEST.CMD contains linker commands and object filenames.

YOUR SOURCE FILES : in the examples, this is just test.c

The pc host program

Type LSIHOST <ENTER> to obtain a help screen.

The default port address can be changed if you recompile the host program. The server requires Borland BGI screen drivers if the graphics output functions are used. The EGA-VGA driver is required to run some of the example programs, and is supplied on this disk(s). A path to the directory holding the BGI files should be defined as an environment variable

'BGIPATH' (e.g. SET BGIPATH=C:\VIRTUOSO\C30).

5. Recompiling the PC programs

All PC executables have been developed using BORLAND C and TASM, but they should be quite portable.

6. Using the serial links as a host interface

It is possible to use the TI EVM board to connect a stand-alone C30 board to the PC. A simple program running on the EVM passes the host protocol packets from the target board to the EVMHOST running on the PC and back. Serial port 0 on the target board should be connected to serial port 1 on the EVM. The clock for the serial link is generated on the target board. The cable should be wired as shown below :

```
EVM TARGET
----------
 R Data ---<-------------- X Data
 R Sync ---<-------------- X Sync
 X Data -------->--------- R Data
 X Sync -------->--------- R Sync
 R Clock |----<----------| X CLock (out)
 X Clock | not connected | R Clock (in)
```


To use this host interface, link with SERH.LIB instead of LSIH.LIB (modify the TEST.CMD file and re-MAKE). The interface program on the EVM should be running before the application is booted on the target board. It is not necessary to reload it each time, but a reset will be required.

You can test the interface using the LSI board and only one PC as follows :

1. remake one of the examples, as described above
2. Type EVML to boot the EVM

Type ERUN TEST to reset the EVM, boot the LSI board and restart the EVM server.

35.6. Interrupt handlers and device drivers for Virtuoso on the TMS320C3x.

35.6.1. Interrupt handling in Virtuoso.

This section describes how interrupts are handled in Virtuoso. It will be of interest to users who need to write their own interrupt handlers and device drivers.

Interrupt handling is easily the most processor specific part of any piece of software. Some of today's processors are able to accept interrupts at frequencies well into the megahertz range. It is virtually impossible for a software kernel to support task swapping at anything approaching this speed. Therefore, interrupt handlers cannot always be implemented as a task.

The interrupt processing mechanism implemented in Virtuoso is designed to separate interrupt handling into two distinct parts.

The first part is done entirely in the background, without any support from the kernel. This would typically be the code required to service the hardware that generated the interrupt, or to buffer data generated by high frequency

interrupts. The second part is the one that is not related to the hardware, but to the logic of an application. As an example, special action may be required when a received message is complete. Virtuoso encourages the application writer to do this part of interrupt processing at the task level, by providing a fast and easy to use mechanism to transfer control from an ISR to the kernel, and indirectly, to a task. If an interrupt occurs at a frequency that can be handled at the task level, then only a very simple ISR (an example is supplied) is necessary to transfer control to a task.

This approach has a number of important advantages :

- When the thread of execution of an ISR enters the kernel, it is no longer an ISR. The interrupt service routine has effectively ended. This is so because the kernel entry point used by ISR's is fully reentrant, and processing inside the kernel is performed with interrupts enabled. In this way, the kernel provides automatic reentrancy for long-winded ISR's. This could be difficult to achieve otherwise.

- In most cases the hardware related processing can be done using very few instructions, and only a small subset of the processor registers. Therefore, this type of interrupt handler can be very short, and latency will be minimal.

- Separating the 'hardware' and the 'system' part makes is possible to optimize each of them individually. Inside the ISR, full use can be made of any processor specific features. When control moves to the task level, the full power of the kernel is at the disposal of the user.

In order to understand how control is passed from an ISR to the kernel, we should have a look at how the kernel operates in the first place.

The kernel maintains a FIFO buffer of 'things to do'. It reads and handles entries in this FIFO the one after the other. When the FIFO becomes empty, the kernel determines which task should run next, and releases the CPU.

Each 'thing to do' is represented by a single word. When this word is a small integer, (typically 64) it is interpreted as an event number. Some events (i.e. from the TICKS timer, or from the routing layer in an MP system) are handled internally by the kernel. The others can be waited for by a task, using KS_EventW. If the FIFO entry is not a small integer, it is assumed to be a pointer to a command packet. There are three routines that will put a 'thing to do' into the kernel FIFO. These routines are in fact the entry points of the kernel. The first two can be called as C functions, while the third is just a label to jump to.

```
void kernel_task_entry (void *);
```

This is called when a task requests a kernel service. A command packet is assembled and the entry point is called with a pointer to the command packet as the parameter (see IFACE.C for examples). The pointer is put into the FIFO and control is given to the kernel. This call 'returns' when the calling task resumes execution - the task may have been swapped out in between.

```
void kernel_call_entry (void *);
```

This is used to put a new element in the kernel FIFO from within the kernel itself. This is the opposite of the previous one - it puts the pointer into the FIFO, assumes the kernel is already running, and returns immediately. It is

used by the timer system and the routing layer to put timed or remote command packets into the FIFO.

```
kernel_sign_entry;
```

This is the entry point for an ISR that wants to hand over an interrupt to the kernel. This is done by branching to this entry point from within the ISR. When the jump is performed, it is assumed that the ISR return address and a small (implementation defined) subset of CPU registers are still pushed on the stack of the interrupted process. One of the saved registers contains an entry to be put into the kernel FIFO. Since an interrupt may occur at any time, there are two cases to consider:

- A task was interrupted. This means the ISR has been using the task's stack, and has already partially saved its context. In this case the rest of the task's context is saved and control is given to the kernel. Note that the registers pushed onto the task's stack by the ISR have become part of the task's saved context. This means that for all practical purposes, the ISR has ended. Allowing the ISR to continue when the task is rescheduled would be useless anyway.

- The kernel itself was interrupted, and the ISR has been using the kernel's stack. In the case, the kernel will jump to an ISR exit sequence to restore its registers, and continue. Again, the ISR has ended.

For the TI C30 and C40 implementations of Virtuoso the following conditions should be satisfied when an ISR branches to kernel_sign_entry :

- interrupts are disabled

- the return address and the saved ST and R0 registers are still on the stack.

The following sequence would perform a normal interrupt return :

```
POPF R0
POP R0
POP ST
RETI
with R0 = event number.
```

As an example, this is the ISR used to maintain the TICKS time. The interrupt is generated by a hardware timer. The ISR updates a local variable 'hitime', and then passes an event to the kernel. In the final version, a delayed branch

should be used of course.

```
= _timer1_irqh
push ST                      ; save minimal register set
push R0
pushf R0
ldi @timer1divd, R0              ; hitime += timer1divd
addi @hitime, R0
sti R0, @hitime
ldi TIMER1SIG, R0           ; signal the TIMER1SIG event
b kernel_sign_entry
```

This is a very simple ISR - it does a minimum of local processing and always hands over the interrupt the kernel. A more complex example can be found at the end of the next section.

35.6.2. Parts of a device driver.

To implement a device driver in the Virtuoso environment, in general three pieces of code will be needed :

(1). An application interface at the task level,

These are the procedures called directly by the application code. In most cases, these functions should :

1. Protect the device by locking a resource.
2. Issue a number of device commands,
3. For each command, deschedule the caller until the command has been executed. This is done by calling KS_EventW ().
4. Unlock the device for other users.

These routines are not needed if the device is used by the kernel only, e.g. for internal routing.

(2). One or more low-level device control procedures,

These are used to send commands and parameters to the low-level interrupt handlers in a structured way.

(3). An interrupt handler for each interrupt used by the device.

These are the routines that are called via the hardware interrupt vector table when an enabled interrupt is raised. There are discussed in the previous chapter.

As an example, we will describe a simple device driver used to transmit data

packets over the C30 serial link. Each packet is an array of 32 bit words. The lower eight bits of the first word indicate the packet length. We assume that the hardware has already been initialized.

```
(1). Task level interface
void sendpacket (int *P)
{
KS_LockW (SERIAL1);     /* protect the device from other
        users */
serial1_send (P);           /* start the transmission */
KS_EventW (6); /* deschedule until packet transmitted */
KS_Unlock (SERIAL1); /* release device to other users */
}
```

(2). Device control function

This could have been written in C as well, but putting it in the same .ASM file as the ISR really simplifies the interface and minimizes the number of global variables.

```
SP1BASE .set 000808050h ; hardwarebase address
GLBCTL .set 0 ; register offsets
TXPCTL.set 2
RXPCTL.set 3
TIMCTL.set 4
TIMCNT.set 5
TIMPER.set 6
TXDATA.set 8
RXDATA.set 12
TX1INT.set 6 ; kernel signal codes (IE bitnumber)
RX1INT.set 7
.data
port1base .word SP1BASE ; hardware base address
tx_data.word0 ; pointer to tx data buffer
tx_count.word 0 tx word counter

; void serial1_send (void *P)
;
; Set up tx_data and tx_count and send the first word.
; The ISR will be called to handle the rest

.text
```

```
_serial1_send:
ldiSP, AR2 ; get first argument on stack
ldi*-AR2, AR0 ; pointer to packet
ldi*AR0++, AR1 ; get first word, point to next
sti AR0, @tx_data ;save pointer for isr
ldi AR1, R0 ; extract length
and 0FFh, R0
subi 1, R0 ; number of word remaining
sti R0, @tx_count ; save length for isr
ldi @port1base, AR0 ;transmit first word
sti AR1, *+AR0(TXDATA)
rets
```

