Virtual Channel Router Version 2.0 User Guide

PUMA:033/VCR2/USER Revision 1.3

Mark Debbage Mark Hill
Denis Nicole

Department of Electronics & Computer Science,
University of Southampton

© University of Southampton 28th October 1991

The Virtual Channel Router (VCR) software, the accompanying documentation and the virtual occam compiler are licensed free of charge with absolutely no warranty. Copyright of the VCR software and documentation is retained by the University of Southampton. Copyright of the virtual occam compiler is retained by Inmos. The make utility provided in the distribution is, to the best knowledge of the VCR authors, entirely public domain code.

Permission is granted to anyone to make or distribute verbatim copies of this document as received, in any medium, provided that the copyright notice and this permission notice are preserved, and that the distributor grants the recipient permission for further redistribution as permitted by this notice. Further copies of the VCR software system may be made and distributed at will, provided that it is distributed in full with no modifications and that no charge is made other than media costs. Parties wishing to distribute modified versions of the VCR package, including commercial products utilising VCR components, should contact the authors for further clarification. Permission will usually then be granted.

Since this software is licensed free of charge it is supplied without warranty of any kind, either expressed or implied. The entire risk as to the quality and performance of the VCR system is with the user.

Abstract

A software package has been developed within the Esprit P2701 $\mathcal{PUMA}$ project to provide unrestricted point-to-point communication across arbitrary networks of transputers. The implementation has been fully embedded within the occam language preserving occam syntax and semantics for communication. A static configuration level groups user processes onto appropriate virtual processors and allows inter-connection of these processes using static virtual channels. The mapping of virtual channels to the available communications resources and the mapping of virtual processors onto the available computing resources is performed at run-time by the kernel.

VCR version 2.0 forms an extension to the new Inmos C and occam Toolsets. The system supports arbitrary networks using virtual links to guarantee deadlock-freedom. Off-line tools are provided to generate the system's routing tables from a network specified by either a wiring diagram or an Inmos network checker description.

Contents

1 Introduction

occam programs are constructed from concurrent processes communicating by channels. The language itself imposes no restrictions on the number of channels connected to each process. This is fully implemented by the standard Inmos occam compiler [5] when generating programs for execution on a single transputer, but for transputer networks there is a limitation of 4 physical links per node. In a distributed implementation of *full* occam this restriction (among others) is removed. The topology of a general purpose machine becomes a detail invisible to the programmer and process placement incurs no semantic restrictions. The first step to the implementation of a full occam system is the provision of virtual channels and message routing on a transputer network [6]). The virtual channel router (VCR) implements such facilities in software for T-series transputer machines.

The VCR user guide is targeted at users familiar with transputers, occam and the Inmos Toolset. If the use of the C bindings is required, then familiarity with the C language and integrating occam and C code is assumed. The concepts involved in writing virtual channel applications are explained, with examples presented to show the ease of expression of virtual channel topologies. The compilation and execution of user code (including C modules) is covered with simple examples. Details of the construction of VCR for user networks are given, and techniques for handling devices discussed. Installation details for PC and Sun host and subsequent execution of the example programs are described in the *Getting Started* section.

This document does not cover the internal workings of the virtual channel router nor the extensions provided to support an implementation of distributed full occam (see [2]).

2 Motivation

One of the major restrictions imposed upon users of current-generation transputer networks is the limited valency at each transputer. Unless an application maps on to the network such that data is exchanged between nearest neighbours only (for example, a grid problem), then a routing kernel of some description must be employed. In many cases the user is forced to implement a custom-crafted message-passing harness specific to the application and topology.

This approach is undesirable in two respects. Firstly, the user program has been written for a particular topology and will be difficult to port to another machine: this dependence must be severed if we are to unify concurrent processing onto a variety of general purpose architectures. Secondly, the complexity of an efficient routing kernel is comparable to that required for many applications. This means that a considerable proportion of programming effort is expended in the writing of a kernel which should be a system facility. The temptation is to build the crudest implementation that seems to work, giving only reasonable levels of efficiency and little confidence in overall program correctness.

The intention of this work was to design and implement an efficient and provably deadlock-free router which preserves the occam semantics for communication between non-adjacent processors. This eliminates the valency restriction and allows the user to place as many channels between as many processes as required. Program portability is increased because the provision of virtual channels allows details of a parallel machine's topology to be hidden from the user. The target topology is internal to the routing kernel, so that the user's outer level occam conveys only the required process placements and the connections between them.

This work forms a software simulation of the virtual channel interface to be supported by networks of the Inmos T9000 processor and C104 routing chip. It thus allows T9000-style programming techniques to be employed on current generation hardware, easing the learning gap between the two technologies.

3 VCR Package Overview

VCR applications consists of occam separately compiled units (SCs) generated either directly from the Inmos occam Toolset (D4205, D5205, D6205, D7205) or by wrapping up code from the Inmos C Toolset (D4214, D5214, D6214, D7214) in an occam harness. The configuration file places SCs onto virtual processors with suitable parameter lists and connects the virtual processors together with virtual channels. Configuration generates a topology-independent configuration binary which is mapped onto the physical network at run-time. Application binaries are also independent of adjustments to system parameters, such as packet size or header format: binaries can now be taken to any other VCR installation and executed, provided any devices required by the application are available on the network. VCR directly supports only the new Transputer Common Object File Format (TCOFF) as generated by the more recent Inmos Toolsets. Existing occam and C code from previous Toolsets should be recompiled under the current system before use with VCR (refer to Section 4.8 for modifications to code required).

The VCR system consists of the following components:

- The virtual occam compiler/configurer `voc`, this is an Inmos-written occam compiler based on the sources of the product version of `oc`. It can be switched into a virtual configuration mode where it reads in a VCR configuration file and produces a topology-independent VCR output file.
- The VCR run-time consists of a loader, a routing kernel and the virtual channels implementation itself. The loader is provided in both dynamic and static versions : the former reads a network configuration file (`ncf`) at run-time and boots up the specified network, the latter requires compilation steps to create a bootable from the `ncf` file. The routing kernel is a deadlock-free packet router for arbitrary networks and arbitrary protocols. The top-layer of VCR implements the loading of applications, and performs the mapping of virtual processors, devices and channels to available resources. It also implements the library interface visible to user code.
- The VCR tools are C programs used to support the generation of VCR network configuration files and routing tables, from industry standard checker and wiring diagram formats. These are provided as transputer bootables, but may be recompiled for the host machine using a suitable C compiler.

Typical invocation of a VCR program is performed thus:

```
vcr topology program
```

`vcr` specifies the VCR run-time system, `topology` the target topology and `program` the user's application. Using dynamic loading each of these are decoupled: the interface between run-time and topology is the network configuration file (`ncf`), the interface between run-time and program is the applications virtual configuration file binary (`vcr`).

This release of VCR supports the full-range of Inmos 32 bit transputers. 16 bit devices (the T2 range) are *not* supported. There is additional support for devices compared to earlier releases, but *no* support for source-level debugging.

4 Getting Started

This section of the documentation gives an overview of the VCR package and its demands on the host system. At present, PC and Sun 4 hosts are directly supported through scripts and executables. Reconfiguration for other environments is relatively straightforward, requiring only the availability of a competent command shell environment and a host C compiler. Example command-line dialogues are given for PC users (only the obvious syntax differences are necessary for users of other machines).

4.1 Requirements

VCR forms an extension to the new TCOFF-based Inmos Toolsets. VCR applications in occam require prior installation of the occam Toolset. Mixed C and occam VCR applications, or recompilation of the VCR system and its tools, also require installation of the C Toolset.

This documentation assumes that the necessary Toolsets have been successfully installed on the host. In particular, the environment variables `TRANSPUTER` (if not the default), `ISEARCH`, `ITERM`, `IBOARDSIZE` and `IDEBUGSIZE` must have been already set up in accordance with the Inmos documentation. Paths must exist to the `iserver` and to the tools in the installed Toolset(s).

In addition, the VCR system relies on the (supplied) Inmos checker software. This package determines the connectivity and transputer details of the network attached to the host. Its output is processed to generate VCR configuration files. Installation of this system is recommended, although the use of wiring diagram network descriptions is a possible alternative for broadly homogeneous machines.

Recompilation of the VCR system, its tools or example programs is aided by the use of a UNIX-standard `make` utility. If such a utility is not available, VCR provides source and PC executable for a public domain `make` which is sufficiently UNIX-like to recompile the VCR system and its tools. Documentation for this can be found in Appendix B.

4.2 VCR Installation

The VCR software should be transferred onto the host filestore by following the directions supplied with the media distribution. This will result in a directory `VCR` containing subdirectories for VCR source, VCR libraries, VCR binaries, VCR networks, examples, and a directory to rebuild the system from.

Before installation any previous versions of VCR should be removed from the environment i.e. any references on the executable path, on `ISEARCH` or other environment variables should be deleted.

4.2.1 Sun 4 Hosts

Once the VCR system has been transferred into a suitable directory (`/usr/vcr` is assumed here), a number of environment variables need to be set up and the path extended. This typically involves modifying the `.login` or `.cshrc` file. Firstly, the location of the VCR installation needs to be declared for use by the VCR tools:

```
setenv VCR /usr/vcr
```

VCR's libraries need to be added onto the library search path by *prepending* `$VCR/libs/` to `ISEARCH`. The VCR libraries *must* appear earlier than the occam Toolset libraries, since the VCR virtual library must be found in preference to the occam Toolset version. A typical `ISEARCH` is thus:

```
setenv ISEARCH "/usr/vcr/libs/ /usr/ioctools/libs/ /usr/ictools/libs/"
```

Since user reconfiguration of the target VCR network is simple and arbitrary networks are now possible it is anticipated that users may want to generate their own networks as well as use those provided globally. To support this feature VCR uses a `NETSEARCH` environment variable to locate its target topologies, allowing the network files of multiple users to be stored in separate directories. The syntax for `NETSEARCH` is identical to that for `ISEARCH`. For example:

```
setenv NETSEARCH "/usr/vcr/nets/ /usr/md/mynets/"
```

The VCR system always searches the current working directory first for network files or bootables, and then each directory in the `NETSEARCH` list. If this variable is not defined then the directory `$VCR/libs/` is taken as a default.

The VCR tools should be installed onto the executables path by adding `$VCR/bin/sun4` to the path. It is worth checking that the executables do indeed have executable status, since this information may have been lost in the file transfer. A command such as:

`chmod +x $VCR/bin/sun4/*` will reassert this status if required.

These environment changes should be brought into effect by sourcing the appropriate command file.

4.2.2 PC Hosts

Once the VCR system has been transferred into a suitable directory (`\vcr` is assumed here), a number of environment variables need to be set up and the path extended. This typically involves modifying the `autoexec.bat` file. Firstly, the location of the VCR installation needs to be declared for use by the VCR tools:

```
set VCR=\vcr
```

VCR's libraries need to be added onto the library search path by *prepending* `\vcr\libs\` to `ISEARCH`. The VCR libraries *must* appear earlier than the occam Toolset libraries, since the VCR virtual library must be found in preference to the occam Toolset version.

A typical `ISEARCH` is thus:

```
set ISEARCH="\vcr\libs\ \ioctools\libs\ \ictools\libs\"
```

Since user reconfiguration of the target VCR network is simple and arbitrary networks are now possible it is anticipated that users may want to generate their own networks as well as use those provided globally. To support this feature VCR uses a `NETSEARCH` environment variable to locate its target topologies, allowing the network files of multiple users to be stored in separate directories. The syntax for `NETSEARCH` is identical to that for `ISEARCH`. For example:

```
set NETSEARCH="\vcr\nets\ \md\mynets\"
```

The VCR system always searches the current working directory first for network files or bootables, and then each directory in the `NETSEARCH` list. If this variable is not defined then the directory `\vcr\libs\` is taken as a default.

The VCR tools should be installed onto the executables path by adding `\vcr\bin\pc` to the path. These environment changes should be brought into effect by re-executing `autoexec.bat`.

4.2.3 Other Hosts

The procedure for installing VCR for other hosts is straight-forward. After transferring the system onto the filestore, a new directory off `$VCR/bin` should be created with a unique name identifying the host computer or operating system. For a full installation new scripts should be created to run the VCR transputer-based tools. These should be simple to derive from the Sun 4 versions (the PC uses driver programs rather than scripts due to limitations in MSDOS). Alternatively the C sources in `$VCR/source/tools` (see Section 7.4) may be compiled to run on the host processor.

In addition, the VCR driver program needs to be compiled up for the host. This is a single file C program, found in `$VCR/source/tools/vdriver.c`, which should compile with any Kernighan and Ritchie-standard C compiler. Two compilations are necessary to produce dynamic and static loading variants (see example below). If the host does not sport a UNIX-like make utility then the source for a public domain version is provided (see section B), although this may need some tailoring to the target operating system.

