VCR 2.0 RPC Support and Run-Time Library

PUMA:034/VCR2/RPC Revision 1.2

Mark Debbage Mark Hill
Denis Nicole

Department of Electronics and Computer Science
University of Southampton

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 using a virtual channels interface to preserve occam syntax and semantics for communication. A static configuration level declares static virtual channels and places user processes onto virtual processors. 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.

Dynamic virtual channels may also be declared and moved around the network via library calls. A remote procedure call mechanism is provided to allow fully dynamic distributed programming. These primitives are targeted at a forthcoming full occam compiler where they will be embedded in the language, but can be used directly by the demanding virtual channel programmer. The level of support exceeds that of the implementation of the `KERNEL.RUN` predefine in the Inmos Toolset, since generalized parameter building primitives are provided.

Contents

1 Introduction

The first release of the virtual channel router provided software routing of virtual channels on T-series transputer machines. It removed the dependency of user applications on the target topology, and allowed static distributed algorithms to be designed in a manner where inter-processor communication was semantically free. The version 1.7 release introduced virtual processors allowing applications to be developed irrespective of the number of transputers available. Dynamic virtual channels, remote procedure calls and generalized parameter passing primitives were added in version 1.8 in order to support the system requirements for dynamic distributed occam . This release of VCR offers both occam and C bindings to an improved interface to the dynamic library.

This technical note documents the applications library interface to VCR. This library includes access to virtual communication, access to the system heap, the remote procedure call mechanism and the dynamic channel primitives. The full specification of the occam and C bindings is given in Appendix A. In addition, the remote procedure call information is presented in a more readable format in the main body of the text. Only occam interface calls are shown in the text for clarity.

The improvements in the library over previous versions are:

- System heap can be accessed from user code.
- Provision of a low-level interface to the dynamic channel primitives, independent of the RPC mechanism.
- An improved user interface to the RPC library removes the need for user assembler inserts.
- Proper bracketing of pass channel primitives, permits multiple `PLACED.RUN`s within the scope of a parameter block build.
- The ability to pass both values and variables by reference (older versions supported only the latter).
- Automatic passing of the vector space pointer (if it is required).
- Automatic returning of variable data into the caller's workspace.
- Improved user exception handling.
- Retrograde compatibility with version 1.8 primitives (with one constraint, see Appendix B).

In addition, improvements in the compiler allow pointers to be retyped to channels allowing dynamic channel ends to be used as channels in the same module.

2 Motivation For Distributed occam

The transputer implementation of distributed occam has severe restrictions compared to the single transputer version. Currently, there can only be one PLACED process per processor, the termination of this process is not signaled, it has no free variables and it can communicate to other PLACED processes through only 4 channel pairs (the links) [1]. Hence traditional distributed occam programs are highly configuration dependent.

The virtual channels and virtual processor features of VCR relax these constraints for static configurations. To exploit the same mechanisms dynamically, dynamic virtual channels and the provision of remote procedure calls are required. The eventual aim is an implementation where the keyword PLACED is merely a modifier to any PAR construct informing the compiler to distribute the branches of the PAR over the available processors. Consider the following algorithm to compute the sin function over a vector of real numbers by divide-and-conquer (lifted from [3]):

```
PROC sin.vector ( []REAL32 sinv, VAL []REAL32 v )
  VAL s IS SIZE v :
  IF
    s > 1
      VAL bottomhalf IS s/2 :
      VAL tophalf    IS (s/2) + (s\2) :
      PLACED.PAR
        sin.vector ( [sinv FROM 0 FOR bottomhalf],
                     [v    FROM 0 FOR bottomhalf] )
        sin.vector ( [sinv FROM bottomhalf FOR tophalf],
                     [v    FROM bottomhalf FOR tophalf] )
    s = 1
      sinv[0] := sin(v[0])
:
```

This algorithm cannot be expressed in current occam due to the recursion and the PLACED PAR. To implement the code would require a rewrite to avoid recursion, a configuration level to define the target architecture and explicit message-passing.

3 Remote Procedure Call Implementation

This section describes the occam bindings for the remote procedure call and parameter passing primitives of version 2.0 of the virtual channel router. A more formal description of the calls is given in Appendix A which presents both the occam and the C interface. The VCR library will support the version 1.8 user code (with one constraint, see Appendix B, as well as the improved mechanism discussed here. Information for upgrading version 1.8 codes to the new primitives is given in Appendix C.

The basic unit of distributable code is the separately compiled unit (SC). For occam code these are the direct outputs of the linker, whilst for C source an equivalent occam process must be constructed. An SC has the following characteristics:

- A single entry point
- No unresolved references
- Bounded workspace and vecspace requirements
- A procedural interface specification
- No static link to workspace of the caller

Remote procedure calls (RPC) allow an SC to be dynamically loaded and run from a remote node, with full occam parameter passing into and from the RPC. At present these parameter lists must be built by hand using low-level primitives, with the result that the procedural interface cannot be checked by the compiler.