(3). Interrupt handler

```
     An interrupt will be generated for each transmitted
     word. Send a signal to the kernel if all words have
     been transmitted, otherwise send next word and
     return from interrupt.
.text
_ser1tx_irqh:

push ST ; save minimal register set
push R0
pushf R0

ldi @tx_count, R0 ; get word count, test if zero
bz tx1_1 ; last word transmitted, signalkernel
subi 1, R0 ; decrement word count
sti R0, @tx_count ; and write it back

push AR0 ; save some more registers
push AR1

ldi @tx_data, AR0 ; pointer to next word
ldi *AR0++, AR1 ; read word, increment pointer
sti AR0, @tx_data ; store updated pointer
ldi @port1base, AR0 ; address of tx port
sti AR1, *+AR0(TXDATA) ; transmit word

pop AR1 ; not done yet, just end ISR
```



```
pop AR0 ; continue whatever process
popf R0 ; was interrupted
pop R0
pop ST
reti

tx1_1: bd_kernel_sign_entry
                              ; send signal TX1INT tokernel
ldi TX1INT, R0 ; R0, ST and return address
nop ; are still pushed on the stack

nop ; the kernel will clean this up
```


36. Virtuoso on the Texas Instruments TMS320C40

36.1. Brief description of the processor architecture

This section contains a brief description of the TMS320C40 processor architecture. It is not intended to be a replacement of the Processor's User Manual, but as a quick look-up for the application programmer. Detailed information can be found in the "TMS320C40 User's Guide" from Texas Instruments.

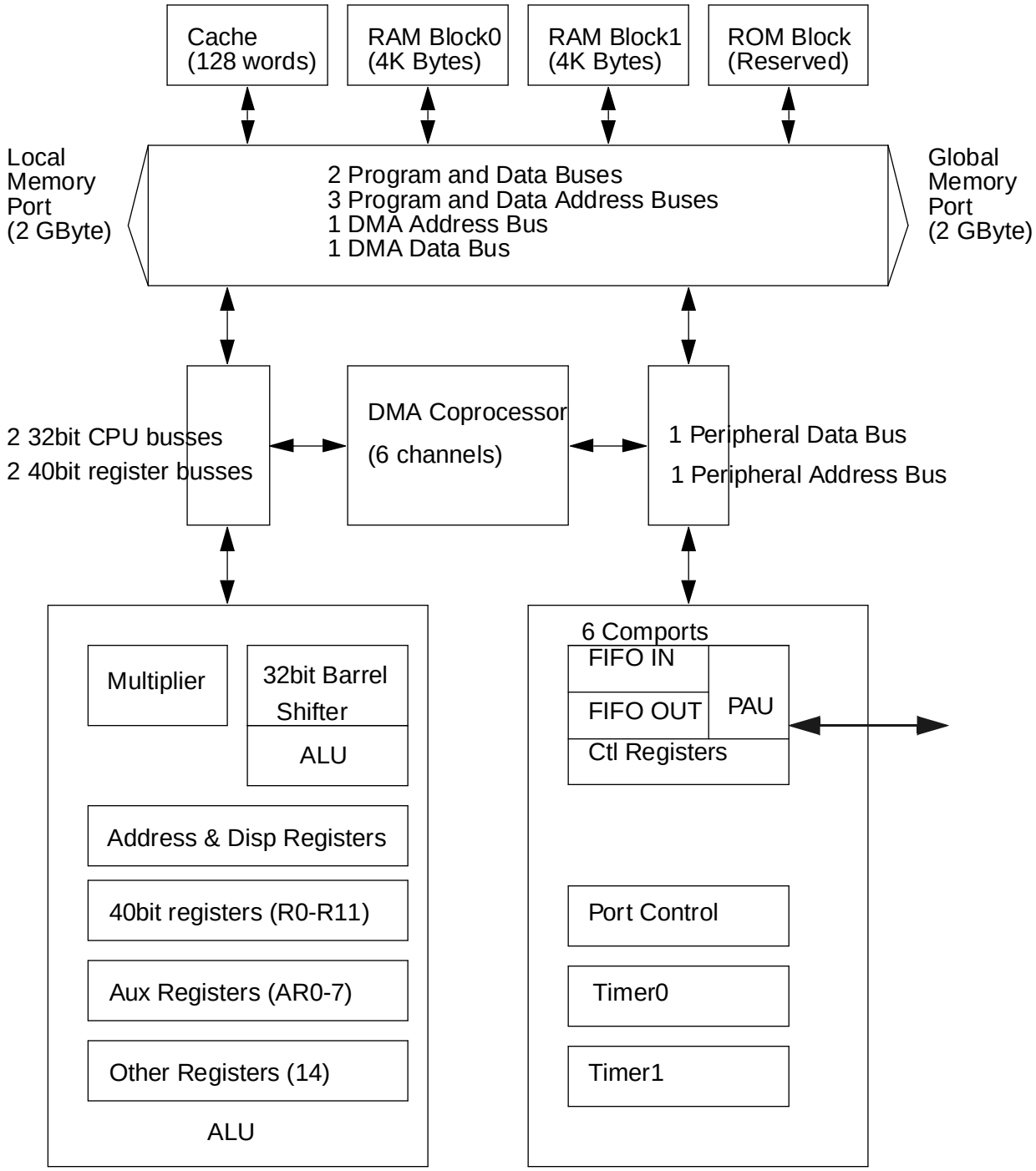
The TMS320C40 has a Harvard-like architecture (separated program- and data-addressing capability) with multiple internal buses. The interface to the outside world is done via two programmable memory ports. Two 4K bytes internal RAM blocks are available as well as a small cache of 512 bytes.

The C40 has 6 FIFO buffered communication ports, each offering up to 20 MBytes/s and two memory interface ports (100 MB/s), providing a total peak bandwidth of up to 320 MB/s. As the links are 8 bit parallel, and run at 20 MHz, they provide for a very high bandwidth. The communication ports can also directly be interfaced to peripheral devices.

Internally there is a six-channel DMA coprocessor that permits to execute memory operations while the CPU is handling computational tasks.

The CPU is upwards binary compatible with the CPU of the TMS320C30. It contains a single cycle floating point and integer multiplier and permits the parallel execution of instructions. Most instructions are single cycle. The floating point operations operate on 40bit floating point numbers.

36.1.1. TMS320C40 Chip Architecture



36.1.2. TMS320C40 Software Architecture

The TMS320C40 has a register based CPU architecture. The CPU contains following components :

- A Floating Point & Integer multiplier;
- ALU for floating point, integer and logical operators;
- 32bit barrel shifter
- Internal buses (CPU1/CPU2 and REG1/REG2)
- Auxiliary Register Arithmetic Units (ARAU)
- CPU register file

We will only list the CPU registers as these are the most important ones for the applications programmer.

Asm Symbol	Assigned Function Name
R0..R11	Extended Precision Registers, 40 bits These registers are used for 40 bit floating point, or for 32 bit integer operations. They cannot be used as pointers for indirect addressing.
AR0..AR7	Auxiliary Registers, 32 bits These registers can be used for integer operations, or for indirect addressing (pointers)
DP	Data Page Register Provides the upper 16 bits of a direct memory address
IR0, IR1	Index Registers Used for some of the indirect addressing modes
BK	Block Size Register Used for circular and bit-reversed addressing
SP	System Stack Pointer The C40 uses a full ascending stack, i.e. a push operation pre-increments the stack pointer
ST	Status Register
DIE	DMA Coprocessor Interrupt Enable Register
IIE	Internal Interrupt Enable Register
IIF	IIOF Flag Register

RS	Repeat Address Register
RE	Repeat End Address Register
RC	Repeat Counter
	Above 3 registers are used by the hardware looping instructions RPTS and RPTB
PC	Program Counter
IVTP	Interrupt Vector Table Pointer
TVTP	Trap Vector Table Pointer

36.1.3. Addressing Modes

The TMS320C40 has a rich set of addressing modes. The major classes are:

Register
Direct, using the DP register
Indirect, with many variations
Circular
Bit-reversed
Immediate
PC-relative

Most data processing instructions fall into one of three categories:

2 operands: one of these is always a register
3 operands: two of these are always a register
Parallel

36.1.4. Relevant documentation

It is highly recommended to read carefully the following documentation available from Texas Instruments. In this manual we only outline the main points.

TMS320C4x User's Guide (Texas Instruments, 1992 Edition)

TMS320 Floating Point DSP Assembly Language Tools (Texas Instruments, 1991)

TMS320 Floating Point DSP Optimizing C Compiler (Texas Instruments, 1991

36.2. Programming in C and assembly

This section introduces some implementation defined features of the Texas Instruments C compiler system. Please refer to the Compiler and Assembly Tools Manuals for full details.

36.2.1. Data representation

All integer based types (including characters) are represented as 32 bit words. This has the unusual consequence that sizeof (char) = sizeof (short) = sizeof (long) = 1.

All floating point types are represented using an internal format with an 8 bit exponent and a 24 bit mantissa. A floating point value stored in a register has a 32 bit mantissa, but this will be truncated when stored to memory. The full 40 bits can be read or written using assembly language, but this requires two memory words for each value.

All pointer types are represented as a 32 bit memory address. The C40 has two external memory interfaces and a large number of internal buses, but they all use a separate address space. As a consequence, every data item in memory can be referenced by a unique pointer.

36.2.2. Big and Small Models

The compiler supports two memory models. The only difference between these is in the way static data are accessed.