The host's environment variables should be set up as for the standard hosts but using the newly installed hostname to determine the executables path. For example, once the environment has been set up the VCR tools can be installed for a Sun 3 by:

```
cd $VCR/bin
cp -R sun4 sun3
cd sun3
chmod +x *
cc $VCR/source/tools/vdriver.c -DUNIX -o vdriver
cc $VCR/source/tools/vdriver.c -DUNIX -DSTATIC -o svdriver
```

4.2.4 Transputer Details

For single-user fixed-link machines, the supplied VCR driver software will be sufficient to load and run the system. For multi-user or reconfigurable-link machines, the `vcr` (dynamic kernel loading) and `svcr` (static kernel loading) scripts will require modification. These changes will typically arbitrate between users or set switches to match the required topology.

For example, the Southampton installation consists of a Sun4 server, a Parsys Supernode (accessed by `/dev/nap0`) and a single transputer site (accessed by `/dev/nap1`). The following two scripts are used for dynamic and static loading respectively:

`$VCR/bin/sun4/vcr`:

```
#!/bin/csh
if (-f $VCR/nets/$1.ncf) then
  if ($1 == "psingle") then
    setenv TRANSPUTER /dev/nap1
  else
    setenv TRANSPUTER /dev/nap0
    $SETSW $VCR/nets/$1.map
  endif
  vdriver $*
else
  echo Invalid topology specified, aborting ...
endif
```

`$VCR/bin/sun4/svcr`:

```
#!/bin/csh
if (-f $VCR/nets/$1.btl) then
  if ($1 == "psingle") then
    setenv TRANSPUTER /dev/nap1
  else
    setenv TRANSPUTER /dev/nap0
    $SETSW $VCR/nets/$1.map
  endif
  svdriver $*
else
  echo Invalid topology specified, aborting ...
endif
```

Where the environment variable `$SETSW` identifies the switch-setter binary.

4.3 Generating a VCR Topology

VCR topologies are described by a network configuration file (`.ncf`) as defined in Appendix E. Tools are provided to generate such files from the output of the Inmos check utility or from wiring diagrams. This section describes how these files can be generated and checked using the VCR system. All the tools work by command-line piping, an input text file is piped into the tool through `stdin` and an output file is produced on `stdout`. Progress reports and errors messages are produced on `stderr`. Since redirection is used, the booting root transputer message is typically disabled. Debug information can be turned on with the `/d` option, and normal information turned off with the `/s` option. A `/m` option details the processor renumbering that occurs when a wiring diagram is used.

VCR topologies are stored in networks directories which are referenced by the `NETSEARCH` path or, if `NETSEARCH` is undefined, by the default location, `$VCR\nets\`.

An `ncf` file describes the transputers in the topology (their type, memory size and connectivity) and the VCR routing tables required to deadlock-free route that topology. The most natural way to generate an `ncf` is by using the Inmos checker utility in conjunction with the associated memory sizer, and to feed the resulting information into the VCR routing generator program, `routegen`. The output of `routegen` should be redirected to an output file with a suitable topology name and a `.ncf` extension. For example, the `ncf` for the network `marks` can be generated by:

```
check | mtest | routegen >marks.ncf
```

If the memory size is the same on every processor then there is no need to perform the `mtest` as the memory size can be stated on the command line of `routegen`:

```
check | routegen 4M >marks.ncf
```

If a wiring diagram is used instead of check output, the command-line option `/w` is required. With a wiring diagram as input the first processor number referenced is assumed to be the root transputer. Since wiring diagram processor numbers are not necessarily contiguous they are mapped by `routegen` into integers from 0 (the root) upwards. Using the `/m` option will display the mappings chosen on `stderr`. The processor type, memory size and root-to-host link need to be specified on the command-line, the former two of these parameters are assumed to apply to every processor in the network (the default parameters are `T414`, `256K` and link `0`). For example a wiring diagram of `marks` topology consisting entirely of T800s with 4Mbytes of memory booting down link 0 of the root processor would be used as follows:

```
routegen -w 4M T800 0 <marks.map >marks.ncf
```

Since the network configuration file is ASCII text, anomalies in a broadly homogeneous network can be introduced by editing the file (the format is described in Appendix E).

The default VCR routing generator uses simple algorithms which are light on run-time and memory usage. Optimised iterative versions have been written outside of the $\mathcal{PUMA}$ project[7]. This software forms two parts, a static traffic balancer `router` and a virtualisation optimiser `virtual`. Use of these tools will result in a more even static network traffic distribution, and lower numbers of virtual links (giving lower memory usage by the routing system). These tools can be invoked by similar means to the above:

```
check | mtest | router | virtual >daves.ncf
```

Note that the intermediate file between router and virtual is *not* a network configuration file. The run-time and memory requirements of these two programs grows rapidly with network size, but only need a single off-line run per topology.

Re-assurance that a particular network configuration file is deadlock-free can be gained using the `ncfcheck` utility:

```
ncfcheck <marks.ncf
```

which produces suitable diagnostics on `stderr`.

4.4 Loading a VCR Topology

4.4.1 Dynamic Loading

Confidence in the successful generation of an `ncf` file can be gained by calling:

```
vcr topology
```

(this script may need site-specific modification to set the switches).

This uses the dynamic loader to boot the specified network with the routing kernel. Success will result in messages of the form:

```
Auto-loader, MD MBH 3/12/90
vcr : Virtual Channel Router, Version 2.0, 09:30:43 Jun 11 1991
M. Debbage, M. Hill, University Of Southampton, ESPRIT PUMA P2701

Usage: vcr <topology> [-ld] [-kd -kf] [-d -s] <program>

Loader Options:
 ld   Debug loader
Kernel Options:
 kd   Kernel debug
 kf   Go faster kernel (default secure)
VCR Options:
 d    debug
 s    silent
```

If any run-time error messages are produced they should be dealt with as appropriate. Failure to display the first line indicates that the root transputer is unable to communicate to the host over the link specified in the network configuration file. Failure to display the remainder would occur if the target transputer network did not match the connectivity in the network configuration file.

If any error flag becomes set on the network the easiest way to clear it is to run the `check` utility. This also gives an opportunity to check that the network is appropriate to the topology the user is attempting to execute over.

4.4.2 Static Loading

The VCR kernel can be installed on a network from a single bootable (this allows conventional source-level debugging of the routing kernel for example). This requires topology-specific compilation steps, but provides similar functionality to the dynamic version. In order to create a static load the system needs to be recompiled for that `topology`. This requires switching to the `build` directory and use of a make utility. For UNIX machines the standard make is ideal, for PCs the supplied public domain make (`pdmake`) is suitable:

```
cd \vcr\build
pdmake MODE=sload TARGET=topology
```

This creates the required network bootable in the default networks directory, and can be loaded by issuing:

```
svcr topology
```

(this script may need site-specific modification to set the switches).

Success will be rewarded by a similar banner to the dynamic version.

4.5 Compiling an Example

Source for a suite of examples exercising particular features of VCR are distributed in sub-directories off `examples` in the VCR installation. These are provided without makefiles or scripts as these can be generated automatically using the `vmakef` utility[1]. This should be invoked with the name of the `.cfg` file, but with a `.vcr` extension, which will generate a makefile of that name with a `.mak` extension. Since it is conventional to give makefiles the filename `makefile` the default output filename is typically over-ridden using the `vmakef` option `/o`. Hence, to produce the `makefile` for `examp0` on a PC:

```
cd \vcr\examples\examp0
vmakef example.vcr /o makefile
```

The make file produced is UNIX standard, and can be processed using `make` on UNIX machines or the supplied `pdmake` (or equivalent) on PCs. If scripts are required rather than makefiles, these can be generated using the (hidden) `vmakef` option `/b`:

[1]This is derived from the product occam Toolset sources of `imakef`, and contains the extra rules for building VCR applications

```
vmakef /b example.vcr
```

to generate a batch file `example.bat`. Using the `vmakef` option `/i` will give extra information during processing to expose the workings of the utility. Example 4 contains C code units and therefore needs the additional `vmakef` option `/c` to turn on the building of C units.

4.6 Running an Example

With an example compiled as directed above, it can be executed using the dynamic loader on a VCR `topology` by:

```
vcr topology example
```

or using the static loader by:

```
svcr topology example
```

The former requires a `topology.ncf` file to be on the `NETSEARCH` and the latter a `topology.btl`. If supernode switches must be set `topology.map` files will also be required.

4.7 Changes From VCR Version 1.8

The major changes between VCR versions 1.8 and 2.0 are listed here.

- Targeted at the new Inmos TCOFF-based Toolsets for C and occam .
- Arbitrary networks are supported by deadlock-free virtual links routing.
- Improved run-time library[3], `vcr.lib` with occam and C bindings.
- A stream library, `streamio.lib`, compiled for virtual communications is now provided called `vstream.lib`.
- Virtual communications can be called from both low and high priorities
- T800 2D block move functions correctly.
- Full virtual processor implementation.
- Improved device handling.
- No source-level debugging.

4.8 Updating existing VCR programs

This section covers the modifications required to existing VCR programs before recompiling with the new occam Toolset and running under VCR 2.0.

- Modify the `hosthook.lib` reference in the configuration file to `hosthook.tax` or `hosthook.t8x`.
- Add a `CHAN OF BOOL stopper:` at configuration level and pass one end as the third parameter to `hosthook`. The other end may be left dangling or may be used to terminate the `hosthook` process from an appropriate user process.
- Change any references to the `dynamic.lib` library to `vcr.lib`. We strongly recommend that the user refers to the RPC document[3] for details of the significant improvements to the user-interface.
- Existing VCR programs referencing scientific language code through the VCR 1.8 occam harness support need additional care. For C source the use of the new Inmos C Toolset is encouraged; the incorporation of such code units is thus covered in detail in Section 5.3. The support for non-TCOFF languages is limited to the Inmos Toolset command `icvlink` which converts LFF binaries into TCOFF binaries. Techniques for 3L languages (version 2 compilers for C, FORTRAN and Pascal) are briefly presented in Appendix G.

5 Virtual Channel Applications

Writing virtual channel applications imposes few new demands on programmers familiar with the Inmos Toolsets. Applications consist of a top-level configuration file instantiating processes on virtual processors, connecting them via virtual channels and passing in suitable parameter lists. This configuration language is a variant on the D705B occam Toolset/TDS standard and simplifies the expression of user process networks by eliminating channel placement, relaxing the four-valent limitation and allowing arbitrary channel array constructors. The code called from this configuration layer are the conventional separately compiler units (SCs) produced by the Toolset linker for occam code.

This document assumes a PC-hosted system. Only the obvious cosmetic modifications are required for other users (such as command-line option being introduced by / rather than -). The notation `$VCR` is used to indicate the location of the VCR installation. Command-line dialogues for compiling, linking, and configuring VCR applications are given in the text for reference, but it should be noted that use of the `vmakef` and `make` utilities is strongly recommended for applications. This allows the details of system building to be hidden from the user. It is usually quicker than scripts and helps prevent inconsistencies.

5.1 Virtual Channels

Traditional occam channels occur in only two varieties; hard and soft. Soft channels are used to implement internal processor communication and require a word of memory for state. The number of soft channels is bounded only by memory availability thus allowing arbitrarily complicated process topologies to be formulated. Hard channels are used to implement communication between neighbouring transputers and are bounded by the number of link engines fabricated on the transputer. This is typically four, giving only four bi-directional channels which severely limits the user topologies implementable on a network of transputers.

Under VCR, static virtual channels[2] are those channels declared at the configuration level with channel ends in different user processors. Channels declared within code units are implemented by the soft channel mechanism but may require (cheap) run-time checks to ascertain their nature.

In the virtual channel routing system a kernel process is given control of some (or all) of the links, and implements packet-routing across the network to give many virtual channels across the physical links. The interface to the router is via virtual communications procedures. These are invisibly called by the occam compiler when a user communication is (potentially) virtual. The ALT mechanism works without modification to the enabling sequence. VCR therefore retains the occam syntax for communications.

The occam model of communicating processes requires that a communication between input and output processes has an inherent synchronization. It is vital that a packet-routing system for transputers does not violate this constraint; doing so loses the attractions of the formal basis from which occam is derived. Also, most deadlock-free routing

[2]There are also dynamic virtual channels [3] declared within code units but these fall outside the scope of this document.

schemes require an acknowledgement to satisfy their eager reader assumption[3]. The virtual channel protocol therefore includes an acknowledgement on the first packet of the message. The implementation of the protocol permits messages of arbitrary length. In particular messages of zero-length are allowed and synchronise the participating processes without data transfer. This particular feature cannot be expressed in occam .

Hence, user code is traditional occam , with potential excursions into scientific languages, and obeys the same formal rules as the single processor implementation.