3.1 RPC Format

The implementation is based on the dynamic channel, dynamic memory management and distributed code-loading facilities of VCR. The general structure of the coding of a RPC is shown below:

```
INT pblock :
VAL target    IS ... :
VAL num.chans IS ... :
VAL num.refs  IS ... :
VAL num.vals  IS ... :
VAL data.size IS ... :
SEQ
  INIT.CALL  (pblock, target, num.chans, num.refs, num.vals, data.size)
  ... Build parameter block
  PLACED.RUN (pblock, "SCname")
  DONE.CALL  (pblock)
```

A parameter block is first allocated and initialised with some parameters characterizing the RPC. Parameters to the RPC unit are built into the parameter block by a list of primitives (not shown here). The code `SCname` is then invoked on the remote node by the call to `PLACED.RUN`. This procedure creates a channel to the target transputer and passes the code filename and parameter block down the channel. Upon receiving the filename the target transputer checks its code database to see if it has that code, otherwise it requests the code from the root. Once the code is available it is run, and on completion a returning parameter block is sent to the caller signaling termination of the RPC. The parameter block in then passed into a primitive (`DONE.CALL`) which returns variable data to the user's workspace and deallocates the parameter block memory.

3.2 Parameter Blocks

The parameter block consists of the following internal fields:

State storage to hold information between successive calls to the parameter building primitives.

Parameter field created by calling a primitive for each object to be passed in the order in which they are specified in the target code's procedural interface.

Variable data field to hold variable copy-out data, and to receive copy-in data upon completion of the RPC.

Value data field to hold constant copy-out data.

Relocation list to store the offsets of pointers in the parameter list for translation at the target end.

Channel list to store references to all virtual channels moved to the target end.

Variable data pointer list to allow the automatic returning of variable copy-in data to the caller's workspace.

3.3 Parameter Block Initialization

A parameter block is characterized by specifying five values.

1. The target processor number (`target`) determines the destination for the RPC. This is a virtual processor id which is mapped to a physical processor by the run-time. The current virtual processor id is usually passed in from the configuration level. The current physical processor number can be found by:

   ```
   PROC.ID (proc.id)
   ```

   The number of worker transputers in the network (i.e. *excluding* the root) can be found using:

   ```
   NUM.PROCS ( num.procs )
   ```

A random processor number (excluding the root) can be generated using the primitive:

```
RANDOM.PROC ( target )
```

2. The number of channels to be passed (`num.chans`) is used to size the length of the channel list in the parameter block.

3. The number of objects to be passed by reference (`num.refs`) is used to size the length of the relocation list in the parameter block.

4. The number of words to be passed by value (`num.vals`) is a constant and can be deduced from the procedural interface.

5. The number of words to hold the copy-out data area (`data.size`); this value will need to be calculated if variable sized arrays are passed.

Note that each of the values `num.chans`, `num.refs`, `num.vals` and `data.size` are used to size fields in the parameter block. Techniques to ascertain their values for any particular interface are given in Section 3.8, but they can be oversized if necessary. A run-time error will be generated if any are under-sized.

These values are passed into a parameter block initialization primitive (`INIT.CALL`), along with a word to hold state for the parameter block pointer. This primitive requests sufficient memory for the parameter block and stores its pointer in `pblock`. State for the primitives list is initialized into the bottom end of the block, and if necessary the parameter list is automatically padded out to 3 words (the minimum for a transputer procedure call).

3.4 Pass Word By Value

In occam only objects that fit in a single word can be passed by value — all others are passed by reference. The data types `VAL INT`, `VAL BYTE`, `VAL INT16`, `VAL INT32`, `VAL BOOL` and `VAL REAL32` are thus passed as words by value. The index ranges of variable-sized arrays are also passed as `VAL INT`s and appear after the array reference in order of decreasing dimensions. To pass one of these data types it must be retyped or converted (if necessary) to a word and passed by the following primitive:

```
PASS.BY.VALUE ( pblock, word )
```

3.5 Pass Value Array Of Byte By Reference

All constant objects requiring more than one word of storage are passed by reference. It is necessary to retype the `object` into an array of bytes before passing it into the primitive:

```
VAL []BYTE object.in.bytes RETYPES object :
PASS.VAL.BY.REF (pblock, object.in.bytes)
```

3.6 Pass Variable Array Of Bytes By Reference

All objects which may be modified by the called procedure are passed as variables by reference. It is necessary to retype the `object` into an array of bytes before passing it into the primitive:

```
[]BYTE object.in.bytes RETYPES object :
PASS.VAR.BY.REF (pblock, object.in.bytes)
```

Note that this primitive implicitly word aligns after the object has been copied into the copy-out area, and that this padding space must be catered for by the data size value passed into the initialization routine (this value can be oversized if required).