With the small model the DP (data page pointer register) is only initialized once. All static data references are made using single instructions, assuming the value of DP is valid. The means that the .bss section cannot span any 64K address boundaries.

With the big model, the C compiler explicitly loads the correct value in the DP register whenever a variable is accessed. Assembly language programs must do the same. In most cases, three cycles will be required to access a static data object, but the size of .bss is not limited to 64 K.

The Virtuoso system uses the small model for speed. Please note that this imposes a 64 K word limit on static data only. Data objects accessed via pointers can be of any size.

36.2.3. Parameter passing conventions

The compiler supports two methods for passing parameters to C subrou-

tines.

The standard method in C is to push parameters on the stack in reverse order. The called function initializes its frame pointer from the stack pointer value at entry, and finds the arguments at constant offsets from the frame pointer.

Using the register parameter model (-mr option), the first few (up to six) arguments are passed using a subset of the available registers. The exact assignment depends on the types of the arguments, and is fully documented in the Compiler Manual.

Virtuoso uses the register method for optimal performance. Mixing both models is normally not possible, so all application code must be compiled using the -mr option.

36.2.4. Memory sections for the C compiler and Virtuoso

The following sections are created by the C compiler:

.text	program code and string literals
.cinit	initial values, see remark below
.const	string literals and switch tables
.data	not used, see remark below
.bss	global and static variables
.stack	system stack (argument passing and local variables)
.sysmem	dynamic allocation memory pool

Contrary to normal COFF conventions, the C compiler does not use the .data section for initialized data. This is placed instead in .bss, and initialized at load time (ram model) or at run time (rom model) from data tables in the .cinit section. All assembly language modules in the Virtuoso system use the same method to allocate static data objects.

In addition to the standard sections listed above, the Virtuoso system creates some special sections. These are used to enable placement of critical pieces of code or data in fast internal RAM, to enhance performance. The Virtuoso special sections are:

nanok_code	nanokernel code
nanok_idle	process control structure for the low priority process
minik_code	microkernel swapper code
minik_stck	stack for the microkernel process
minik_args	microkernel input channel

system_vec	interrupt and trap vector table

Of these, nanok_code and minik_stck have the most profound effect on system performance. Some of the other special sections may be removed in future versions.

37. Programming the nanokernel

37.1. Introduction

The nanokernel provides the lowest level of functionality in the Virtuoso system. It is designed to perform extremely fast communication and context swapping for a number of *processes*. It also provides the entry points necessary to integrate interrupt handlers with the rest of the system. The price to pay for speed is that nanokernel processes and interrupt handlers must observe very strict rules regarding their use of CPU registers and the way they interact with each other.

From the point of view of the nanokernel, an application program consists of a collection of three types code modules:

- a single low priority (PRLO) process
- any number of high priority (PRHI) processes
- any number of interrupt handlers

It is important to understand what exactly is meant by a ‘process’. A process is a thread of execution that has both an identity and a private workspace. These two properties (which are logically equivalent) make it possible for a process to be swapped out, and wait for an external event while another process is allowed to continue. Interrupt handlers in contrast, do not have a private workspace.

The PRHI processes are scheduled in strict FIFO order, and must observe the special register conventions mentioned above. The PRLO process is assumed to be a C function (using the compiler register conventions), and must always be ready to execute - it is in fact the IDLE process of the nanokernel.

All communication inside the nanokernel is performed using *channels*. Several types of channel exist. The simplest type is used for synchronization only, and corresponds to a counting semaphore. Other types can be used to transfer data. Given a good understanding of how the nanokernel operates, a user could add his own channel types.

The Virtuoso microkernel (managing the TASKs), is build as an application on top of the nanokernel. The main component is a PRHI process that executes commands it receives from a channel. When the channel is empty, the microkernel process finds the next TASK to run, replaces the nanokernel IDLE process by that TASK and performs the additional register swappings required for C tasks. So when the nanokernel swaps in its IDLE process, it

actually executes one of the microkernel TASKs.

The nanokernel is not ‘aware’ of the manipulations performed by the microkernel. As far as it concerned, there is only one PRLO process, which it executes whenever no PRHI process is ready to continue. This makes it possible to use the nanokernel on its own.

37.2. Internal data structures

The user does not normally need to access the internal data structures used by the nanokernel. The documentation in this section is provided only for a better understanding of how the nanokernel operates.

A process is represented by a pointer to a Process Control Structure (PCS). For PRHI processes, the PCS occupies the first eight words of its stack. A static PCS is used for the IDLE process. The first word of a PCS is a pointer to another PCS, or NULL. This is used to build linked lists of processes. More details of the PCS will be introduced in the section on process management.

A channel is represented by a pointer to a Channel Data Structure (CDS). The first word of a CDS is a pointer to the PCS of a process waiting on that channel, or NULL. Other fields depend on the type of the channel and will be introduced in the section on nanokernel communications.

The following static variables are used by the nanokernel to keep track of the state of the system:

NANOK_PRHI :

Pointer to the PCS of the current PRHI process, or NULL if there is none.

NANOK_HEAD, NANOK_TAIL :

Head and tail pointers for a linked list of PRHI processes that are ready to run. When a process becomes ready to execute, it is added to the tail of the list. When the current PRHI process is swapped out, the process at the head of the list is removed, and becomes the current process. If the list is empty, the PRLO process is swapped in.

NANOK_PRLO :

Pointer to the PCS of the PRLO process. This is a constant as far as the nanokernel is concerned. The microkernel modifies this pointer ‘behind the scenes’.

NANOK_CRIT : the critical level.

This is the number of interrupt handlers running with global interrupts enabled that have not yet terminated. All process swapping is disabled while this value is not zero. Since the C40 does not support hardware interrupt levels, some cooperation is required from interrupt handlers in order to keep this value up to date.

These five variables, together with the IDLE process PCS, actually represent ***all*** the global information maintained by the nanokernel (all other static variables are effectively constants). In other words, the nanokernel only knows about processes that are ready to execute. It does not maintain a list of all processes or channels.

Symbolic constants for accessing kernel variables and elements of a PCS are defined in NANOK.INC.

37.3. Process management.

The nanokernel variables are initialized as follows :

```
NANOK_PRHI   = 0;
NANOK_HEAD   = 0;
NANOK_TAIL   = &(NANOK_HEAD);
NANOK_PRLO   = &(PCS for IDLE process);
NANOK_CRIT   = 0;
```

In other words, when an application is started, the single thread of execution that exists at that time will be adopted by the nanokernel as its low priority IDLE process.

In the current version of the nanokernel, all PRHI processes must be created and started by the PRLO process (it is possible to do this from within another PRHI process, but no services are provided to support this).

Three steps are required to create a process:

- create a stack for the process.
- initialize the PCS.
- start the process.

The stack can be placed anywhere in memory. It can be a C array of integers, a memory block allocated by malloc () or (or even KS_Alloc ()), or a pointer to the start of a named section.

The function **_init_process (stack, entry, ar4, ar5)** is used to initialize the PCS. It writes the following values to the first 8 words of the stack:

0	PR_LINK	0	link pointer
1	PR_SSTP	stack + 7	saved stack pointer
2	PR_PAR3	0	saved AR3, not used for PRHI
3	PR_PAR4	ar4	initial / saved value of AR4
4	PR_PAR5	ar5	initial / saved value of AR5
5	PR_BITS	0	flags, not used for PRHI
6	PR_PEND	NANOK_TRMP	pointer to terminate code
7	PR_PRUN	entry	pointer to entry point

Calling **_start_process (stack)** starts the process. As the caller is the PRLO process, there can be no other PRHI processes and the new process will start execution immediately. Control returns to the caller when the new process terminates or deschedules by waiting on a channel.

The first time a PRHI process is swapped in, it 'continues' from its entry point. The stack pointer will point to the PR_PEND field in the PCS (SP = stack + 6). A process terminates by 'returning' to the address in this field. The code at NANOK_TRMP invokes the nanokernel swapper to switch to the next process. To restart a terminated process, repeat the calls to _init_process () and _start_process ().

When a PRHI process is swapped in, AR3 points to the start of the PCS. A process can create local variables by incrementing the initial stack pointer by the number of words required. The first available word is the entry point field, at AR3 + 7.

37.4. Nanokernel communications

A *channel type* is defined by a data structure and a number of nanokernel services that operate on it. Each instance of the data structure is called a *channel*. Channels can provide both process synchronization and data communication.

The nanokernel does not itself use or create channels. However, the services that operate on channels should be considered part of the nanokernel, as they may modify the process FIFO or invoke the nanokernel swapper.

All channels have an internal state. What exactly is represented by the state depends on the type of the channel - typically this will be the occurrence of an event or the availability of data.

An operation on a channel can consist of any combination of the following action types:

Wait :

A channel operation has a waiting action if the calling process can be descheduled as a result of the call. The process is then said to 'wait on the channel'.

Signal :

A channel operation has a signaling action if a waiting process can be rescheduled as a result of the call. If the current process is the PRLO process, a process swap will be performed.

Test & modify :

A test & modify action modifies the state of a channel and returns information about it, without changing the execution state of any process.

Three channel types are predefined in the current nanokernel implementation. It is possible to create new channel types if necessary; an example will be given at the end of this chapter. A full description of the nanokernel services for each of these channel types can be found in the alphabetical listing in the next chapter.