5.2 Coding occam Processes

The coding of static occam separately compiled units (SCs) under VCR remains unchanged since the router imposes no language changes to the user. However, VCR's remote procedure call, dynamic channels and low-level features can only be accessed through the VCR run-time library using procedure and function calls. Use of this library is documented elsewhere [3].

In contrast with previous implementations, virtual communications may be invoked from both low and high priorities. It should be noted however, that it is generally beneficial to communications performance for the majority of user code to run at low priority.

The Inmos *virtual* occam compiler (invoked by `voc`) is used as a direct replacement for the Toolset occam compiler `oc`, and compiles all occam source in VCR. References to the host library should be modified to access `vhostio.lib`, and references to the stream library to `vstream.lib`, so that host connections may be via virtual as well as soft channels. All other Toolset tools can be used as normal[4]. An existing Toolset process will thus need the following modifications:

- References to `hostio.lib` changed to `vhostio.lib`
- References to `streamio.lib` changed to `vstream.lib`
- Recompilation using the virtual occam compiler and re-linking to make an SC unit

For example, a process to repeatedly send a string to the host is:

```
#INCLUDE "hostio.inc"
#USE "vhostio.lib"

PROC user( VAL INT proc.id, CHAN OF SP fs, ts )
  WHILE TRUE
    so.write.string.nl(fs, ts, "Hello World !!...")
:
```

[3] Eager readership requires that a packet can only be inserted into the network if the destination is guaranteed to accept it within bounded time. Thus a virtual output must not be allowed to produce unbounded network traffic if its corresponding virtual input is inactive.

[4] The Toolset tools `icconf` and `icollect` are not applicable since the user no longer produces hard program configurations or transputer bootables.

This can be compiled for transputer type `TA` and error mode `h` into an object file `node.tah` by the command:

```
voc node.occ /ta /h /o node.tah
```

This needs to be linked with the VCR and `occam` run-time libraries for type `TA` by referencing the linker indirect file `vcra.lnk` (for `T8` code `vcr8.lnk` would be substituted). The link command to produce an SC units `node.cah` is thus:

```
ilink node.tah /ta /h /f vcra.lnk vhostio.lib convert.lib /o node.cah
```

This process can now be called from a configuration level with suitable parameters and be executed. If many such processes wanted to communicate with the host then a multiplexor could be used:

```
#INCLUDE "hostio.inc"
#USE "vhostio.lib"

PROC root(CHAN OF SP fs,ts,[]CHAN OF SP vfs, vts )
  [sp.max.packet.data.size]BYTE buffer :
  INT16 len :
  INT last.alt :
  INT num.chans, count :
  SEQ
    num.chans := SIZE vfs
    last.alt := -1
    count := 0
    WHILE TRUE
      SEQ
        -- Fairer than the average ALT
        ALT i = 0 FOR SIZE vfs
          VAL choice IS (i+count)\num.chans :
          vts[choice] ? len :: buffer
            SEQ
              IF
                last.alt <> choice
                  SEQ
                    so.write.nl(fs, ts)
                    so.write.string(fs, ts, "<")
                    so.write.int(fs,ts,choice,1)
                    so.write.string(fs,ts, "> : ")
                TRUE
                  SKIP
              ts     ! len :: buffer
              fs     ? len :: buffer
              vfs[choice] ! len :: buffer
              last.alt := choice
        count := count + 1
:
```

Alternatively a replicated `PAR` over `hosthook` can be used at the configuration level to perform the same function (See `/examples/examp1`).

An SC unit `root.cah` would be produced using the commands:

```
voc root.occ /ta /h /o root.tah
ilink root.tah /ta /h /f vcra.lnk vhostio.lib convert.lib /o root.cah
```

5.3 Coding Scientific Language Processes

All of the facilities of the virtual channels router are available to users of the Inmos C Toolset. Bindings for other languages will become available as Toolsets are released by Inmos. Details of wrapping alien language units into equivalent occam processes is discussed at length in [4] and [5], and will not be repeated here. It should be borne in mind that the interface between `icconf` and C code units is not relevant to VCR applications in C. In particular, the call `get_param` is not of interest to the VCR programmer. Instead, C routines can be called from within occam harnesses at the function or the main level.

Calling at the function level allows particular functions to be coded in the language of choice, and invokes the C code without any supporting run-time. The Inmos occam Toolset provides the `callc.lib` library to implement static and heap initialisation if this is required by the called C. Integration at this level requires no extra support from VCR. It should be noted that the VCR communications library requires no statics and no heap.

Calling at the main level allows a fully fledged C run-time to be invoked (in either full or reduced mode). The initialisation overhead in this mechanism dictates that this technique is generally used for long-lived C code, often code which is used for the life of the application, the advantage is that full access to the C run-time library is available. Support for this is via the occam Toolset `centry.lib` and the type 1, 2, and 3 procedural interfaces (analagous to the C support in previous VCR releases). The current Toolset imposes the restriction of one C reference through this mechanism per occam SC unit (this is to avoid multiple instances of the C main entry symbol).

5.3.1 Virtual Communications Library

Virtual communication calls from C can be exposed by including the definitions file `<virtual.h>` in the required C source. The calls are akin to the Inmos routines for local communication, but with the function names preceded by `V` and the channel parameter being a pointer to a virtual channel block (`VCB`) rather than a `Channel`. The C bindings are thus:

```
EXTERN void VChanIn      (VCB *c, void *message, int length) ;
EXTERN char VChanInChar  (VCB *c) ;
EXTERN int  VChanInInt   (VCB *c) ;
EXTERN void VChanOut     (VCB *c, void *message, int length) ;
EXTERN void VChanOutChar (VCB *c, char ch) ;
EXTERN void VChanOutInt  (VCB *c, int i) ;
```

Since zero-length messages are allowed the `length` parameter to `VChanIn` and `VChanOut` may be zero, and in this case the `message` parameter if often `NULL`. Such communications act so as to synchronise the processes without data transfer.

VCR virtual channels are referenced as pointers to the `VCB` data structure with the lowest bit set. Pointers with their lowest bit clear are conventional local channels (soft or hard) and result in the VCR communications library reverting to conventional transputer instructions. The VCR libraries can thus be used as direct replacements for the Inmos C routines, as local channels will be detected and handled accordingly. Retypes for the channel pointers will be needed to prevent cast errors. Automatic conversion of Inmos communications routines to the virtual library can be achieved by adding the following code:

```
#include <virtual.h>
#define ChanIn(c,m,l)    VChanIn ((VCB *)c,m,l)
#define ChanInChar(c)    VChanInChar ((VCB *)c)
#define ChanInInt(c)     VChanInInt ((VCB *)c)
#define ChanOut(c,m,l)   VChanOut ((VCB *)c,m,l)
#define ChanOutChar(c,b) VChanOutChar ((VCB *)c,b)
#define ChanOutInt(c,i)  VChanOutInt ((VCB *)c,i)
```

The direction of virtual channels is contained in their structure. Attempting to output on an input virtual channel, or to input an an output virtual channel, will generate a run-time error.

Alternation over *input* virtual channel ends can be achieved using the standard Inmos C calls. Alternation over output virtual channel ends is technically feasible, but has not been implemented. No corresponding calls to the communicate or fail routines are provided. As in the occam model both parties to a communication must know the length of the communication before committing. T9000-style variable length communications are *not* directly supported.

At least 50 words of workspace should be available on the stack when user applications call these routines.

5.3.2 Function Level Invocation

Calling C from occam at the function level within VCR has no further complications over the C Toolset mechanism. Consider an example C function to input an integer from a channel, compute its factorial and output the result to another channel. This could be coded as the file `factorial.c` containing:

```
#include <stdiored.h>
#include <process.h>
#include <virtual.h>

int factorial(int x)
{ if (x>0) return x * factorial(x-1) ;
  else     return 1 ; }

void fac (VCB *in, VCB *out)
{
  int number, result ;
  number = VChanInInt (in) ;
  result = factorial (number) ;
  VChanOutInt (out, result) ;
}
```

It can be simply called from an occam file `harness.occ` by:

```
#PRAGMA EXTERNAL "PROC fac (VAL INT gsb, CHAN OF INT in, out) = 200"
PROC occam (CHAN OF INT from.root, to.root)
  INT gsb, dummy :
  SEQ
    LOAD.INPUT.CHANNEL  (dummy, from.root)
    LOAD.OUTPUT.CHANNEL (dummy, to.root)
    WHILE TRUE
      fac (gsb, from.root, to.root)
:
```

The dummy occam accesses to the virtual channels are necessary to mark the direction of channels passed into the C. The building sequence for this is as follows:

```
icc factorial.c /ta /o factorial.tah
oc harness.occ /ta /h /o harness.tah
ilink harness.tah /f vcra.lnk factorial.tah /ta /h /o harness.cah
```

The resulting occam SC (`harness.cah`) can then be incorporated in VCR configurations as usual.

5.3.3 Type 3 Procedural Interface Invocation

The example below receives an integer on a channel, compute its factorial and outputs the result to another channel. This requires a type 3 interface since there are user channels but no host channels. The source, `factorial.c`, could be coded as follows:

```
#include <stdiored.h>
#include <process.h>
#include <virtual.h>

int factorial(int x)
{ if (x>0) return x * factorial(x-1) ;
  else     return 1 ; }

int main(int argc, char *argv[], char *envp[],
         Channel* in_vec[], int inlen,
         Channel* out_vec[], int outlen)
{
  int number, result ;
  while (1)
  {
    number = VChanInInt ((VCB *) in_vec[2]) ;
    result = factorial (number) ;
    VChanOutInt ((VCB *) out_vec[2], result) ;
  }
}
```

This needs to be embedded within an occam harness `harness.occ` which sizes the stack and heap requirements of the C code and marks the direction of the C channels:

```
PROC harness(CHAN OF INT from.root, to.root)
  PROC main (CHAN OF INT from.root, to.root)
    #USE "centry.lib"
    [10000]INT ws1 :
    [1]INT ws2 :
    [3]INT in, out :
    SEQ
      LOAD.INPUT.CHANNEL  (in[2], from.root)
      LOAD.OUTPUT.CHANNEL (out[2], to.root)
      PROC.ENTRY.RC (1, ws1, ws2, in, out)
  :
  main (from.root, to.root)
:
```

The building sequence for this (using the stand-alone C run-time library) is as follows:

```
icc factorial.c /ta /o factorial.tah
oc harness.occ /ta /h /o harness.tah
ilink harness.tah /f vcra.lnk centry.lib factorial.tah
                     libcred.lib /ta /h /o harness.cah
```

The resulting occam SC (`harness.cah`) can then be incorporated in VCR configurations as usual.

5.3.4 Modifications for Type 1 or 2 Procedural Interface

With these interfaces the full C run-time library is required, and `SP` channels need to be passed into the C. Since the C run-time library itself does not reference the VCR virtual library for communication, it is necessary to provide an `SP` buffer in the occam harness which connects the soft channels passed into the C to the virtual channels to the outside world. This can be accomplished using the `vhostio` routine `so.buffer`. The code for the C routine becomes:

```
#include <stdio.h>
#include <process.h>
#include <virtual.h>

int factorial(int x)
{ if (x>0) return x * factorial(x-1) ;
  else     return 1 ; }

int main(int argc, char *argv[], char *envp[],
         Channel* in_vec[], int inlen,
         Channel* out_vec[], int outlen)
{
  int number, result ;
  printf("C module\n") ;
  while (1)
  {
    number = VChanInInt ((VCB *) in_vec[2]) ;
    result = factorial (number) ;
    VChanOutInt ((VCB *) out_vec[2], result) ;
  }
}
```

The occam harness is recoded as:

```
#INCLUDE "hostio.inc"
#USE "vhostio.lib"
PROC harness(CHAN OF SP fs, ts, CHAN OF BOOL host.stopper,
                                CHAN OF INT from.root, to.root)
  PROC main (CHAN OF SP fs, ts, CHAN OF INT from.root, to.root)
    #USE "centry.lib"
    [10000]INT ws1 :
    [1]INT ws2 :
    [3]INT in, out :
    SEQ
      LOAD.INPUT.CHANNEL  (in[2], from.root)
      LOAD.OUTPUT.CHANNEL (out[2], to.root)
      PROC.ENTRY (fs, ts, 1, ws1, ws2, in, out)
```

```
  :
  CHAN OF SP main.to.server, server.to.main :
  CHAN OF BOOL stopper :
  SEQ
    so.write.string.nl(fs, ts, "occam module")
    PAR
      SEQ
        main (server.to.main, main.to.server, from.root, to.root)
        stopper ! TRUE
      so.buffer (fs, ts, main.to.server, server.to.main, stopper)
    host.stopper ! TRUE
:
```

The necessary compilation steps are now:

```
icc factorial.c /ta /o factorial.tah
oc harness.occ /ta /h /o harness.tah
ilink harness.tah /f vcra.lnk vhostio.lib centry.lib
    factorial.tah libc.lib /ta /h /o harness.cah
```

5.4 Writing Configuration Files

The virtual channels configuration language is very similar to that used for the old D705B Toolset or for TDS. The configuration declares the virtual processors required for the application, and the code associated with them. The code may be any valid occam process, bar library references, but is typically used to reference SC code units and bind parameters to them. Channels declared at the outermost layer are potentially virtual channels. There is no requirement to `PLACE` these channels to links, and no limit on node valency.