3.7 Channel Parameters

Passing a channel end through a remote procedure call requires that channel end to be moved to a remote node on the call, and moved back upon returning. This is a complex problem as the interactions between both ends of a channel moving must be secure. Only dynamic virtual channels may be moved; configuration-level virtual channels are static and are therefore not valid parameters to RPCs. All virtual channels in the VCR are unidirectional.

3.7.1 Dynamic Virtual Channels

Primitives are provided to declare and release state for virtual channels, and to initialise them to form a unidirectional virtual channel within a processor. The parameters to `NEW.CHANNEL` are directed : the output end first followed by the input end. The order of parameters was not specified in the old interface documentation since it was immaterial, with the result that some old user codes may require modifications. These channel ends can be used internally as virtual channels or passed as parameters to RPCs.

A unidirectional channel is created and disposed using:

```
INT out.end, in.end :
SEQ
  NEW.CHANNEL (out.end, in.end)
  ... use dynamic virtual channel
  DISPOSE.CHANNEL (out.end, in.end)
```

3.7.2 Channel Representation

All virtual channels are handled by the RPC interface as being of type `INT`, this integer actually containing a pointer to the channel structure. The reasoning for this is that the call to `NEW.CHANNEL` needs to return a channel pointer, an invalid operation with proper `CHAN`s, and because the interface routines need to be independent of user channel protocol.

A `channel` can be retyped into a `chan.ptr` by the occam code:

```
VAL INT chan.ptr RETYPES channel :
```

and a `chan.ptr` can be retyped into a `channel` (of a particular `PROTOCOL`) by the occam code:

```
CHAN OF PROTOCOL channel RETYPES chan.ptr :.
```

Arrays of virtual channels are thus of type `[]INT`, `[][]INT` ... depending on the required dimensionality, where each element of the array is a pointer to a channel structure. These arrays can be `RETYPE`d to and from `CHAN` arrays using the obvious extensions to the single channel examples above.

Note that occam does not differentiate between input and output channel ends so that care is required to ensure that these unidirectional channel pointers are used in the correct orientation.

3.7.3 Passing Single Channels

The channel must be a dynamic virtual channel, and represented by a channel pointer held in an `INT` as described above. The pointers returned by `NEW.CHANNEL` are thus suitable. The channel reference is built into the parameter block by the call:

```
PASS.CHANNEL (pblock, chan.ptr).
```

Note that this call does not actually move the channel. This is delayed until the point of RPC. This allows local RPCs to be implemented without the move channel overheads and also allows multiple RPCs to be undertaken with the same parameter block.

3.7.4 Passing Arrays of Channels

The channels must be dynamic virtual channels, and represented by a single-dimensional array of `INT`s. Higher dimension arrays will require an abbreviation. The channel references are built into the parameter block by the call:

```
PASS.CHANNEL.ARRAY (pblock, chan.ptr.array).
```

Again, this call does not actually move any channels.

3.7.5 Implementation of Move Channel

Low-level move channel primitives are implicitly called by an RPC to move channel parameters to the target node upon invocation, and to return them to the parent node upon completion. This is accomplished by a sequence of secure control packet transfers between the channel ends, which maintain channel integrity whatever the state of the other channel end (idle, outputting, inputting or moving). If the other end is not moving then there can be at most one packet from that channel end in the network, due to the message acknowledgement of VCR. This packet can arrive at the moving channel end and has to be forwarded to the new channel end. Space is reserved in the virtual channel blocks in order to through-route the control packets as they pass from node to node.

3.8 Sizing Parameter Blocks

The values of the RPC constants `num.refs` and `num.vals` can be ascertained by inspection of the parameter building list of the SC unit to be called. Every occurrence of a `PASS.BY.VALUE` or `PASS.CHANNEL` counts as one towards `num.vals`[1]. Every occurrence of a `PASS.VAL.BY.REF`, `PASS.VAR.BY.REF` or `PASS.CHANNEL.ARRAY` counts as one towards `num.refs`.

The vector space parameter (if required) is automatically handled by the system and is not of concern to the user.

The value of `num.chans` is determined by the total number of channel ends passed through the RPC. A single channel obviously counts as one channel, whilst a one-dimensional array of `n` channels counts as `n` channels. This value will need to be calculated if variable sized arrays of channels are passed.