37.4.1. C_CHAN - Counting channel

This is an implementation of a counting semaphore. It is typically used by interrupt handlers to reschedule a process that was waiting for the interrupt. The C_CHAN structure has two fields:

0	CH_PROC	pointer to the PCS of a waiting process, or NULL
1	CH_NSIG	event counter.

Two nanokernel services are available for this channel type:

PRHI_WAIT	wait action
PRHI_SIG	signal action

37.4.2. L_CHAN - List channel

This type of channel maintains a linked list of memory blocks, using the first word in each block as a link pointer. The microkernel uses this type to implement its free lists of command packets, data packets and timers. If used for data communication, it behaves as a LIFO buffer.

The L_CHAN structure has two fields:

0	CH_PROC	pointer to the PCS of a waiting process, or NULL
1	CH_LIST	pointer to first element of the linked list, or NULL

The nanokernel services that operate on this type are:

PRHI_GETW	wait action
PRHI_GET	test & modify action
PRHI_PUT	signal action

37.4.3. S_CHAN - Stack channel

This type of channel uses a memory block as a data stack. The microkernel uses a stack channel to input commands from tasks and the network drivers, and to receive events from interrupt handlers.

The S_CHAN structure has three fields:

0	CH_PROC	pointer to the PCS of a waiting process, or NULL
1	CH_BASE	pointer to base of the stack
2	CH_NEXT	pointer to the next free word on the stack

The nanokernel services that operate on this type are:

PRHI_POPW	wait action
PRHI_POP	test & modify action
PRHI_PSH	signal action

37.5. Register conventions

In order to understand the register conventions adopted by the Virtuoso nanokernel, the following register sets should be introduced:

```
CSAVE = R4-R8, AR3-AR7, DP, SP
CFREE = ST, R0-R3, R9-R11, AR0-AR2, IR0, IR1, BK, RC, RE, RS
NSWAP = AR3-AR5, SP
SYSSET = DIE, IIE, IIF, IVTP, TVTP
INTSET = ST, R11, AR0-AR2
```

The CSAVE and CFREE sets are defined by the procedure calling standard of the C compiler. CSAVE is the set of registers that are preserved across a subroutine call - if a function uses any of these, it must restore the initial value on return. CFREE is the set of registers that are freely available to all functions - the caller of a subroutine is responsible for preserving them if necessary. The definition of these two sets largely determine what the micro-

kernel is expected to do when swapping tasks. When a task is swapped out as a result of calling a kernel service (which to the task is just a C function), only the CSAVE set need be saved. When a task is preempted (which means that an interrupt handler has woken up the kernel), the CFREE set must be saved as well. Actually, since most of the microkernel is written in C, the CFREE set must be saved before the actual service requested by the interrupt handler is called.

Note : ST is included in the CFREE set because it contains the flags tested by the conditional instructions (bits 0 - 6). Other bits in ST have system control functions, and should be treated as part of SYSSET. In particular, for correct operation of the nanokernel, the SET COND flag (bit 15) must remain reset at all times.

The SYSSET registers are used for system and peripheral control only. They are never swapped, and should be regarded as global resources. Only very low level routines (such as hardware drivers) will ever need to access these registers.

The INTSET registers are those that must have been pushed on the stack when an interrupt handler terminates and wakes up the kernel by calling one of the ENDISR services (this is discussed in more detail in the section on interrupt handling below). At that point, the nanokernel needs some registers to work with. It would be a waste of time to pop all registers saved by the ISR, only to have to push them again when entering the kernel.

The registers in NSWAP are saved and restored by the nanokernel when swapping processes. For the PRLO process (assumed to be a C function, using AR3 as its frame pointer) the nanokernel will save and restore AR3 in the normal way. When a PRHI process is swapped in, AR3 will be set to point to its process control structure. A PRHI process can use AR3 to access local variables created in its workspace, and should normally not modify this register. If it does, the initial value can always be reloaded from NANOK_PRHI. AR3 must point to the PCS whenever the process calls a nanokernel service, and when it terminates.

Given these definitions, it is now possible to determine which registers can be used by a PRHI process, and how it should call C functions if this is required.

The NSWAP set is always available, but note the special use of AR3.

If a PRHI process is swapped in as the result of a C function call by the PRLO process, then the CFREE set is available for use by the PRHI process. This means that the process can safely call any C function. It should of course save those registers in CFREE that it wants to preserve across the

call.

If a PRHI process is swapped in as the result of an interrupt handler calling an ENDISR service, then the INTSET registers are available to the PRHI process. Before calling a C function, the process must save the set CFREE - INTSET, and it must restore the same registers before it is swapped out (this is always possible, since a PRHI process is never preempted).

Note that INTSET is a subset of CFREE, so the minimal register set that is always available is INTSET + NSWAP. Also, NSWAP is a subset of CSAVE. In fact NSWAP and INTSET have the same meaning to the nanokernel level that CSAVE and CFREE have to the C level. The two set inclusions mentioned above also mean that the nanokernel is already doing part of the job of swapping microkernel tasks. This is, of course, no coincidence.

37.6. Interrupt handling

In the Virtuoso system model, interrupt handlers are the interface between asynchronous events and the processes that are waiting for them. To be useful, most interrupt handlers will have to interact with the rest of the system at some time. Using flags to be 'polled' by the foreground process is usually not an acceptable practice in a real-time system. This method introduces a 'superloop' structure into the application, with all its inherent problems.

In a system using the nanokernel, interrupt handlers can communicate with processes using the same channel operations that are available to processes. There are, however, some restrictions.

In contrast to a process, an interrupt service routine does not have a private workspace - it executes on the stack of whatever process was interrupted. Even if a processor supports a separate interrupt mode stack (the C40 doesn't), this will be shared by all interrupt handlers. An ISR of course can itself be interrupted by another one, so any number of interrupt handlers can be piled on top of each other on the same stack, owned by the current process. This has some important consequences:

1. If an ISR calls a channel service that has a SIGNAL action, any process swap that results from this call must be delayed until all interrupt handlers have terminated. These implies that only the PRHI_ type of channel operations can be used, as these do not invoke the swapper for a SIGNAL action (there is no need to swap, as the caller already has highest priority). When the last stacked interrupt terminates, the swapper must be called to verify if a swap from the PRLO process to a PRHI process is necessary.

2. An ISR must never call any channel service that has a WAIT action. Doing so would also block all other interrupt handlers that are stacked below it, as

well as the current process. Another way of seeing this is that an ISR cannot wait for something because it doesn't have a separate identity - the producer of the external event (another ISR) has no means of representing who is waiting for it.

The C40 does not support a separate 'interrupt mode' - an ISR cannot determine whether it interrupted a foreground process or another ISR by examining the processor state only. The Virtuoso nanokernel defines a software protocol, that must be observed by all interrupt handlers, to implement the logic described above. In this system, an interrupt handler can run at either of two 'execution levels':

The ISR0 level

All interrupt handlers are entered at this level. Interrupts are disabled globally (GIE = 0), and the ISR is not allowed to re-enable them. Consequently, at any time there can be only one ISR running at this level. It runs entirely in the background and remains unknown to the kernel until it terminates by calling the ENDISR0 service. At that point, the nanokernel will verify if a process swap is required and allowed. The condition tested is the logical AND of

- NANOK_PRHI = 0 the current process is PRLO
- NANOK_HEAD != 0 a PRHI process is ready to execute
- NANOK_CRIT = 0 no other ISR stacked below this one

If the interrupt handler did not call a channel operation with a SIGNAL action, the condition above can never be satisfied, so in this case the ISR is allowed to terminate by a normal ISR exit sequence (popping all saved registers and RETI).

The ISR1 level

An interrupt handler running at the ISR1 level is known to the kernel, and is allowed to modify the global interrupt enable state. Calling the SETISR1 service moves an interrupt handler from the ISR0 to the ISR1 state. The kernel increments NANOK_CRIT and returns with interrupts enabled (GIE = 1). After having called SETISR1, the ISR is allowed to directly modify the GIE bit as often as required. It must always terminate by calling the ENDISR1 service. The kernel decrements NANOK_CRIT and then performs the same test as for ENDISR0.

SETISR1 is implemented so that it can be called very early in an ISR. Only the ST and R11 registers are modified, and must have been saved.

Sophisticated interrupt priority schemes can be set up by manipulating individual interrupt enable bits before calling SETISR1. If this is done, the origi-

nal state of the IIE and IIF registers must be restored before terminating the ISR.

The ENDISR0 and ENDISR1 calls never return to the caller. When calling these services, the ISR should leave the stack in a defined state, so that the following code would perform a correct return to the interrupted context :

```
pop ar2
pop ar1
pop ar0
popf r11
pop r11
pop st
reti
```

If no process swap is performed, the kernel will actually perform the interrupt exit sequence listed above. If the current process is swapped out, the return address and the INTSET registers remain on the stack of the interrupted process until it is swapped in again.

The code fragment below show the skeleton of an interrupt handler running at the ISR1 level. Please note that the three instructions following an LAT are actually executed before the LAT itself.

```
push st                 ; push the INTSET registers
push r11
pushf r11
lat SETISR1             ; re-enable interrupts ASAP
push ar0                ; continue saving INTSET, or
push ar1                ; modify IIE, IIF
push ar2                ;

....                    ; body of ISR, interrupts are enabled
                        ; st, r11, ar0, ar1, ar2 available

lat ENDISR1
nop                     ; any three useful instructions
nop                     ; e.g. pop registers not in INTSET,
nop                     ; or restore IIE, IIF
```

37.7. Communicating with the microkernel

As mentioned before, the Virtuoso microkernel is implemented as a PRHI

process. It uses a single stack based channel to receive commands from the tasks, the network drivers, other PRHI processes and interrupt handlers. A pointer to this channel is exported in the C variable K_ArgsP.