Each virtual channel may be passed into zero, one or two virtual processors. In the zero case the channel is not used and requires no run-time state. If it is passed into a single virtual processor and only used in one direction, it is a dangling channel and an attempt to communicate over it will deadlock as in the occam model. If it is passed into a single virtual processor and used by two processes, one in each direction, then the channel is implemented by a conventional soft channel. If it is passed into two virtual processors one end must be an input and the other an output, and the result is a virtual channel. If both ends of this virtual channel are mapped at run-time onto a single physical processor then the implementation is by a soft channel mechanism. Other cases will be detected as usage violations by the configurer.

5.4.1 The Host Connection

The channels used by the application for host communication must be declared to the routing system since the router shares the single host connection. This is achieved by passing the application's host channels into a VCR procedure (`hosthook`) placed on the root processor (a run-time error will be generated if it is invoked elsewhere). The procedural interface to this routine is:

```
hosthook (CHAN OF SP fs, ts, CHAN OF BOOL stopper)
```

By calling the routine with channels `fs` and `ts` (which may be soft or virtual), the user host channels are given access to host services. A boolean communication on the `stopper` channel will stop the routine, permitting the complete termination of VCR applications. The `stopper` can be left dangling if this functionality is not required. The routine is supplied in variants for type TA (`hosthook.cax`) and T8 (`hosthook.c8x`) root transputers.

Note that, in contrast with previous VCR versions, multiple `hosthook`s can be called on the root processor since the implementation is by an efficient queuing semaphore rather than a fixed size multiplexor. Hence applications need not introduce multiplexors of their own for this purpose.

5.4.2 Process Farm Example

A process farm of a host multiplexor and 10 users for the example processes given in Section 5.2 can be simply expressed by:

```
#INCLUDE "hostio.inc"
VAL num.nodes IS 10 :
[num.nodes] CHAN OF SP vfs, vts :
PLACED PAR
  PROCESSOR 0 TA
    #USE "hosthook.cax"
    #USE "root.cah"
    CHAN OF SP fs,ts :
    CHAN OF BOOL stopper :
    PAR
      root     (fs, ts, vfs, vts )
      hosthook (fs, ts, stopper)

  PLACED PAR i = 0 FOR num.nodes
    PROCESSOR (i+1) TA
      #USE "node.cah"
      user( i, vfs[i], vts[i] )
```

Note that even such a simple configuration is invalid in standard Toolset since it violates the four-valent node restriction. A user would be forced have to design a network and implement through-routing. Such an implementation would require substantially more code for the router than for the application making configuration an expensive operation.

5.4.3 Configuration Notes

The configuration level maps particular pieces of user code to specific processors. It is not necessary for the user to know how the processors in the network are labelled or even how many there are. If there are insufficient physical processors available the user code is mapped onto those present. Hence, the application and the target topology are completely independent : the application is compiled and configured without target knowledge, and the topology then bound in at run-time. In this way the application binaries are fully portable between VCR-fronted machines; provided functionality, such as a FPU or a graphics card, exploited by the application are available on the target hardware.

The actual mapping of the user's virtual processors to transputers is determined by a mapping function from the user ids to the network configuration file labelling. This is, by default, a modulo wrap-around over all processors bar the root. Device processors are handled separately through the mechanism described in Section 6; this mechanism can also be used to override the default mapping and thus allow optimising of processor placement without recompilation.

Configuration is achieved using the virtual occam configurer `vconf`[5]:

Configuring : `vconf <name>.cfg`

The configuration generates a VCR topology-independent configuration binary `<name>.vcr`. This contains information describing the virtual processor interconnect and configuration code. Code referenced from the configuration is not included in the file, only unlinked stubs to call such code with the required parameters. The resolution of these stubs are delayed until run-time.

A change from the D705 configuration language is that SC units must be referenced by **#USE**s at the same scope as the code that refers to them: they are not allowed at the outermost level. This is because the configurer now permits arbitrary occam at the code level making outer level references an unnecessary language addition.

Normally virtual channels are connected to one inputting process and one outputting process. In some configurations, *dangling* channels help the applications writer to reduce special-case coding, and these are therefore allowed by the configurer. A dangling channel is actually implemented as a soft channel and attempting to communicate over one will result in process deadlock as in the occam model. The configuration option `-j` displays channel allocation information to aid debugging.

5.4.4 Channel Arrays in Configuration Files

The configurer sometimes requires extra help from the programmer to work out how elements of channel arrays are passed into user code. The various rules are summarized here:

[5]This actually invokes the virtual occam compiler with the option `/m` to switch it into configuration mode.

- Channel array elements where the subscript does not include replicator indices may be passed into user code without restriction.
- Channel array elements where the subscript includes *processor* replicator indices may be passed into user code without restriction.
- Channel array elements where the subscript includes replicator indices *within a processor* require extra syntactic sugar.

All channel elements within an array slice must be used in the same direction by the processor it is passed to.

An example of this syntactic sugar is given below:

```
#INCLUDE "hostio.inc"

VAL num.nodes IS 10 :

[num.nodes] CHAN OF SP fs, ts :
[num.nodes] CHAN OF BOOL stopper :

PLACED PAR
  PROCESSOR 0 TA
    #USE "hosthook.cax"
    []CHAN OF SP fs IS fs :
    []CHAN OF SP ts IS ts :
    []CHAN OF BOOL stopper IS stopper :
    PAR i = 0 FOR num.nodes
      hosthook (fs[i], ts[i], stopper[i])

  PLACED PAR i = 0 FOR num.nodes
    PROCESSOR (i+1) TA
      #USE "node.cah"
      user( i, fs[i], ts[i], stopper[i] )
```

The call to `user` passes channel parameters where the subscript depends on a processor replicator index — this is allowed. However, the call to `hosthook` passes channel parameters where the subscript depends on a replicator index scoped within the processor. It is necessary to insert the (seemingly pointless) identity abbreviations as shown. These act so as to associate a slice of the channel array with the processor. The abbreviations thus impose the constraint that all channel elements within the array slice are used in the same direction — this same constraint is imposed by the procedural interface to `user` in the other case.

5.4.5 Channel Array Constructor

A language extension[6] is used to construct arbitrary array slices. The syntax for the array constructor is:

```
[ARRAY i = base FOR length OF exp]
```

The `base` and `length` expressions must be compile time constants, i.e. constant values or replicator indices. These constructors can be nested in the obvious fashion. The example below fully connects up `n` processes:

```
#INCLUDE "allall.inc"
#INCLUDE "hostio.inc"

[n][n] CHAN OF ANY v :
[n]    CHAN OF testing nodes.to.root :

PLACED PAR
  PROCESSOR 0 TA
    #USE "hosthook.cax"
    #USE "root.cah"
    CHAN OF SP fs, ts :
    CHAN OF BOOL stopper :
    PAR
      hosthook ( fs, ts, stopper)
      root     ( fs, ts, nodes.to.root, stopper)

  PLACED PAR i = 0 FOR n
    PROCESSOR (i+1) TA
      #USE "node.cah"
      horiz IS v[i] :
      vert  IS [ARRAY j = 0 FOR n OF v[j][i]] :
      node ( i, nodes.to.root[i], horiz, vert )
```

5.5 Termination

VCR version 2.0 encourages the correct termination of user code. When a user virtual processor completes, all of its allocated memory is returned to the run-time pool. Termination of the last virtual processor will free any memory used on the root for the application load and shut down the `iserver` in a controlled manner.

The `hosthook` routine can be terminated by a boolean communication on its `stopper` channel. If multiple `hosthook`s exist they will all need to be stopped if graceful application termination is to occur.

[6] This extension forms part of an Inmos proposal for occam 3 [1].

The previous, asynchronous, method of shutting the `iserver` down by a call to `so.exit` can still be used if preferred.

5.6 Applications Compilation Using vmakef

The details of compiling, linking and configuring applications can be hidden from the programmer by using `vmakef` to produce makefiles (and/or scripts). The use of this tool is highly recommended since it reduces the demands of Toolset programming, gives more efficient (and/or secure) compilations than batch files, and is host independent.

This utility has been derived from the product occam Toolset sources of `imakef`, the modifications made to the source are distributed in UNIX-standard `diff` files in the release. The reader is referred to the Inmos Toolset documentation for details of the conventional operation of `imakef`, only the extensions in functionality relevant to VCR applications are discussed here. It should be noted that the new occam Toolset permits only one linking operation per SC unit: partial or repeated linking is no longer supported.

As with `imakef`, `vmakef` uses filename extensions to work out the commands and options needed to generate the object modules of the application. It is therefore imperative that the standard extensions are used and not the Toolset defaults. VCR introduces two extra extensions, `.cfg` for the application configuration and `.vcr` for the application binary, the latter being generated from the former using the virtual configurer. `vmakef` thus contains the extra rules to backtrack from a `.vcr` to a `.cfg` and then to the user SCs.

`vmakef` is invoked with the name of the target file to build (for example `program.vcr`) and, by default, generates an output makefile `program.mak`. Here, we show the default output filename being changed to the more conventional name `makefile` using the `/o` option.

Using the `vmakef` option `/i` will give extra information during processing to expose the workings of the utility. It is recommended that until familiarity is reached with the tool, this option should be used and the information checked to make source that the utility references *all* of the expected source filenames.

The `vmakef` utility does not create make rules for the hosthook as this is part of the VCR system, not the user code.

5.6.1 occam applications

For a VCR application with configuration file `program.cfg` and written entirely in occam `vmakef` can be invoked as:

```
vmakef program.vcr /o makefile
```

to generate a UNIX standard `makefile`. The application can then be made using `make` on UNIX machines or the supplied `pdmake` (or equivalent) on PCs. If scripts are required rather than makefiles, these can be generated using the (hidden) `vmakef` option `/b`:

```
vmakef /b example.vcr
```

to generate a batch file `example.bat`.

5.6.2 Mixed occam and C applications

If the application contains C modules then `vmakef` requires the extra option `/c` in order to generate the necessary C modules. In this mode `vmakef` requires the user to generate a `.lnk` for each occam unit calling a C routine. This file needs to contain a one-per-line list of *all* units which need to be linked together to form the occam linked object file (the VCR library is typically referenced by a `#INCLUDE`, see example below). This list should include the C unit and any other units or libraries referenced by it[7]. In particular the referenced C unit object filename must conform to the standard `imakef` extension conventions. For example, to force compilation of the code fragments in Section 5.3.4 using the halting error mode and transputer type `TA`, a file `harness.lnk` should be created as listed below:

```
harness.tah
vhostio.lib
factorial.tah
centry.lib
libc.lib
#INCLUDE vcra.lnk
```

The `/b` option can again be used to generate scripts.

5.7 Loading and Executing Code

A VCR application `program` can be dynamically loaded and executed on a network called `topology` by the command:

```
vcr topology [/ld] [/kd /kf]  [/s /d] program
```

Note that no extensions are given on either the program or topology name. Giving a `/s` option (either before or after the `program`) turns off all VCR-generated screen output bar error messages (which are produced on `stderr`).

The `/ kf` option copies the code and workspace of the routing kernel into internal memory and turns off all error-checking normally performed while through-routing. This reduces the processor consumption on intermediate nodes quite dramatically, but should only be introduced once confidence in the correctness of the application code has been asserted.

[7] A trick in generating these lists is to run the `vmakef` utility first *without* the `/c` option. In this mode it will generate partial linker indirect files referencing all of the units needed to resolve the occam bar C references. This can then be copied to a `.lnk` extension, edited to add the missing C modules and the utility re-run but *with* the `/c` option.

The debug options `ld`, `kd` and `d`, turns on loader, kernel or VCR debug information respectively. Note that loader options must occur before kernel options which in turn must occur before VCR options. This is because the system has been developed in a highly modular fashion with each layer retaining the potential to accept command line parameters.

If a reconfigurable network is used then the switches must be set to match the wiring diagram of the desired topology by modifying the `vcr` script.

The users virtual processor numbers are mapped onto the network computing resources at run-time allowing multiple virtual processors to execute on each physical processor. The user program and target topology are thus totally independent until the run-time binding.

This loading is dynamic in two respects:

1. The loader places the routing kernel onto each processor in a network according to a description of that network taken from a `.ncf` file. This is accomplished by single node bootables and hence the VCR kernel code is entirely topology independent (thus no compilation steps are required to reconfigure VCR for a new network). The kernel loader can be reverted to a static version by compiling a topology dependent VCR kernel (see Section 7.1) and issuing:

   ```
   svcr topology [/ld] [/kd /kf] [/s | /d]
   ```

   Again, if a reconfigurable network is used then the switches must be set to match the wiring diagram of the desired topology by modifying the `svcr` script.

2. The loading of the application itself is dynamic: user code is not bound into the run-time kernel. This means that disk requirements and program build times are reduced compared to an equivalent static load. More importantly it means that applications binaries are topology independent. However, the standard Toolset debugger cannot be used at the source-level with such a scheme. Unlike previous versions of VCR there is no provision for reverting to a static load of user code: there is thus no support for source-level debugging. Typically, user applications should be developed and proven on a single transputer environment before configuration onto a network.

5.8 Examples

The standard VCR release contains a number of examples aimed at illustrating particularly interesting features of the system. Each example is provided as source only, makefiles or scripts can be generated using the `vmakef` utility.

- Example 0 configures a process farm in order to calculate a times table.
- Example 1 allows many processes to communicate with the host using virtual channels.
- Example 2 demonstrates all to all process connection and uses the resulting message traffic to gauge the processing requirement of packet routing.

- Example 3 times message latencies for 1, 2 or 3 hops distances. Note that the processor numbers have been selected for a particular cube network and may give misleading results on other topologies.
- Example 4 demonstrates the free mixing of C and `occam` under VCR by computing factorials using the various calling techniques. The formulation of the `.lnk` files and the ease of compilation with `vmakef` are of particular interest (note that `vmakef`, when executed, will require the `/c` option). The building of this example is dependent on prior installation of the Inmos C Toolset.
- Example 5 demonstrates a simple remote procedure call used to perform a matrix multiplication.
- Example 6 demonstrates a distributed recursive algorithm used to compute the sine of an array of real numbers. The source of this example is set up for a T800 network.
- Example 7 demonstrates the dynamic creation of virtual topologies.
- Example 8 times remote procedure call latencies for various distances and parameter block sizes.

Examples 5 to 8 use dynamic aspects of VCR [3] not discussed in this user guide. Example 9 has been deleted since source-level debugging is no longer supported.

5.9 Hints And Tips

The efficiency of applications running under the virtual channel system can be dramatically improved by following a few simple rules.

- Communications of a few large messages is obviously preferable to those of many small messages. In particular, a protocol such as

  ```
  PROTOCOL record IS BYTE; INT; INT; INT; INT; INT :: []BYTE :
  ```

  would compile to seven virtual communications, each one requiring an end-to-end acknowledgement. Grouping of the `INTs` into a single array would reduce the number of communications and improve efficiency. From C the information is best transmitted by building an appropriate `struct`.
- Algorithms should be designed to incorporate excess parallelism to hide the message latency.
- By activating virtual inputs in advance of their complementary virtual outputs the end-to-end acknowledgement latency can be hidden. This is due to VCR's pre-acknowledgement scheme.
- Placement of processors to shorten virtual communications path lengths should be considered as a final optimization step, not one of functionality, and can be achieved using the topology specific mapping mechanism discussed in Section 6.

6 Device Support and Processor Remapping

When a VCR program is executed on a specific topology the virtual processors declared during configuration, or to which any remote procedure call are made, are mapped to the physical processors according to the following defaults:

- Virtual processor 0 maps to physical processor 0.
- Virtual processors with ids greater than zero wrap around the physical processors with ids greater than zero in modulo fashion.

6.1 Performing remappings

Obviously the above scheme may not always be appropriate. Sometimes the user may wish to vary the mapping according to the topology of the network upon which the program is to be executed. This can be achieved by adding `<topology>.dvc` files to the directory from which the VCR program is executed or, if more universally applicable, to a directory on the `NETSEARCH` path. The first entry of a `<topology>.dvc` is the number of records in the file. Each record resides on a single line and consists of three fields followed by a free format string. The three fields are virtual processor id, the physical processor it maps to and a flag which is true (1) if implicitly mapped virtual processors may be placed on that physical processor.

For example, if an application performs most communications between processor ids' only differing by one, it might be sensible to ensure that the processors are remapped into a chain which snakes around the physical processors available.

Remappings are also useful to segregate mixed networks into virtual processor sets with or without floating point units, for example.

Some users may prefer to utilise a number other than zero for their root processor, this can also be achieved by adding/modifying an appropriate `<topology>.dvc` file.

The `.dvc` file format is described in Appendix F.

6.2 Installing Devices

Devices can be connected onto a VCR network in a number of ways depending on whether they are VCR processors or raw devices, and whether they are long or short lived. The recommended technique is to run VCR on the device; this makes devices easier too add and automatically allows the user to exploit the bandwidth of every device link connected to the network. Raw devices may have to be used when memory is tight or when source code is not available.

6.2.1 VCR Devices

A VCR device differs from a raw device in that it runs the VCR kernel and thus has access to global communications. The key problem is that the processor identity of the device will vary according to the network topology on which the application is run. This variation can be eliminated by selecting a negative processor id from those given in the `devices.inc` file which is appropriate to the device to be connected. For example the `inmos.B007.graphics.card` is, by convention, processor -100. Installing the device is just a matter of adding to the `<topology>.dvc` file a mapping from the chosen id to the id given of the device given in the `<topology>.ncf` file. This correct id may be difficult to identify; if the `.ncf` file was generated from a wiring diagram a `-m` option can be used while executing `routegen` to produce a list of mappings on the *stderr* stream. The wrap flag, third field, will usually be switched off (0) so that no other user code gets mapped to the device.

The device code can then be declared at the configuration level using the identifier chosen from the `devices.inc` file, i.e. `inmos.B007.graphics.card`, to select the appropriate processor, or can be RPCed to that processor from any other in the network. Any extra information given in the `<topology>.dvc` may be obtained at run-time using the following procedures:

`char* DeviceInfo(int id)`
: C function returning a pointer to the extra information given after the three mapping fields of a device record. It returns a `NULL` string if the device is not available on the network.

`PROC DEVICE.INFO( VAL INT id, []BYTE info, INT size )`
: occam procedure which copies as much of the extra information as possible into the array `info` and returns with `size` set to the number of bytes used. It returns a negative `size` if the device does not exist.

6.2.2 Raw Devices

Any links in a VCR network which are not used by the kernel (i.e. links which have `x` entries in the `<topology>.ncf` file) can be connected to *raw devices*. Two variants are possible:

- *A long-lived device which is already booted when VCR is invoked.* The device server can itself be considered as a device and mapped to a special processor, for example `inmos.B007.graphics.server`, in the same way as in the previous section. The link down which the raw device resides should be recorded in the free format section of the device record. A device record should appear in the `.dvc` file of each topology to which the device is connected.
- *A short-lived device which needs to be booted at the same time as VCR.* Installation is as above but in addition the user code must fire the device bootable down the raw link. The device code can discover its boot link by `ALT`ing over each hard link when it first attempts external communication.

Note that in both these cases the user may prefer not to exclude the placement of workers on these processors; this is achieved by asserting (1) the wrap flag in the device record.

6.2.3 An Example

The following `.dvc` is used at Southampton to identify the devices connected to an 8 by 8 processor grid. The `.ncf` processor ids were obtained using the `-m` option on `routegen`. This option displays the mapping of processors in the `.map` file to those in the `.ncf` file.

```
5
-200  17  0         parsys graphics master
-201  26  0         parsys graphics slave
-204   2  1  link 0 to graphics master
-207  35  0         parsys memory server
-208  27  1  link 0 to memory server
```

Note that the negative processor numbers are selected from those given in the `devices.inc` file which should be included into the configuration level of any program which is to use devices.

6.2.4 The `devices.inc` file

The following device identities are those made with the current release of VCR. They should be adhered to if applications are to execute on installations at other institutions without reconfiguration.

The naming convention used in the file is that entries ending in `.card` are transputers with special functions (such as graphics or disk connections), while those ending in `.server` are standard network processors which happen to be adjacent to a processor with that special function.

```
-- @(#)    Title : List of device processor numbers
-- @(#) Filename : devices.inc
-- @(#)  Version : 1.4
-- @(#)     Date : 6/19/91

--{{{  generic devices
VAL graphics.card          IS -1 :
VAL graphics.server        IS -2 :
VAL graphics.one.card      IS -3 :
VAL graphics.one.server    IS -4 :
VAL graphics.two.card      IS -5 :
VAL graphics.two.server    IS -6 :
VAL graphics.three.card    IS -7 :
VAL graphics.three.server IS -8 :
VAL disk.card              IS -9 :
```

```
VAL disk.server          IS -10 :
VAL memory.card          IS -11 :
VAL memory.server        IS -12 :
VAL ethernet.card        IS -13 :
VAL ethernet.server      IS -14 :
--}}}
--{{{  inmos devices
VAL inmos.B007.graphics.card     IS -100 :
VAL inmos.B007.graphics.server   IS -101 :
VAL inmos.ethernet.card          IS -102 :
VAL inmos.ethernet.server        IS -103 :
--}}}
--{{{  parsys devices
VAL parsys.graphics.master.card  IS -200 :
VAL parsys.graphics.slave0.card  IS -201 :
VAL parsys.graphics.slave1.card  IS -202 :
VAL parsys.graphics.slave2.card  IS -203 :
VAL parsys.graphics.server       IS -204 :
VAL parsys.memory.card           IS -205 :
VAL parsys.memory.server         IS -206 :
VAL parsys.disk.card             IS -207 :
VAL parsys.disk.server           IS -208 :
VAL parsys.ethernet.card         IS -209 :
VAL parsys.ethernet.server       IS -210 :
--}}}
--{{{  meiko devices
VAL meiko.MK015.graphics.card         IS -300 :
VAL meiko.MK015.graphics.server       IS -301 :
VAL meiko.MK015.graphics.one.card     IS -302 :
VAL meiko.MK015.graphics.one.server   IS -303 :
VAL meiko.MK015.graphics.two.card     IS -304 :
VAL meiko.MK015.graphics.two.server   IS -305 :
VAL meiko.MK015.graphics.three.card   IS -306 :
VAL meiko.MK015.graphics.three.server IS -307 :
VAL meiko.MK052.graphics.card         IS -308 :
VAL meiko.MK052.graphics.server       IS -309 :
VAL meiko.MK052.graphics.one.card     IS -310 :
VAL meiko.MK052.graphics.one.server   IS -311 :
VAL meiko.MK052.graphics.two.card     IS -312 :
VAL meiko.MK052.graphics.two.server   IS -313 :
VAL meiko.MK052.graphics.three.card   IS -314 :
VAL meiko.MK052.graphics.three.server IS -315 :
VAL meiko.MK021.SCSI.driver.card      IS -316 :
VAL meiko.MK021.SCSI.driver.card      IS -317 :
VAL meiko.frame.grabber.card          IS -318 :
VAL meiko.fram.grabber.server         IS -319 :
--}}}
```

7 Recompiling the System

7.1 VCR Dynamic/Static System

A make file is provided in the `build` directory off the base of the VCR installation. Rebuilding of the system is performed by switching to this directory and typing one of:

`make MODE=dload`
: This recompiles any out of date binaries associated with the dynamic loader. Note that as `dload` is the default the setting of `MODE` is unnecessary.

`make MODE=sload TARGET=<topology>`
: This creates a topology specific VCR system bootable and places it in the `nets` directory. This bootable is executed using the `svcr` command detailed in Section 5.7.

Note that to prevent any inconsistencies developing between the `makefile` and any linker indirect files, the linker indirect files are generated, when appropriate, by the `makefile`. This is a trivial action for UNIX systems but causes problems on PCs because of the limitation on command line lengths. To get around this problem a public domain make utility is distributed with this release which has been modified to allow any make environment variable to be written to a given file. PC users will have to install and use this make utility if they wish to recompile system code units.

7.2 System Parameters

There are several system parameters that the user may wish to alter:

Name	Default	Location	Description
`UPR_MAX_PKT_SIZE`	160	`upri.h`	The maximum packet size in bytes.
`UPR_MAX_NUM_PORTS`	2500	`upri.h`	The maximum number of virtual channel ends on a processor.
`UPR_buffers_per_vlink`	10	`router.h`	Number of buffers per virtual
`INIT_CODE_BUFFER`	100000	`vcr.h`	Initial size (in bytes) of the code cache.

Lowering `UPR_buffers_per_vlink` trades a loss in efficiency against a reduction in memory required for buffering. We strongly recommend that the number should not be reduced below two. The number of virtual links per physical link depends upon the routing tables generated for a specific topology.

These parameters can be changed by modifying the relevant C header files in the `libs` directory and the system recompiled. Note that user applications are entirely independent of these parameters: no recompilation of application code is necessary.

7.3 Modifying the header format

The format of the packet header has been abstracted out to the file `header.h` in the `libs` directory. The default data structure is as follows:

8 bits to store processor number, thus limiting network to up to 256 processors.