Two types of data can be pushed onto this channel:

1. Small integers (0 - 63) are interpreted as events. Events are simple binary signals that a task can wait for using the KS_EventW () service. Most events will be generated by interrupt handlers and driver processes. For the C40 version, event numbers have been assigned as follows:

0 - 31	interrupts enabled in the IIE register
32 - 35	external interrupts iiof0 - iiof3
36 - 47	used by the raw link drivers
48 - 55	reserved for Virtuoso internal use
56 - 63	free

Event numbers 0 - 35 should be used to represent interrupts. This convention makes the event number the same as the interrupt number used in KS_EnableISR ().

The remaining numbers are used for events that do not directly correspond to an interrupt. For example, the dma based raw link drivers install an ISR that accepts a DMA interrupt, reads the DMA status, and generates the receiver ready and / or transmitter ready events.

The interrupt handlers installed by timer0_driver () and timer1_driver generate event 48. This is used within the kernel to increment the TICKS time.

The code fragment below shows how to send an event from an ISR or a PRHI process:

```
 .ref _K_ArgsP
 .def _my_isr
 ...
_my_isr
 ...
 lat PRHI_PSH           ; send event to microkernel
 ldi @_K_ArgsP, ar1     ; microkernel input channel
 ldi EVENT_NUM, ar2     ; event number
 nop
 ...
```

2. All other values pushed onto the microkernel input channel are interpreted as a pointer to a command packet. Command packets are the primary form of communication used within the Virtuoso system. They are used by the

tasks to request microkernel services, sent across the Virtuoso network to implement remote kernel calls, and put on waiting lists to represent a task that is waiting for something. It is outside the scope of this manual to present a complete description of the command packet data format. The basic structures and the command codes are defined in K_STRUCT.H.

The microkernel maintains a list of free command packets, implemented as a list based channel. A pointer to this channel is exported in the C variable K_ArgsFreeP. Other PRHI processes can get command packets from this pool, but they must never wait on the channel (i.e. always use PRHI_GET). If the list is empty, correct behavior is to call YIELD and try again later.

In the Virtuoso network, the Srce field of a command packet identifies the sending node, and it is used as a return path for reply messages. The same field also has a secondary function: since all packets sent or received through the network are allocated from the K_ArgsFree list, they should be deallocated after use. The network transmitters always free a packet after it has been sent. The microkernel deallocates a packet if the Srce field is not zero. Consequently, command packets not allocated from the free list must have their Srce field set to zero to prevent deallocation.

Note: we are aware of the fact that this logic is a bit confusing. Future versions of the microkernel will probably use a separate flag to indicate if a packet was dynamically allocated.

Interrupt handlers and PRHI processes can request a microkernel service by building a command packet, and pushing a pointer to it on the microkernel input channel. The only services that can be safely called are the equivalents of KS_Signal and KS_SignalM, and the DRIVER_ACK service. Also note that using events will be faster than signals.

The code fragments below show how to perform a KS_Signal () or KS_SignalM () call from within an ISR. In this example the command packet is created and initialized in C, but the same thing could be done entirely in assembly language.

The function install_my_isr () is called to initialize the command packet and install the ISR:

```
K_ARGS CP1, *CP1P; /* command packet for use by my_isr */
K_SEMA SLIST1 [] = { SEMA1, SEMA2, SEMA3, ..., ENDLIST };
extern void my_isr (void);

void install_my_isr (...)
 {
 ...
```

```
 /* create a pointer to the command packet */
 CP1P = &CP1;
 /* initialize CP1 for a KS_Signal (SEMA1) service */
 CP1.Srce = 0;
 CP1.Comm = SIGNALS;
 CP1.Args.s1.sema = SEMA1;
 /* or for a KS_SignalM (SLIST1) service */
 CP1.Srce = 0;
 CP1.Comm = SIGNALM;
 CP1.Args.s1.list = SLIST1;
 /* install the ISR */
 KS_EnableISR (..., my_isr);
 ...

 }
```

For the ISR, something like the code listed below will be required:

```
 .ref _CP1P
 .ref _K_ArgsP
 .def _my_isr
 ...
_my_isr
 ...
 lat PRHI_PSH           ; signal semaphore(s)
 ldi @_K_ArgsP, ar1     ; microkernel input channel
 ldi @_CP1P, ar2        ; pointer to command packet
 nop
 ...
```

37.8. Virtuoso drivers on TMS320C40

Drivers are the interface between the processor and peripheral hardware, and the application program. They normally serve two purposes: data communication, and synchronization. As polling is not a recommended practice in a real-time system, most drivers will use interrupts in one way or another.

The Virtuoso system does not provide a standard interface to drivers - this allows the application writer to optimize this important part of their implementation. Some basic services, that will be required for almost all drivers, are provided.

Most low-level details have already been described in the previous sections on interrupt handling and communication with the microkernel. At a higher level, a typical driver can usually be divided into three functional parts:

1. The first component is a function to install the driver. This should initialize the hardware and any data structures used, and install interrupt handlers for the driver. A call to this function is usually placed inside a DRIVER statement in the system definition file. The SYSGEN utility copies this call into a function named init_drivers () it generates in the node#.c files. The init_drivers () subroutine is called by kernel_init () just before it returns.

2. Most drivers will provide one or more subroutines that can be called from the task level, and that implement the actual functionality of the driver. At some point, these functions will call KS_EventW () or KS_Wait () to make the calling task wait for the completion of the driver action.

3. One or more interrupt handlers are required to generate the events or signals waited for by these subroutines.

In the simplest case, the only actions required from the ISR will be to service the hardware and to reschedule a waiting task, and all data handling and protocol implementation can be done at the task level. This method can be used if the interrupt frequency is not too high (< 1000Hz).

For higher data rates, some of the task code should be moved to the ISR, in order to reduce the number of task swaps. In most cases, the actions required from the interrupt handler will not be the same for each interrupt, and some form of state machine will have to be implemented into the ISR.

If the number of possible states grows, it is often much easier to use one or more PRHI processes to implement the protocol. Processes can wait for interrupts at any number of places in their code, and each of these points represents a state of the system. As an example, the Virtuoso network drivers have been designed using this method.

The microkernel provides the DRIVER_REQ and DRIVER_ACK services to interface tasks to drivers based on PRHI processes. At the time of writing, these services are the subject of further development. They will be documented in a separate application note, and in the next release of this manual.

38. Alphabetical List of nanokernel entry points

In the pages to follow, all Virtuoso nanokernel entry points are listed in alphabetical order. Most of these are C40 trap routines, some are C callable.

- BRIEF . . . . . Brief functional description
- CLASS . . . . . One of the Virtuoso nanokernel service classes of which it is a member.
- SYNOPSIS . . . . . The ANSI C prototype (C callable), or

  Assembly language calling sequence (Traps)
- RETURN VALUE . . The return value, if any (C callable only).
- ENTRY CONDITIONS Required conditions before call (Traps only)
- EXIT CONDITIONS. Conditions upon return of the call (Traps only)
- DESCRIPTION . . . A description of what the Virtuoso nanokernel service does when invoked and how a desired behavior can be obtained.
- EXAMPLE . . . . . One or more typical Virtuoso nanokernel service uses.
- SEE ALSO. . . . . List of related Virtuoso nanokernel services that could be examined in conjunction with the current Virtuoso nanokernel service.
- SPECIAL NOTES . . Specific notes and technical comments.

38.1. _init_process

• BRIEF . . . . . Initialize a nanokernel process

• CLASS. . . . . Process management

• SYNOPSIS . . . . . void _init_process (void *stack, void entry(void), int ar4, int ar5);

• DESCRIPTION . . . This C function initializes the process control structure of a process. It must be called before the process is started using start_process (). The entry point, the initial values for AR4 and AR5 and some internal variables are written to the PCS.

• RETURN VALUE . . none

• EXAMPLE . . . . . In this example, two processes using the same code but different parameters are initialized and started.

```
int adc1[100];          /* stack for first process */
int adc2[100];          /* stack for second process */
extern void adc_proc (void); /* process code */
extern struct adc_pars ADC_Params [2]; /* parameter structs */

_init_process (adc1, adc_proc, &ADC_Params [0], 0);
_init_process (adc2, adc_proc, &ADC_Params [1], 0);
_start_process (adc1)
_start_process (adc2)
```

• SEE ALSO. . . . . _start_process

• SPECIAL NOTES . .

38.2. _start_process

- BRIEF . . . . . Starts a nanokernel process from the low priority context
- CLASS . . . . . Process management
- SYNOPSIS . . . . . void _start_process (void *stack);
- DESCRIPTION . . . Starts a nanokernel process by making it executable. The process must have been initialized before. The process will start executing immediately. This call returns when the started process deschedules or terminates.
- RETURN VALUE . . none
- EXAMPLE . . . . .

```
int wsp1[100]
int wsp2[100]
extern void proc1 (void);
extern void proc2 (void);
int N = 1000;
_init_process (wsp1, proc1, 0, N)
_init_process (wsp2, proc2, 0, N)
_start_process (wsp1)
_start_process (wsp2)
```

- SEE ALSO. . . . . _init_process ()
- SPECIAL NOTES . . This function cannot be used from within a high priority nanokernel process. It must be called from the C main () function or by a microkernel task only.