8 bits to store actual packet lengths, thus limiting `UPR_MAX_PKT_SIZE` to 255 bytes with this particular scheme.

16 bits to store port number, thus limiting `UPR_MAX_NUM_PORTS` to 65536 ports per processor for this header format.

This default gives a good compromise between header size, encoding/decoding efficiency and field limits. An alternative scheme using 16 bits integers for all fields is distributed in `header16.h`, while a link protocol identical to that used in earlier VCR versions resides in `head_1v8.h`.

Note that applications are independent of header format: no recompilation of user code is necessary.

7.4 Tools

Only transputer binaries with supporting scripts/`EXE`s are distributed in the release. However a `makefile` is provided in `/bin/sun4` which will generate SPARC binaries for these tools. In addition a `makefile` is provided in `\bin\pc` for regenerating the `EXE`s used to invoke the transputer binaries. This technique is preferable to using batch files in four ways:

- I/O can be redirected on the command line of an `EXE`, this is not possible using a batch file to call the `iserver`.
- Environment variables can be read and used to build an appropriate `iserver` command line.
- Batch files cannot be nested in other batch files.
- If the `iserver` command built is too long this can be flagged. This is preferable to the command line truncation that would occur in a batch file.

8 System Performance

This section describes the message latency, communications bandwidth, processing efficiency and memory requirements of VCR. No attempt is made to compare the harness with kernels such as Tiny, Helios, Express and ECCL but we are confident that independent benchmarking would produce favourable results.

The figures in this section were measured using 25 MHz T800 transputers. The links were run at 20 MBit/s with overlapped acknowledges and through Supernode switches. The boards had 4K of internal single-cycle memory and 4M of external 5-cycle memory.

8.1 Messaging Performance

In any packetization scheme there is always a trade-off between latency and bandwidth regulated by the choice of packet size. With a small packet size message pipelining hides the number of hops to the destination, but at the cost of many through-routing switches (requiring many processor cycles) and degrading attainable throughput. A single choice of packet size could never be optimal for all applications; we have tried to achieve a sensible compromise.

The relatively slow degradation of switch-cost with packet size is due to the cycle-stealing DMA engines used to implement memory-to-link transfers. This behaviour causes the achievable bandwidth to be highly dependent on packet size. However, using very large packet sizes has a disastrous effect on end-to-end latency for multi-hop communications and increases buffering requirements on intermediate nodes. Figures 1 and 2 show the end-to-end message latency for 1,2 and 3 hops under VCR Version 2.0 with the system in fast and secure mode respectively. The graphs show a zero length message overhead of about 105 μs. These graph results were measured using the example in `examples/examp3` with and without the fast kernel option (`/kf`).

The release is distributed with a default packet size of 160 bytes. This was considered a reasonable choice for a general-purpose T800-based machine.

8.2 Memory Requirements

The VCR dynamic loader assumes a target network of generic 32-bit transputers with at least 256K of memory present — after loading, the system heap on each processor is automatically extended to that declared in the network configuration file. The static loader makes no such assumptions about target network but will fail to configure if target memory is insufficient.

The VCR kernel on the root processor requires approximately 300 KBytes of memory (excluding routing buffers). Of this total, nearly 100 KBytes is system code, 10 KBytes is system stack and statics, 100 KBytes is the initial user code buffer and the remainder is system heap usage.

The VCR kernel on the remaining processors uses less than 100 KBytes of memory

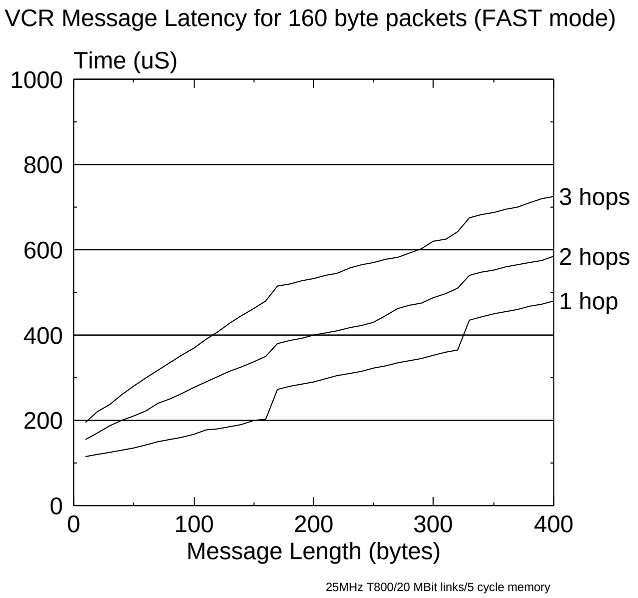

Fig. 1: Message Latency in fast kernel mode

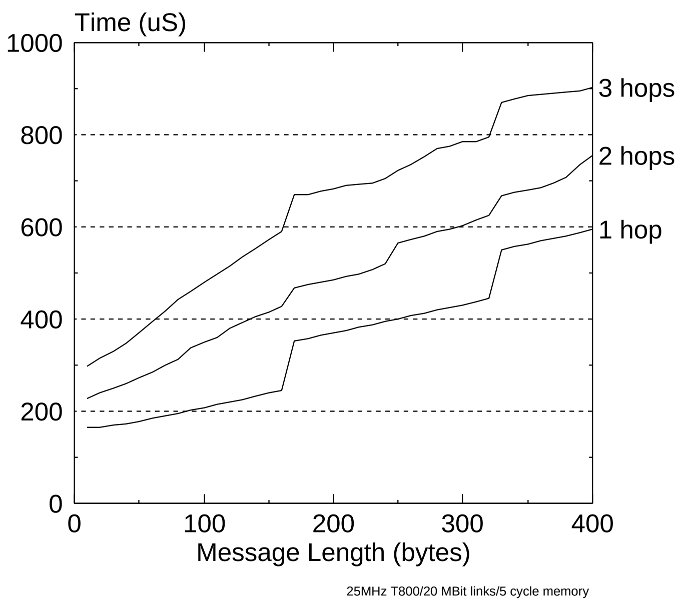

Fig. 2: Message Latency in default secure mode

(excluding routing buffers).

The memory requirement of routing buffers per node is given by the product of the number of active physical links (worst case is 4), the number of virtual links (depends on target topology), the number of buffers per virtual link (system constant `UPR_buffers_per_vlink`, default is 10) and the packet size (system constant `UPR_MAX_PKT_SIZE`, default is 160 bytes). For the `routegen tool`, the number of virtual links required to deadlock-free route a network is equal to the diameter (d) of its topology. This gives a worst case routing buffers requirement of $6400d$ bytes. For the optimized tools (`router` and `virtual`) a factor of 2 improvement can be expected.

For example, an 8 by 8 torus (with the root inserted in one arc) will have a diameter of 9 and a `routegen`-generated routing buffering requirement of less than 60KBytes on each node.

All memory not used by the VCR kernel is set up as a system heap to allocate user processes and code. This can be accessed directly by the application through the occam and C run-time support library.

9 Acknowledgements

The authors would like to thank the following:

- Inmos Limited, in particular Andy Sturges, for providing a compiler for virtual channels and designing a suitable interface for virtual communication at code and configuration levels.
- Ed Zaluska for his advice and support during the project.
- Dave Pritchard for suggestions on networks and routing strategies, and the provision of optimised route selection and virtualisation tools.
- Mike Moore for the design and construction of C^3PO, a 256 transputer machine used for testing and benchmarking of the routing kernel.

Bibliography

1 Barrett, G. The development of occam: types, classes and sharing. In *Proceedings of the 13th Occam Users Group* (1990), IOS Press.

2 Debbage, M., Hill, M. B., and Nicole, D. A. Towards a distributed implementation of occam. In *Proceedings of the 13th Occam Users Group* (1990), IOS Press.

3 Debbage, M., Hill, M. B., and Nicole, D. A. VCR 2.0 rpc support and run-time library. Technical notes, University Of Southampton, June 1991.

4 Inmos Ltd. *INMOS C Toolset User Manual.* 1000 Aztec West, Almondsbury, Bristol BS12 4SQ, UK, November 1990.

5 Inmos Ltd. *New occam2 Toolset Manual.* 1000 Aztec West, Almondsbury, Bristol BS12 4SQ, UK, Oct. 1990.

6 May, D. Implementing full occam. Esprit 2701 PUMA working paper 2, Inmos Limited, 1989.

7 Pritchard, D. J., Debbage, M., Hill, M., and Nicole, D. A. Optimised network configurer user guide. In preparation at University of Southampton, Nov. 1990.

8 Sun Microsystems. *Programming Utilities and Libraries*, unix version 4.0 ed. 2550 Garcia Avenue, Mountain View, CA, May 1988.

A Tools

The tools required to process checker files and/or wiring diagrams and then to execute user code upon these networks are documented in this section. They can be regenerated using the `makefile` provided in the appropriate architecture subdirectory off `bin`.

The synopses of command lines given in this Appendix are for MSDOS; UNIX users should substitute `-` for `/` on any command line switches that occur.

A.1 **ncf2inc** — Inmos C configurer includes generator

SYNOPSIS

```
ncf2inc [/s /d] <network.ncf >network.inc
```

DISTRIBUTION FILES

`$VCR\bin\tp\ncf2inc.btl`	Transputer version of the tool
`$VCR\bin\pc\ncf2inc.exe`	PC driver for the tool
`$VCR\bin\sun4\ncf2inc`	UNIX scripts to drive the tool
`$VCR\source\tools\ncf2inc.c`	ANSI C source

DESCRIPTION

The generation of single system bootables for arbitrary networks uses a generic Inmos C Configurer `.cfs` file (`/source/sloader/static.cfs`) and the generation of a `.inc` configurer includes file to tailor the configuration to a specific network.

This tool performs a textual manipulation of the data contained in an `.ncf` file, presenting it an a form acceptable to the Inmos configurer. Note that this tool never needs to be called explicitly since it is invoked, when appropriate, by the system `makefile`.

The `.ncf` file format is described in Appendix E, that of the `.inc` file in Appendix **??**.

OPTIONS

s turns on silent mode, thus preventing diagnostics.
d turns on (potentially voluminous) debug information.

SOURCE

The source is ANSI C and has been tested with Inmos `icc`, GNU `gcc` and Zortech C.

A.2 **ncfcheck** — routing table checker

SYNOPSIS
`ncfcheck [/d /s] <network.ncf`

DISTRIBUTION FILES

`$VCR\bin\tp\ncfcheck.btl`	Transputer version of the tool
`$VCR\bin\pc\ncfcheck.exe`	PC driver for the tool
`$VCR\bin\sun4\ncfcheck`	UNIX script to drive the tool
`$VCR\source\tools\ncfcheck.c`	ANSI C source

DESCRIPTION
As some improvement is possible in the route allocation made by `routegen` it seemed sensible to have a tool for confirming that `ncf` files generated by third parties produce provably deadlock free routing tables.

This tool performs that function. It accepts an `ncf` file as input and then proceeds to simulate packet transits between all node pairs recording link usage and the dependencies that develop between links in the network. Once every possible combination has been tried the channel dependency graph is analysed for any cycles: their existence would indicate a potential for deadlock.

The tool is also quite useful for checking that an `ncf` file has a valid format.

OPTIONS
s turns on silent mode, thus preventing diagnostics.
d turns on (potentially voluminous) debug information.

MEMORY REQUIREMENTS
The route generator uses dynamic memory to store network information allowing any practical network to be routed provided that memory is sufficient. The worst case memory requirements of the program grows as the square of the number of nodes in the network (n), and can be approximated by $2n^2$ bytes.

SOURCE
The source is ANSI C and has been tested with Inmos `icc`, GNU `gcc` and Zortech C.

LIMITATIONS
The depth of link virtualisation is limited to 255 levels since this information is stored in bytes. It is difficult to envisage a network in which this would be a problem.

A.3 **routegen** — routing tables generator

SYNOPSIS

```
routegen [/d /s /r] [#M | #K]  <network.chk > network.ncf
routegen /w [/m /r] [#M | #K] [Tname] [#] <network.map > network.ncf
```

DISTRIBUTION FILES

`$VCR\bin\tp\routegen.btl`	Transputer version of the tool
`$VCR\bin\pc\routegen.exe`	PC driver for the tool
`$VCR\bin\sun4\routegen`	UNIX script to drive the tool
`$VCR\source\tools\routegen.c`	ANSI C source

DESCRIPTION

The routing generator tool, `routegen`, reads in a network description file (in checker or wiring diagram formats) on `stdin` and produces a VCR network configuration file on `stdout`. Diagnostics information is produced on `stderr`. The options to `routegen` and their default settings are:

`/d`	Debug mode	FALSE
`/s`	Silent mode	FALSE
`/w`	Wiring diagram mode	FALSE
`/m`	Generate mappings	FALSE
`/r`	Recognise grids	FALSE
`#[M|K]`	Memory size	256K
`Tname`	Transputer type	T414
`#`	Boot link	0

If an environmental variable `ROUTEGEN_ARG` is set then its value is parsed as a command-line string and the default values adjusted. This facility can be used to provide sensible defaults thus simplifying the subsequent use of this command. In order to allow TRUE options to be overriden, negating options can be formed by adding the character 'n' between the option introducer and the option itself. Thus `/nd` clears the debug flag.

Checker input mode is the default operation of the utility. The checker file (format defined in Appendix C) will normally have memory information present. If it is not present then a memory size must be specified in either kilobytes (`K` suffix) or megabytes (`M` suffix).

Wiring diagram input mode is activated by the `/w` switch. Since a wiring diagram (format defined in Appendix D) has insufficient information to produce a network configuration file, extra information must be passed in. This information comprises the boot link (a single digit in the range 0 to 3), the memory size in either kilobytes (`K` suffix) or megabytes (`M` suffix) and the transputer part name (string beginning with `T`). The tool assumes an homogeneous network of the same transputer type and memory sizes. If this is not appropriate hand-crafting of the output file is necessary. With a wiring diagram the first processor number referenced is assumed to be the root transputer.

The option `/r` makes `routegen` attempt to match the input network to a grid or torus topology. Only uniform meshes are matched, that is each slice must be of a fixed size (in each dimension) and aligned with each other. For grids the root processor can be attached to any edge of the grid, and for torii one link in the network can be split in two and a root processor inserted. Upon a successful match the topology details are encoded into a comment filed in the output file format. Grid topologies are routed using a deadlock-free scheme, whilst torii are routed using a minimal number of virtual links. Although this information enables programming environments to map process meshes onto processors sensibly, VCR itself makes no use of this possibility.

The output numbering scheme of `routegen` may not be identical to the input numbering. This can occur either when the wiring diagram numbers are non-contiguous or when the topology is matched to a grid. Using the `/m` option will display the mappings chosen on `stderr`.

Transputer versions of the tools should be invoked with the `iserver` booting message turned off, since screen output is directed to a file.

MEMORY REQUIREMENTS
The route generator uses dynamic memory to store network information allowing any practical network to be routed provided that memory is sufficient. The memory requirements of the program grows as the square of the number of nodes in the network (n), and can be approximated by $7n^2$ bytes. Thus even for a 256 processor network this requirement is less than half of one megabyte.

SOURCE
The source is ANSI C and has been tested with Inmos `icc`, GNU `gcc` and Zortech C.

LIMITATIONS
The diameter of the input network is limited to 255 since distance information is stored in bytes.

Iterative or simulated annealing solution to route selection can improve link balancing. The virtual links strategy for arbitrary networks uses structured buffer pools. It is therefore simple but non-optimal: buffers are segregated into more classes than necessary and the frequency of local acknowledgements is higher, degrading communication bandwidth and increasing buffer requirements. These issues are addressed by the `router` and `virtual` tools (See Appendix A.4).

grid, ring, torus — wiring diagram generators

SYNOPSIS
`[grid | ring | torus] nx ny`

DISTRIBUTION FILES

```
$VCR\source\tools\mesh2map.c
$VCR\bin\pc\grid.exe         $VCR\bin\pc\ring.exe   $VCR\bin\pc\torus.exe
$VCR\bin\sun4\grid.s4        $VCR\bin\sun4\ring.s4  $VCR\bin\sun4\torus.s4
```

DESCRIPTION
These programs generate 2-dimensional mesh wiring diagrams on the standard output. They are included in the release as an aid to producing VCR networks for mesh topologies. The parameter `nx` sets the number of processors in the x dimension, and `ny` the number of processors in the y dimension. Grids are open with no looping connections, rings have

loops in the x dimension only, and torii have loops in both dimensions.

LIMITATIONS
No root processor is included in the output, so these must be added manually. Standard `news` style labelling is assumed; that is link 0s connect to link 3s, and link 1s to link 2s.

A.4 **router** and **virtual** — Optimising network configurer

SYNOPSIS

```
router [/m#] [/d /s] [#M | #K]  <network.chk | virtual [/s] > network.ncf
router [/m#] [/d /s] [#M | #K] /w [Tname] [#] <network.map | virtual [/s] > network.ncf
```

DISTRIBUTION FILES

`$VCR\bin\tp\router.btl`	Transputer version of the tool
`$VCR\bin\pc\router.exe`	PC driver for the tool
`$VCR\bin\sun4\router`	UNIX scripts to drive the tool
`$VCR\source\tools\router.c`	ANSI C source
`$VCR\bin\tp\virtual.btl`	Transputer version of the tool
`$VCR\bin\pc\virtual.exe`	PC driver for the tool
`$VCR\bin\sun4\virtual`	UNIX scripts to drive the tool
`$VCR\source\tools\virtual.c`	ANSI C source

DESCRIPTION
These routines have been written by Dave Pritchard at Southampton to improve upon the routing tables generated by `routegen`. The `router` process attempts to balance the link traffic by careful choice of the available shortest paths between processors. The `virtual` process analyses the channel dependency graph of the routing tables generated by `router`. Any cycles discovered are broken by inserting appropriate link virtualisations and the new dependency graph analysed again. Observation suggests that the improvements made by `router` are slight, however the reductions in the degree of link virtualisation made by `virtual` can be quite significant.

More details are available in the *Optimised Network Configurer User Guide*[7].

OPTIONS
`virtual` has a silent option.
`router` has the same options as `routegen`, bar the `/m` and `/r` options. Additionally it takes an argument which dictates the number of iterations of the load-balancing algorithm (0 - No re-runs, 1 - Re-run once, 2 - Re-run until overall link utilisation is unchanged). Environment variables cannot be used to set default options.

MEMORY REQUIREMENTS
Scales badly (See [7]).

LIMITATIONS
Slow and scales badly (and the network diameter must be less than fifteen). For further details see [7]).

A.5 **svcr** - A script for static kernel loading

SYNOPSIS

`svcr <topology> [/ld] [/kd /kf] [/d /s] <program>`

DISTRIBUTION FILES

`$VCR\bin\pc\svcr.bat`	Batch file for program execution
`$VCR\bin\sun4\svcr`	UNIX script for program execution
`$VCR\bin\pc\svdriver.exe`	Builds `iserver` command on PC
`$VCR\bin\sun4\svdriver`	Builds `iserver` command on sun4
`$VCR\source\tools\vdriver.c`	Source of `vdriver`

DESCRIPTION

The static program execution script, `svcr`, calls `svdriver` which, in turn, builds an `iserver` command line appropriate to the execution of the user program. The script is provided as a suitable point to perform any installation specific actions, such as setting the switches on a supernode or modifying environment variables.

Note that static refers to the way the VCR system is loaded onto the network: the application is still loaded dynamically and remains totally independent of system code.

OPTIONS

Loader options

`/ld` – Loader debug

Kernel options

`/kd` – Kernel debug

`/kf` – Fast kernel (using internal RAM and reduced error checking)

VCR options

`/d` – VCR debug

`/s` – silent VCR (no loading messages)

SOURCE

The source of `svdriver` is the same as that of `vdriver`. Differences are introduced using a command-line macro during compilation.

LIMITATIONS

There is a potential problem on PCs of the `iserver` command built by `vdriver` being too long (maximum is 127 characters). If this occurs an exception is generated.

A.6 **vcr** - A script for dynamic kernel loading

SYNOPSIS
`vcr <topology> [/ld] [/kd /kf] [/d /s] <program>`

DISTRIBUTION FILES

`$VCR\bin\pc\vcr.bat`	Batch file for program execution
`$VCR\bin\sun4\vcr`	UNIX script for program execution
`$VCR\bin\pc\vdriver.exe`	Builds `iserver` command on PC
`$VCR\bin\sun4\vdriver`	Builds `iserver` command on sun4
`$VCR\source\tools\vdriver.c`	Source of `vdriver`

DESCRIPTION
The dynamic program execution script, `vcr`, calls `vdriver` which, in turn, builds an `iserver` command line appropriate to the execution of the user program. The script is provided as a suitable point to perform any installation specific actions, such as setting the switches on a supernode or modifying environment variables.

The dynamic loader requires that all processors in the network are 32 bit transputers and have a minimum of 256K of memory.

OPTIONS
Loader options
- `/ld` – Loader debug

Kernel options
- `/kd` – Kernel debug
- `/kf` – Fast kernel (using internal RAM and reduced error trapping)

VCR options
- `/d` – VCR debug
- `/s` – silent VCR (no loading messages)

LIMITATIONS
There is a potential problem on PCs of the `iserver` command built by `vdriver` being too long (maximum is 127 characters). If this occurs an exception is generated.

A.7 **vmakef** – VCR makefile generator

SYNOPSIS

`vmakef <target> [/i] [/o <filename>] [/y] [/d] [/c] [/r] [/ni] [/b]`

DISTRIBUTION

`source\vmakef`	Contains differences between `vmakef` and `imakef` product sources
`bin\tp\vmakef.btl`	Transputer bootable
`bin\pc\vmakef.exe`	PC driver for the tool
`bin\sun4\vmakef`	Script to drive the tool

DESCRIPTION

The `vmakef` tool has been developed from the `imakef` sources distributed with the current occam Toolset. As well as retaining the entire functionality of `imakef`, `vmakef` allows a `.vcr` file to be given as the makefile `target`. The makefile generated, with `.mak` extension by default, includes all compilation stages required to generate a `.vcr` file from its component configuration and program code sources. All the examples in the release need only be `vmakef`ed and then made to generate the appropriate binaries.

If the makefile to be generated already exists then only the rules below the **`### IMAKEF CUT ####`** comment will be modified. This allows make file dependencies to be updated without losing any modifications made to environment variables.

Note that the rules for the generation of `hosthook` binaries from source are suppressed by `vmakef` by an explicit test for this name; thus users should not use this filename for their own code units.

OPTIONS

i displays information during processing
o allows the make file filename to be given on the command line
y introduces compilation options to disable interactive debugging
d introduces compilation options to disbale generation of debug information
c enable the building of C programs (see Section 5.6.2 for details)
r enables the deletion of intermediate files
ni prevents inclusion of files on ISEARCH
b generates a script, with a `.bat` extension, rather than a makefile

B The pdmake Utility

pdmake is a public domain make utility which implements a subset of the features of the UNIX-standard make utility [8]. It is supplied with VCR for those machines where competent make utilities are not readily available. Compilation of the VCR system itself requires a make utility. For PCs the pdmake has been modified to overcome deficiencies in the command shell implementation, making this the only PC make utility which will recompile the VCR system as distributed.

When used in conjunction with vmakef the compilation of VCR applications is dramatically simplified. The user no longer has to be concerned with the necessary commands to produce consistent build sequences, and need not specify endless option lists — this is all hidden in the makefile.

The original sources for pdmake were acquired from the public domain software library at the Lancaster University Computer Centre (uk.ac.lancs.pdsoft). This comprised files f01.msg, f01pdmk.boo and f01pdmk.doc from directory /micros/ibmpc/f01. We are indebted to Neil Russell, Jwahar Bammi and Ton van Overbeek for their previous work on this software.

Since the pdmake utility has little associated documentation, the user is referred to the documentation of the UNIX version. To summarize, the command-line interface is:

```
make [-f makefile] [-inpqrst] [target(s) ...]
```

where the options are,

f to supply makefile name other than defaults of makefile or Makefile.

i to ignore exit status of called commands.

n to not execute any commands, only pretend to make.

p to print all macros and targets.

q to question the up-to-dateness of the target. 0 is returned if up-to-date, 1 otherwise.

r to disable in-built rules.

s to perform silently.

t to touch files instead of making them.

The modifications we have made to the source of the utility to recompile the VCR system are listed below.

- Command-line environment variables now take precedence over internal macros.
- The time-stamping has been fixed for machines were the local time is not necessarily the same as the filestore time.

- White space between the end of macros and an ensuing comment is now removed.
- Adapted for Zortech C++ version 2.18 (used in a traditional ANSI C mode).
- Added the custom internal make function `ECHO.MACRO`. A command of the form,

  ```
  ECHO.MACRO $ENV > FILE
  ```

  is intercepted by the make utility and creates a `FILE` which contains the value of the macro `$ENV`, with each white space separated record in `$ENV` given a separate line. This feature sidesteps the PCs limited command line length and allows `.lnk` files to be produced from long macros. This modification is a real hack but justified in the simplifications it makes to compilations of the VCR system on a PC.

This utility is not part of the VCR but is merely provided for the benefit of VCR users. We make no claims about the correctness or portability of this software.

C Checker File Format

The output of the public domain network checker software is used as an example network description language for the network loader.

It is generated by executing `check`, piping the output to `mtest` and redirecting this to a file; conventionally with a `.chk` extension. The `check`er explores the network and discovers its connectivity, while the memory tester checks for sticky bits and then appends the memory size to the record of each processor.