38.3. ENDISR0

• BRIEF . . . . . Terminates a level 0 ISR and conditionally invokes the process swapper

• CLASS. . . . . Interrupt service management

• SYNOPSYS . . . . . LATcond ENDISR0

```
ENDISR0 is defined in TRAPS.INC
```

• DESCRIPTION . . . This is the normal way to terminate an ISR running at level 0 (global interrupts disabled). It must be called if the ISR has previously used any nanokernel service that can reschedule a process, e.g. PRHI_SIG. If the ISR did not interact with the kernel, a normal ISR exit sequence (popping saved registers and RETI) can be used instead.

A nanokernel process swap will be performed IFF

- the calling ISR interrupted the nanokernel low priority process
- a high priority process is ready to execute

• ENTRY CONDITIONS The ISR should have saved the interrupted context so that the exit sequence listed below would correctly terminate the ISR.

```
pop ar2
pop ar1
pop ar0
popf r11
pop r11
pop st
reti
```

• EXIT CONDITIONS . This call terminates the ISR and does not return.

• EXAMPLE . . . . . This ISR accepts the IIOF0 external interrupt and sends EVENT 32 to the microkernel.

```
.include “traps.inc”
.ref _K_ArgsP
.def __iiof0_isr
.text
__iiof0_isr
push st
push r11
pushf r11
push ar0
push ar1
```

```
push ar2
....acknowledge interrupt if necessary
lat PRHI_PSH
ldi @_K_ArgsP, ar1      ; pointer to microkernel input channel
ldi 32, ar2             ; event number
nop
lat ENDISR0             ; terminate the ISR
nop
nop
nop
```

• SEE ALSO. ENDISR1, SETISR1

38.4. ENDISR1

• BRIEF . . . . . Terminates a level 1 ISR and conditionally invokes the process swapper

• CLASS. . . . . Interrupt service management

• SYNOPSYS . . . . . LATcond ENDISR1

```
ENDISR1 is defined in TRAPS.INC
```

DESCRIPTIONThis entry point must be called to terminate an ISR running at level 1 (global interrupts enabled). It decrements the level 1 interrupt counter and preforms a nanokernel process swap IFF

- the calling ISR interrupted the PRLO process
- a high priority process is ready to execute

• ENTRY CONDITIONSThe ISR should have saved the interrupted context so that the exit sequence listed below would correctly terminate the ISR.

```
pop ar2
pop ar1
pop ar0
popf r11
pop r11
pop st
reti
```

• EXIT CONDITIONS. This call terminates the ISR and does not return.

• EXAMPLE . . . . . This ISR accepts the IIOF0 external interrupt and sends a signal to two hi-priority processes.

```
.include “traps.inc”
.ref _chan1_ptr
.ref _chan2_ptr
.def __iiof0_isr
.text
__iiof0_isr
push st
push r11
pushf r11
lat SETISR1             ; move to ISR level 1
push ar0
push ar1
push ar2
```

```
; ... interrupts are enabled from this point
; ... any other useful code
lat PRHI_SIG            ; send signal on chan1
ldi @_chan1_ptr, ar1
nop                     ; or other useful instructions
nop                     ; executed before the PRHI_SIG
; ...
lat PRHI_SIG            ; send signal on chan2
ldi @_chan2_ptr, ar1
nop
nop
; ...
lat ENDISR1             ; terminate the ISR
nop
nop
nop
```

• SEE ALSO. SETISR1

• SPECIAL NOTES . . A normal interrupt exit (popping saved registers and RETI) is not allowed for an ISR running at level 1.

38.5. K_taskcall

- BRIEF . . . . . Send a command packet to the microkernel process
- CLASS. . . . . Process communication
- SYNOPSYS . . . . . void K_taskcall (K_ARGS *A);
- DESCRIPTION . . . This C-callable function is used by all KS_... services to send command packets to the microkernel process.
- RETURN VALUE . . none
- EXAMPLE . . . . . .
- SEE ALSO. . . . . PRLO_PSH
- SPECIAL NOTES . . This function must be called by microkernel tasks only.

38.6. KS_DisableISR()

• BRIEF . . . . . Remove an ISR from the interrupt vector table

• CLASS . . . . . Interrupt service management

• SYNOPSYS . . . . . void KS_DisableISR (int isrnum);

• DESCRIPTION . . . This C function is equivalent to KS_EnableISR (isrnum, NULL). The interrupt is disabled, and the corresponding entry in the interrupt vector table is cleared.

• RETURN VALUE . . none

• EXAMPLE . . . . . .

```
KS_DisableISR (34) ;    /* remove the IIOF2 handler */
```

• SEE ALSO. . . . . KS_EnableISR, SYSVEC

• SPECIAL NOTES . . Interrupt numbers are:

- 0..31 for interrupts enabled in the IIE register
- 32..35 for IIOF0..IIOF3

38.7. KS_EnableISR

- BRIEF . . . . . Install an ISR and enable the corresponding interrupt.
- CLASS. . . . . Interrupt service management
- SYNOPSYS . . . . . void KS_EnableISR (int isrnum. void isr (void));
- DESCRIPTION . . . This C function is used to install, remove, or replace an interrupt handler. It takes two parameters: an interrupt number, and a pointer to an ISR. The pointer is entered into the interrupt vector table, and if it is not zero, the corresponding interrupt enable bit is set in the IIE or IIF register. If the pointer is NULL, the interrupt is disabled.
- RETURN VALUE . . none
- EXAMPLE . . . . . .

```
extern void _iiof2_isr (void);
KS_EnableISR (34, _iiof2_isr);
```

- SEE ALSO. . . . . KS_DisableISR, SYSVEC
- SPECIAL NOTES . . Interrupt numbers are:
  - 0..31 for interrupts enabled in the IIE register
  - 32..35 for IIOF0..IIOF3

38.8. PRHI_GET

• BRIEF . . . . . Remove next packet from linked list channel

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . LATcond PRHI_GET

```
PRHI_GET is defined in TRAPS.INC
```

• DESCRIPTION . . . If the channel is not empty, the first packet is removed from the linked list, the Z flag is reset, and a pointer to the packet is returned. If the channel is empty, the Z flag is set and a NULL pointer is returned. The calling process is never swapped out as a result of calling this service.

• ENTRY CONDITIONS

```
AR1 = pointer to linked list channel struct
```

• EXIT CONDITIONS. If the list is not empty:

```
AR0 and R11 are corrupted
AR2 = pointer to removed list element
the Z flag is cleared
```

If the list is empty

```
AR0 and R11 are corrupted
AR2 = 0
the Z flag is set
```

• EXAMPLE . . . . . .

```
.include “traps.inc”
; assume AR5 points to a parameter struct
; obtain a free packet
lat PRHI_GET
ldi *+ar5(FREE_LIST), ar1
nop
nop
bz list_empty           ; test if call failed
; use packet pointed to by AR2
```

• SEE ALSO. . . . . PRHI_GETW, PRHI_PUT

• SPECIAL NOTES . . This service must not be called from the low priority context.

38.9. PRHI_GETW

• BRIEF . . . . . Get next packet from linked list channel, or deschedule

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . LATcond PRHI_GETW

```
PRHI_GETW is defined in TRAPS.INC
```

• DESCRIPTION . . . If the channel is not empty, the first packet is removed from the linked list and a pointer to it is returned. If the channel is empty, the calling process is swapped out and set to wait on the channel. It will be rescheduled by the next call to the PRHI_PUT service on the same channel.

• ENTRY CONDITIONS

```
AR1 = pointer to linked list channel struct
AR3 = pointer to PCS of calling process
```

• EXIT CONDITIONS.

```
AR4 = pointer to list element
AR0, AR1, AR2, R11 and ST are corrupted
```

• EXAMPLE . . . . . .

```
.include “traps.inc”
; assume AR5 points to a parameter struct
; obtain next packet from free_list
lat PRHI_GETW
ldi *+ar5(FREE_LIST), ar1 ; pointer to free list channel
nop
nop
; use packet pointed to by AR4
```

• SEE ALSO. . . . . PRHI_GET, PRHI_PUT

• SPECIAL NOTES . . This service must not be called from the low priority context or by an isr.

38.10. PRHI_POP

• BRIEF . . . . . Remove next element from a stack channel

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . LATcond PRHI_POP

```
PRHI_POP is defined in TRAPS.INC
```

• DESCRIPTION . . . If the stack is not empty, the top element is removed and returned to the caller. The Z flag is reset. If the stack is empty, the Z flag is set and an undefined value is returned. The calling process is never swapped out as a result of calling this service.

• ENTRY CONDITIONS

```
AR1 = pointer to stack channel struct
```

• EXIT CONDITIONS. If the stack is not empty:

```
AR0 and R11 are corrupted
AR2 = the element removed from the stack
the Z flag is cleared
```

If the stack is empty:

```
AR0 and R11 are corrupted
AR2 = undefined
the Z flag is set
```

• EXAMPLE . . . . . .

```
.include “traps.inc”
; assume AR5 points to a parameter struct
; obtain top of stack element
lat PRHI_POP
ldi *+ar5(INPUT_CHAN), ar1
nop
nop
bz stack_empty          ; test if call failed
; use stack element in AR2
```

• SEE ALSO. . . . . PRHI_POPW, PRHI_PSH

• SPECIAL NOTES . . This service must not be called from the low priority context.

38.11. PRHI_POPW

• BRIEF . . . . . Remove next element from a stack channel, or deschedule

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . LATcond PRHI_POPW

```
PRHI_POPW is defined in TRAPS.INC
```

• DESCRIPTION . . . If the stack is not empty, the top element is removed and returned to the caller. If the stack is empty, the calling process is swapped out and set to wait on the channel. It will be rescheduled by the next call to the PRHI_PSH service on the same channel.

• ENTRY CONDITIONS

```
AR1 = pointer to stack channel struct
AR3 = pointer to PCS of calling process
```

• EXIT CONDITIONS.

```
AR4 = element removed from the stack
AR0, AR1, AR2, R11 and ST are corrupted
```

• EXAMPLE . . . . . .

```
.include “traps.inc”
; assume AR5 points to a parameter struct
; obtain element at top of stack
lat PRHI_POPW
ldi *+ar5(INPUT_CHAN), ar1
nop
nop
; use stack element in AR4
```

• SEE ALSO. . . . . PRHI_POP, PRHI_PSH

• SPECIAL NOTES . . This service must not be called from the low priority context or by an isr.

38.12. PRHI_PUT

• BRIEF . . . . . Add a packet to a linked list channel

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . LATcond PRHI_PUT

```
PRHI_PUT is defined in TRAPS.INC
```

• DESCRIPTION . . . If a process is waiting on the channel, the pointer to the packet is passed on, and the waiting process is rescheduled. Otherwise the packet is linked in at the head of the list. In either case, control returns to the caller.

• ENTRY CONDITIONS

```
AR1 = pointer to channel
AR2 = pointer to packet to add to the list
```

• EXIT CONDITIONS.

```
AR0, AR1, R11 and ST are corrupted
All other registers are preserved
```

• EXAMPLE . . . . . .

```
.include “traps.inc”
; assume AR5 points to a parameter struct
; assume AR0 points to packet to add to the list
lat PRHI_PUT
ldi *+ar5(FREE_LIST), ar1
ldi ar0, ar2
nop
; the packet is added to the list
```

• SEE ALSO. . . . . PRHI_GET, PRHI_GETW

• SPECIAL NOTES . . This entry point must not by called from the low priority context, but it can be used by interrupt handlers.

The first word of the packet is used as a link pointer, and will be overwritten.

38.13. PRHI_PSH

• BRIEF . . . . . Push a word on a stack channel

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . LATcond PRHI_PSH

```
PRHI_PSH is defined in TRAPS.INC
```

• DESCRIPTION . . . If a process is waiting on the channel, the data word is passed on, and the waiting process is rescheduled. Otherwise the data word is pushed on the stack. In either case, control returns to the caller.

• ENTRY CONDITIONS

```
AR1 = pointer to channel
AR2 = data word to push
```

• EXIT CONDITIONS.

```
AR0, AR1, R11 and ST are corrupted
All other registers are preserved
```

• EXAMPLE . . . . . .

```
.include “traps.inc”
.ref _K_ArgsP           ; microkernel input stack
; send a command packet to the microkernel
; assume ar0 points to the command packet
lat PRHI_PSH
ldi @_K_ArgsP, ar1
ldi ar0, ar2
nop
```

• SEE ALSO. . . . . PRHI_POP, PRHI_POPW

• SPECIAL NOTES . . This entry point must not by called from the low priority context, but it can be used by interrupt handlers.

38.14. PRHI_SIG

• BRIEF . . . . . Send an event on a signal channel

• CLASS . . . . . Process communication

• SYNOPSIS . . . . . LATcond PRHI_SIG

```
PRHI_SIG is defined in TRAPS.INC
```

• DESCRIPTION . . . If a process is waiting on the channel, it is rescheduled (put at the tail of the process FIFO). Otherwise the event count is incremented. In either case, control returns to the caller.

• ENTRY CONDITIONS

```
AR1 = pointer to channel
```

• EXIT CONDITIONS.

```
AR0, AR1, R11 and ST are corrupted
All other registers are preserved
```

• EXAMPLE . . . . . .

```
.include “traps.inc”
; assume AR5 points to a parameter struct
; signal an event on SYNC_CHAN
lat PRHI_SIG
ldi *+ar5(SYNC_CHAN), ar1
nop                    ; or other useful instructions
nop                    ; executed before the LAT
```

• SEE ALSO. . . . . PRHI_WAIT

• SPECIAL NOTES . . This entry point must not by called from the low priority context, but it can be used by interrupt handlers.

38.15. PRHI_WAIT

• BRIEF . . . . . Consume an event from a signal channel, or deschedule

• CLASS. . . . . Process communication

• SYNOPSIS . . . . . LATcond PRHI_WAIT

```
PRHI_WAIT is defined in TRAPS.INC
```

• DESCRIPTION . . . If the event counter is not zero, it is decremented an control returns to the caller. If the event counter is zero, the calling process is swapped out and set to wait on the channel. It will be rescheduled by the next call to the PRHI_SIG service on the same channel.

• ENTRY CONDITIONS

```
AR1 = pointer to signal channel struct
AR3 = pointer to PCS of calling process
```

• EXIT CONDITIONS.

```
AR0, AR1, AR2, R11 and ST are corrupted
```

• EXAMPLE . . . . . .

```
.include “traps.inc”
; assume AR5 points to a parameter struct
; wait for event on SYNC_CHAN
lat PRHI_WAIT
ldi *+ar5(SYNC_CHAN), ar1
nop
nop
; the event has happened
```

• SEE ALSO. . . . . PRHI_SIG

• SPECIAL NOTES . . This service must not be called from the low priority context or by an isr.

38.16. PRLO_PSH

• BRIEF . . . . . This call is for internal use only, and is not exactly the equivalent of PRHI_PSH for the PRLO process. This call assumes that a PRHI process is waiting on the channel, and will crash the system if there isn't. PRLO_PUSH is used by the K_taskcall function to send command packets from a task to the microkernel process.

38.17. SETISR1

• BRIEF . . . . . Moves an ISR to level 1, setting the global interrupt enable bit

• CLASS. . . . . Interrupt service management

• SYNOPSYS . . . . . LATcond SETISR1

```
SETISR1 is defined in TRAPS.INC
```

• DESCRIPTION . . . This call increments the level 1 ISR counter, and returns with the global interrupt enable bit set. It should be used by ISRs that may take a longer time than allowed by the interrupt latency requirements of the application.

• ENTRY CONDITIONS

```
ST and R11 are saved on the stack
```

• EXIT CONDITIONS. .

```
The global interrupt enable bit in ST is set
R11 is corrupted
all other registers are preserved
```

• EXAMPLE . . . . . This ISR disables the timer0 interrupt while processing the IIOF3 interrupt with global interrupts enabled.

This is an example only; it is not normally necessary within Virtuoso to disable the timer interrupts within an ISR.

```
.include “traps.inc”
.def __iiof3_isr
.text
__iiof3_isr
push st
push r11
pushf r11
push ar0
push ar1
; ... acknowledge interrupt if necessary
lat SETISR1             ; move to ISR level 1
push ar2
push iie                ; save current IIE register
andn 0001h, iie         ; disable timer0 interrupt
; ... interrupts are enabled from this point
; ... process the IIOF3 interrupt
lat ENDISR1             ; terminate the ISR
```

```
                pop iie                 ; first restore the IIE register
                nop
                nop
```

• SEE ALSO. ENDISR1

38.18. SYSDIS

- BRIEF . . . . . Disable global interrupts
- CLASS. . . . . Interrupt service management
- SYNOPSYS . . . . . TRAPcond SYSDIS

  `SYSDIS is defined in TRAPS.INC`
- DESCRIPTION . . . This entry point provides a convenient and safe way to reset the GIE bit in the ST register.
- ENTRY CONDITIONS none
- EXIT CONDITIONS . The GIE bit in ST is reset.
- EXAMPLE . . . . . .

```
; ...
trap SYSDIS
; ... code to run with interrupts disabled
trap SYSENA
; ...
```

- SEE ALSO. . . . . SYSENA, SYSVEC
- SPECIAL NOTES . . This trap can be used inside a C function by using the inline assembly statement.

38.19. SYSENA

• BRIEF . . . . . Enable global interrupts

• CLASS . . . . . Interrupt service management

• SYNOPSYS . . . . . TRAPcond SYSENA

```
SYSENA is defined in TRAPS.INC
```

• DESCRIPTION . . . This entry point provides a convenient and safe way to set the GIE bit in the ST register.

• ENTRY CONDITIONS none

• EXIT CONDITIONS . The GIE bit in ST is set.

• EXAMPLE . . . . . .

```
; ...
trap SYSDIS
; ... code to run with interrupts disabled
trap SYSENA
; ...
```

• SEE ALSO . . . . . SYSDIS, SYSVEC

• SPECIAL NOTES . . This trap can be used inside a C function by using the inline assembly statement.

38.20. SYSVEC

• BRIEF . . . . . Install or remove an interrupt handler

• CLASS. . . . . Interrupt service management

• SYNOPSYS . . . . . TRAPcond SYSVEC

```
SYSVEC is defined in TRAPS.INC
```

• DESCRIPTION . . . This entry point provides a convenient and safe way to install, remove, or replace an interrupt handler. It takes two parameters: an interrupt number, and a pointer to an ISR. The pointer is entered into the interrupt vector table, and if it is not zero, the corresponding interrupt enable bit is set in the IIE or IIF register. If the pointer is NULL, the interrupt is disabled.

• ENTRY CONDITIONS

```
AR0 = interrupt number
AR1 = pointer to ISR or 0
```

• EXIT CONDITIONS.

```
AR0, AR1, AR2 and ST corrupted
IIE or IIF register modified as described
all other registers are preserved
```

• EXAMPLE . . . . . .

```
.include “traps.inc”
.ref __iiiof2_isr
; create a variable pointing to the isr
.bss __iiof2_isr_ptr, 1
.sect “.cinit”
.word 1, __iiof2_isr_ptr, __iiof2_isr
.text
; install the ISR
ldi 34, ar0
ldi @__iiof2_isr_ptr, ar1
trap SYSVEC
; __iiof2_isr is installed and enabled
```

• SEE ALSO. . . . . KS_EnableISR, KS_DisableISR

• SPECIAL NOTES . . This entry point is for assembly language programming only. The KS_EnableISR and KS_DisableISR services use this trap, and are C- call-

able.