C.1 File format

The file is headed by a one line comment detailing the checker version used to generate the file. The second line is a banner to identify the fields of the processor records. The following lines contain the records for each processor, one per line.

C.2 Record format

```
a bbbbc-dd e.ee f [   ggg     hhh    iii    jjj ] mK,n {+mK,n} ;
```

`a` - Processor id

`b` - Processor type

`c` - Processor version

`d` - Processor clock speed in MHz

`e` - Speed of boot link in Mbits/s

`f` - Boot link number

`g`, `h`, `i`, `j` - Link connection details for links 0,1,2,3:

> `...` - Link unconnected
> `x:y` - Connected to processor x link y
> `HOST` - Connected to HOST.

`m` - Size of memory region in Kilobytes

`n` - Number of cycles required to access memory region

C.3 Checker output for a cube plus root

```
check 1.5 | mtest 1.5
   # Part rate Mb Bt [  Link0  Link1  Link2  Link3 ]
   0 T800b-25 0.00 0 [   HOST    ...    ...    1:0 ] 4K,1+4048K,5;
   1 T800b-25 0.00 0 [    0:3    2:2    5:1    3:0 ] 4K,1+4048K,5;
```

```
2 T800b-25 0.00 2 [   ...    6:2    1:1    4:0 ] 4K,1+4048K,5;
3 T800b-25 0.00 0 [   1:3    4:2    7:1    ... ] 4K,1+4048K,5;
4 T800b-25 0.00 0 [   2:3    8:2    3:1    ... ] 4K,1+4048K,5;
5 T800b-25 0.00 1 [   ...    1:2    6:1    7:0 ] 4K,1+4048K,5;
6 T800b-25 0.00 2 [   ...    5:2    2:1    8:0 ] 4K,1+4048K,5;
7 T800b-25 0.00 0 [   5:3    3:2    8:1    ... ] 4K,1+4048K,5;
8 T800b-25 0.00 0 [   6:3    7:2    4:1    ... ] 4K,1+4048K,5;
```

D Wiring Diagram File Format

The contents of a wiring diagram file is ignored up to the point that the words `Wiring Diagram` appear in the text. From that point the file should contain entries of the following format:

```
Connect processor SP link SL to processor DP link SL
```

For each link connection between processors, where `SP` is the source processor number, `SL` is the source link number, `DP` is the destination processor number and `DL` is the destination link number.

The first processor number mentioned is, by convention, the root processor.

A wiring diagram for a cube plus root is:

```
Wiring Diagram
Connect processor 0 link 3 to processor 1 link 0
Connect processor 1 link 1 to processor 2 link 2
Connect processor 1 link 2 to processor 5 link 1
Connect processor 1 link 3 to processor 3 link 0
Connect processor 2 link 1 to processor 6 link 2
Connect processor 2 link 3 to processor 4 link 0
Connect processor 3 link 1 to processor 4 link 2
Connect processor 3 link 2 to processor 7 link 1
Connect processor 4 link 1 to processor 8 link 2
Connect processor 5 link 2 to processor 6 link 1
Connect processor 5 link 3 to processor 7 link 0
Connect processor 6 link 3 to processor 8 link 0
Connect processor 7 link 2 to processor 8 link 1
```

A single processor wiring diagram would be just:

```
Wiring Diagram
```

E Network Configuration File Format

These files typically have an extension `.ncf` and contain the information required to boot and maintain a universal packet router on any network.

The network configuration can be split into three stages:

E.1 The Network Record

This record contains just two elements, the number of nodes in the network, including the root, and the diameter of the network; the largest number of hops that any message will ever have to travel.

E.2 Node Records

There is a node record for each processor in the network. Note that the records are listed so that a record of a child processor never occurs before that of its parent.

The record format is as follows:

```
a
bbbbc-dd
mK
g h i j
```

`a` - Processor id is `a`.

`bbbbc-dd` - Processor type (`bbbb`) with optional version(`c`) and clock speed(`dd`).

`m` - Total size of memory in kilobytes.

`g`, `h`, `i`, `j` - Description records for links 0,1,2 and 3.

- `x` - Link unconnected
- `h` - Link connected to host
- `n`p:l - Link connected to processor p link l not on boot path
- `i`p:l - Processor boots from processor p link l on this link
- `o`p:l - Processor boots processor p via link l on this link

E.3 Routing Records

There is a routing record for every processor in the network. These may occur in any order.

Each record begins with the processor number, followed by a number indicating the number of entries in the routing table for that processor. The routing table follows and is an array of bytes. For each destination, starting with processor 0, there is a routing duplet consisting of the outgoing physical link number (or 4 for internal communications) and the virtual link `level` used to guarantee deadlock-freedom on arbitrary networks.

Note that the processor ids must be contiguous.

E.4 Example file for a cube plus root

```
9 4

0   T800b-25   4052K   x     h     x     o1:0
1   T800b-25   4052K   i0:3  o2:2  o5:1  o3:0
2   T800b-25   4052K   x     o6:2  i1:1  o4:0
3   T800b-25   4052K   i1:3  n4:2  o7:1  x
5   T800b-25   4052K   x     i1:2  n6:1  n7:0
4   T800b-25   4052K   i2:3  o8:2  n3:1  x
6   T800b-25   4052K   x     n5:2  i2:1  n8:0
7   T800b-25   4052K   n5:3  i3:2  n8:1  x
8   T800b-25   4052K   n6:3  n7:2  i4:1  x

0 18    4 0 3 0 3 1 3 1 3 2 3 1 3 2 3 2 3 3
1 18    0 0 4 0 1 0 3 0 1 1 2 0 2 1 3 1 1 2
2 18    2 1 2 0 4 0 3 1 3 0 1 1 1 0 1 2 1 1
3 18    0 1 0 0 1 1 4 0 1 0 2 1 1 2 2 0 1 1
4 18    0 2 0 1 0 0 2 0 4 0 2 2 1 1 1 1 1 0
5 18    1 1 1 0 2 1 3 1 1 2 4 0 2 0 3 0 3 1
6 18    1 2 1 1 2 0 3 2 3 1 1 0 4 0 1 1 3 0
7 18    0 2 1 1 2 2 1 0 2 1 0 0 0 1 4 0 2 0
8 18    0 3 2 2 0 1 1 1 2 0 1 1 0 0 1 0 4 0
```

F Device file format (.dvc)

Any topology, as defined by a `<topology>.ncf`, may have a `<topology>.dvc` file associated with it to allow special processors to be ear-marked and the placement of worker processors to be modified. The format of the device file is:

```
<no_of_records>
<record one>
<record two>
......
<record no_of_records>
```

Each record occupies a single line of the file and has the following format:

```
<user_id>  <id_in_ncf>  <wrap_flag> <extra_information> NEWLINE
```

The fields in each record are:

`user_id` is negative if the processor is to be labelled as special. An appropriate number should be chosen from those given in `devices.inc`. The user is free to disregard these but is warned that their binaries will no longer port straight to other machines/topologies even if they contain all facilities necessary to run the application. If the number is zero or positive then any processes destined for processor `user_id` will actually be executed on processor `id_in_ncf` as seen by the VCR system.

`id_in_ncf` is the processor to which `user_id` will be mapped. Note that many `user_id`s may be mapped to a single system processor.

`wrap_flag` is 0 then only code targetted at `user_id` will be placed on processor `id_in_ncf`. If any of several records refering to the same `id_in_ncf` have this flag cleared then only those `user_id`s will be mapped to that processor. If the flag is set then multiple virtual processors may still be placed on the node by the VCR run-time.

G Support For LFF Compilers

The VCR 2.0 run-time and configurer only support TCOFF compilers such as the new Inmos `occam` and C compilers in the relevant Toolsets. Support for LFF compilers is limited to the use of the Inmos `icvlink` command which translates LFF binaries into TCOFF binaries. The description hereafter is pertinent to the 3L FORTRAN compiler version 2.0 but is equally applicable to 3L C and Pascal compilers of the same version number. Components of the old Inmos D705b Toolset are also required for this support; these are the linker and the run-time libraries.

The level of support is less than that provided in VCR 1.8 since the clean termination of the scientific code is not generally possible and virtual channels cannot be directly used. This last point would seem to be a major failing but can be worked around by using soft channels within the alien language and a buffering process in the `occam` harness to convert them into virtual channels. This technique has the advantage of also being applicable to extracting host communications from the scientific language run-time library. The compilation sequence is rather circuitous and is not directly supported by `vmakef`. The information presented here is for a T800 target; only obvious changes are required for other processors.

The example code used here is a 3L FORTRAN routine to repeatedly input an integer from a channel, compute its factorial and output the result to another channel. The FORTRAN also prints to the screen to make the example a bit more interesting:

```
C       Example Fortran program using virtual channels routing harness
C       Uses type 3 procedural interface harness
C       This code linked with reduced run time library binary

        INTEGER FUNCTION factorial (x)
        INTEGER x, fac, i

        fac = 1
        DO i = 2, x
            fac = fac * i
        ENDDO
        factorial = fac
        RETURN
        END

        PROGRAM main
        IMPLICIT INTEGER (a-z)
        INCLUDE 'CHAN.INC'
        INTEGER inchan, outchan, number, result

        inchan =  F77_CHAN_IN_PORT  (2)
        outchan = F77_CHAN_OUT_PORT (2)

        PRINT *,'FORTRAN Loaded'
```

```
      DO 100 I=1,10
          PRINT *,'Reading'
          CALL F77_CHAN_IN_MESSAGE  (4, number, inchan)
          result = factorial (number)
          PRINT *,'Writing'
          CALL F77_CHAN_OUT_MESSAGE (4, result, outchan)
 100  CONTINUE

      PRINT *,'FORTRAN DONE'

      END
```

The general technique is to use the 3L compiler and Inmos D705 `occam` Toolset to generate a fully linked code SC unit that can be converted into TCOFF format using `icvlink`. This is then incorporated into a VCR 2.0 application using the `occam IMPORT` mechanism. The use of the hidden option `-zny` on `icvlink` prevents spurious interactive debugging mismatch errors. `No` virtual channels are allowed in the alien language unit, but soft channels can be converted into virtual channels by buffers outside of the alien language. Note that VCR version 1.8 is NOT required at all.

The first stage of compilation requires a 3L FORTRAN and D705B Toolset environment. The `ISEARCH` path should include the FORTRAN directory and the interface libraries directory of the Toolset; the `3LF_INC` path should include the FORTRAN directory in order to locate includes files. Compilations commands are then:

```
t8f fact.for
ilink entry=procent.t8h -f fact.l8h -o fact.c8h
```

where the linker indirect file `fact.l8h` contains:

```
fact.bin
frtlt8.bin
hostio.lib
afhostio.lib
afproc.lib
```

This results in an LFF object file `fact.c8h`. It is then necessary to switch to a D7205 Toolset (or equivalent) and VCR 2.0 environment and issue:

```
icvlink -zny fact.c8h -o alien.t8h
```

to generate an ordinary TCOFF binary which may be incorporated in a VCR 2 code, for example, as follows:

```
#INCLUDE "hostio.inc"

#USE "vhostio.lib"
#IMPORT "alien.t8h"

PROC root (CHAN OF SP fs, ts, CHAN OF BOOL stopper)

  -- any virtual channels must be buffered into soft channel versions
  CHAN OF SP FS, TS :

  CHAN OF ANY occ.to.for, for.to.occ :

  [1]INT dummy.ws :
  [10000]INT work.space :
  [3]INT in.channels :
  [3]INT out.channels :
  [10]INT fac :
  SEQ
    -- Set up channels in call

    LOAD.INPUT.CHANNEL  (in.channels[2],  occ.to.for)
    LOAD.OUTPUT.CHANNEL (out.channels[2], for.to.occ)

    PAR
      [sp.max.packet.size]BYTE buffer :
      INT16 length :
      WHILE TRUE
        SEQ
          TS ? length :: buffer
          ts ! length :: buffer
          fs ? length :: buffer
          FS ! length :: buffer
      entry(FS,TS,1,work.space, dummy.ws, in.channels, out.channels)
      SEQ i = 0 FOR SIZE fac
        SEQ
          occ.to.for ! i
          for.to.occ ? fac[i]

    stopper ! TRUE
:
```

Given the existence of the object file `alien.t8h` this source can be compiled using a makefile generated by `vmakef`. Note that the call to `entry` will not return so that the termination **occam** code is superfluous.

H Software Performance Report

The authors welcome comments upon this release of the virtual channel router. Please use this report form and address it to :

VCR SPR
University of Southampton Parallel Applications Centre
2 Venture Road
Chilworth Research Centre
Southampton
SO1 7NP
United Kingdom

`Company/Organization`

`Full postal address`

`Telephone number`

`Electronic mail address`

`Name of principal user`

`VCR Version and date`

`Target Hardware`

`Host Machine`

`COMMENTS`