- SPECIAL NOTES . . Interrupt numbers are:
  - 0..31 for interrupts enabled in the IIE register
  - 32..35 for IIOF0..IIOF3

38.21. YIELD

• BRIEF . . . . . Yield CPU to next nanokernel process

• CLASS. . . . . Process management

• SYNOPSIS . . . . . LATcond YIELD

```
YIELD is defined in TRAPS.INC
```

• DESCRIPTION . . . The calling process is swapped out and added to the tail of the process FIFO. The process at the head of the FIFO is swapped in. This may be the same process, if it was the only one ready to execute.

• ENTRY CONDITIONS

```
AR3 = pointer to PCS of calling process
```

• EXIT CONDITIONS.

```
AR0, AR1, AR2, R11 and ST are corrupted
```

• EXAMPLE . . . . . This example shows how to avoid a redundant YIELD operation, by testing the process FIFO

```
.include “nanok.inc”
.include “traps.inc”
;
ldi @NANOK_HEAD, r11    ; test head of process FIFO
nop                     ; avoid possible silicon bug
latnz YIELD             ; yield if there is another process
nop                     ; 3 nops or useful instructions
nop                     ; executed before the yield
nop
```

• SPECIAL NOTES . . This service must not be called from the low priority context or by an isr.

39. Predefined drivers

A number of devices drivers are provided as standard with the C40 release of the Virtuoso kernel. These are:

- the timer device drivers
- the netlink drivers
- the rawlink drivers
- the host interface device drivers

The default drivers are declared in iface.h and must be declared as a DRIVER object in the sysdef file in order to use them.

39.1. The timer device drivers

```
void Timer0_Driver (unit);
void Timer1_Driver (unit);
```

These drivers create an interface to the C40 on-chip timer peripherals. One one of them should be installed on each node.

They provide two services to the kernel:

1. The function timer_read () returns a 32 bit value incrementing at 1/4 of the CPU clock frequency (10 MHz for a 40 MHz C40). This permits very precise time measurements. It is used by the workload monitor, by the task level monitor to timestamp events, and by the KS_HighTimer () service.

2. Both drivers also generate EVENT 48, with a period determined by the ‘unit’ argument. This is used internally by the kernel to maintain the TICKS time. The TICKS time in turn is used for implementing time-outs, and by the KS_Elapse (), KS_LowTime and KS_Sleep () services.

The variable ‘tickunit’ is defined in the MAIN?.C files. This should be used for the ‘unit’ argument (see examples).

39.2. Host interface device drivers

One of these is required on the ROOT node if the host services are to be used (not for booting the network). The following are provided:

```
void HostLinkDma (inlink #, outlink #, prio); /* HLDMA.LIB */
```

DMA based driver for all boards using one or two C40 comports to imple-

ment the PC interface. The link number to use are:

- 0, 3 for Hema DSP1
- 0, 0 for Sang Megalink
- 3, 3 for Hunt Engineering HEPC2

The ‘prio’ parameter sets the relative priority of the CPU and the DMA engines when accessing memory. Use one of the symbolic constants #defined in IFACE.H

```
void HLxx_Driver (void); /* HLxx.LIB, xx = 03, 00 or 33 */
```

Interrupt driven PC to comports interface, not using DMA. You can use one of these in place of HostLinkDma () if you want to keep the DMA engines free for other purposes.

```
void DPCC40_Driver (void); /* DPCC40.LIB */
```

Host driver for the LSI DPCC40 board.

```
void VmeHostV (void); /* HYDRA1.LIB */
```

Host driver for the ARIEL HYDRA1 board.

```
void SP40Host (void); /* SP40.LIB */
```

Host driver for the SONITECH SPIRIT40 board.

39.3. Netlink drivers

Two drivers, both using the C40 comports, are provided:

```
void NetLinkDma (link, prio); /* NLDMA.LIB */
void Netlink(link); /* LINK.LIB */
```

The second parameter determines the relative priority of the CPU and the DMA engines. See fileIFACE.H for possible options.

39.4. Raw link drivers

```
RawLinkDma (link #, prio) : (dma.lib)
```

Raw communication port driver. This provides the KS_Linkin () and

```
KS_Link# ID LOCATION LBCR GBCR IACK FILE
```

39.5. Task Level Timings

Following is a list of task level timings of some of the kernel services provided by Virtuoso. These timings are the result of timing measurement on a TMS320C40 board with a clock speed of 40MHz and zero wait state program- and data-memory.

All timings are in microseconds. The C compiler used was TI C v.4.4.

Minimum Kernel call

```
Nop (1)                 9
```

Message transfer

```
Send/Receive with wait
Header only (2)         59
16 bytes (2)            62
128 bytes (2)           68
1024 bytes (2)          123
```

Queue operations

```
Enqueue 1 byte (1)      17
Dequeue 1 byte (1)      17
Enqueue 4 bytes (1)     18
Dequeue 4 bytes (1)     18
Enqueue/Dequeue (with wait)(2)56
```

Semaphore operations

```
Signal (1)              12
Signal/Wait (2)         46
Signal/WaitTimeout (2)  56
Signal/WaitMany (2)     64
Signal/WaitManyTimeout(2)73
```

Resources

```
Lock or Unlock (1)      12
```

Note, that one char is one 32-bit word on the TMS320C40.

(1): involves no context switch

(2): involves two context switches. Timing is round-trip time.

Glossary

Abort (task)	To halt a task's execution, and put it in a state where it can be started. A task in a waiting state should not be aborted.
API	Application Programming Interface
Class	A category of microkernel service.
Debugger	The task-level debugger used to inspect the state of microkernel objects. Not to be confused with the source-level debugger or emulator.
Driver	Low-level code used to interface to peripheral hardware. Often implemented using interrupt handlers and nanokernel processes.
Event	A synchronization signalled by an interrupt handler or nanokernel process to a waiting microkernel task.
FIFO	Queue
Group	A collection tasks.
Hard Real-time	A system subject to detailed and rigid timing constraints.
High Precision Timer	A clock that can be read with higher resolution than the microkernel timer, often provided by hardware.
Interrupt	An asynchronous event triggered by hardware outside the CPU, to which the processor has to respond.
ISR	Interrupt Service Routine. Code to respond to an interrupt
Level (programming)	An application programming interface (API). Lower levels offer reduced functionality and smaller overheads.
Link	Communication medium between processors (e.g., Comport on the C40)

Mailbox	A rendezvous point for microkernel message passing, to synchronize sending and receiving tasks. No data is stored or buffered in the mailbox.
Memory	A class of microkernel service. In addition to the standard system heap, real-time memory allocation of fixed size blocks is provided.
Message	Information used with a mailbox. Consists of the header structure, and the data to be communicated.
Microkernel	High-level API for real-time programming. Also refers to the system-provided process that implements microkernel services.
Move Data	Copy memory from any address on one processor to any address on another processor, without synchronizing with any other task.
/MP	Multi-processor. A product version that provides the full range of services within one processor, plus raw link drivers to communicate with other processors. Also appropriate for single-processor systems where the links may be used to communicate with peripherals.
Nanokernel	An multi-tasking API between the microkernel and interrupt handlers, providing simplified functionality for lower overheads.
Node	Processor containing a single CPU, with associated memory and peripherals. Each node runs a separate instance of the kernel.
Object	A structure on which microkernel services act. Objects are declared in the system definition file.
Priority	The integer value that controls the order of scheduling of microkernel tasks. A smaller value indicates a higher priority, and the valid range is 1 to 64.

Process	A thread of execution, with its own stack, using the nanokernel API. The difference between a process and an ISR is that the process may block, or wait, whereas the ISR must continue executing until it returns.
Queue	A structure providing buffered, asynchronous communication between microkernel tasks. Data is communicated in fixed size blocks. Items are dequeued (received) in the same order that they were enqueued (sent).
Resource	A microkernel object that is used to sequentialize operations of several tasks. When a tasks locks a resource, no other task may lock the same resource until the first task unlocks it.
Semaphore	A flexible means of synchronization of between tasks. A task signals a semaphore to wake up a task that may be waiting on the semaphore. Semaphores are counting, allowing several tasks to signal and wait on one semaphore.
/SP	Single-processor. A product version that provides the full range of services within one processor.
Suspend (task)	Temporarily halt the execution of a microkernel task, and prevent it from being rescheduled until it has been Resumed. A task in a waiting condition may be suspended.
System definition	Declaration of the target hardware, and of microkernel objects and device drivers. The specification of each object includes which node it is placed upon.
Task	A thread of execution, with its own stack, using the microkernel API.
Timer	A microkernel clock providing timing information to tasks and services. Time-outs and elapsed time can be used without declaring a separate timer. Timed semaphore signalling (one-shot or periodic) is provided by a timer declared in the system definition file.

Yield	A task or process voluntarily gives the CPU to other tasks or processes. In the microkernel, execution will only be passed to tasks of equal priority.
Virtuoso	A family of tools for real-time programming.
/VSP	Virtual Single Processor. A product version that provides the full range of services in a way that is transparently distributed over multiple processors.

Index

L

M

N

O

P

Q

R

S

T

U

V

W

Y