The value of `data.size` is determined by the total amount of copy-out data storage required in the parameter block. The amount is specified in words, and needs to include space for all objects passed by reference (whether as `VAL`s or as `VAR`s), *and* for any channel arrays passed. Note that since the primitives implicitly word align after the object has been copied into the copy-out area, any extra padding space must be catered for by this value (it can be oversized if required).

3.9 Other RPC Calls

The call:

```
RPC.TARGET (pblock, target)
```

allows the RPC target to be switched to real processor `target`. The same constraints apply to the target as in the `INIT.CALL` routine. This call is normally used if it is wished to make multiple RPCs to different targets but with the same parameter block.

[1]Note that dimensions of unsized arrays, including unsized channel arrays, are passed in as `VAL INT`s and thus contribute towards this total.

4 Examples

When developing a remote version of a procedure call the remote procedure call sequence should be wrapped up in occam with the original procedure name preceded by `remote.` and the same procedural interface (bar channels which should be `INT`s rather than `CHAN`s). This ensures that some type checking can still be performed by the compiler.

This examples section contains remote procedure calls for typical parameter lists. The last example is a complete source listing for the VCR implementation of the divide-and-conquer sine algorithm mentioned in Section 2.

1.
```
PROC remote.test ( VAL INT a, INT b, VAL [10]INT c, []INT d )

  INT pblock :
  VAL num.chans IS 0 :
  VAL num.refs IS 3 :
  VAL num.vals IS 2 :
  VAL data.size IS (1 + SIZE c) + SIZE d :
  []BYTE b.in.bytes RETYPES b :
  VAL []BYTE c.in.bytes RETYPES c :
  []BYTE d.in.bytes RETYPES d :
  INT target :
  SEQ
    RANDOM.PROC     (target)
    INIT.CALL       (pblock, target, num.chans, num.refs, num.vals, data.size)
    PASS.BY.VALUE   (pblock, a)
    PASS.VAR.BY.REF (pblock, b.in.bytes)
    PASS.VAL.BY.REF (pblock, c.in.bytes)
    PASS.VAR.BY.REF (pblock, d.in.bytes)
    PASS.BY.VALUE   (pblock, SIZE d)
    PLACED.RUN      (pblock, "test.cah")
    DONE.CALL       (pblock)
:
```

2.
```
PROC remote.test (VAL REAL32 a, REAL32 b)

  INT pblock :
  VAL num.chans IS 0 :
  VAL num.refs IS 1 :
  VAL num.vals IS 1 :
  VAL data.size IS 1 :
  VAL INT a.word RETYPES a :
  []BYTE b.in.bytes RETYPES b :
  INT target :
  SEQ
    RANDOM.PROC     (target)
    INIT.CALL       (pblock, target, num.chans, num.refs, num.vals, data.size)
    PASS.BY.VALUE   (pblock, a.word)
```

```
    PASS.VAR.BY.REF (pblock, b.in.bytes)
    PLACED.RUN      (pblock, "test.cah")
    DONE.CALL       (pblock)
:
```

3. PROC remote.test ([]BYTE a, b)

```
  INT pblock :
  VAL num.chans IS 0 :
  VAL num.refs IS 2 :
  VAL num.vals IS 2 :
  -- Must ensure sufficient space is reserved in case arrays are not
  -- multiples of a word in length
  VAL data.size IS (((SIZE a)/bytes.per.word)+1)+
                   (((SIZE b)/bytes.per.word)+1) :
  INT target :
  SEQ
    RANDOM.PROC     (target)
    INIT.CALL       (pblock, target, num.chans, num.refs, num.vals, data.size)
    PASS.VAR.BY.REF (pblock, a)
    PASS.BY.VALUE   (pblock, SIZE a)
    PASS.VAR.BY.REF (pblock, b)
    PASS.BY.VALUE   (pblock, SIZE b)
    PLACED.RUN      (pblock, "test.cah")
    DONE.CALL       (pblock)
:
```

4. PROC remote.test (VAL INT channel.in, channel.out)

```
  -- Remote end is PROC test(CHAN OF protocol channel.in, channel.out)
  INT pblock :
  VAL num.chans IS 2 :
  VAL num.refs  IS 0 :
  VAL num.vals  IS 2 :
  VAL data.size IS 0 :
  INT target :
  SEQ
    RANDOM.PROC    (target)
    INIT.CALL      (pblock, target, num.chans, num.refs, num.vals, data.size)
    PASS.CHANNEL   (pblock, channel.in)
    PASS.CHANNEL   (pblock, channel.out)
    PLACED.RUN     (pblock, "test.cah")
    DONE.CALL      (pblock)
:
```

5. Distributed divide-and-conquer algorithm

```
#USE "vcr.lib"
#USE "snglmath.lib"
#INCLUDE "vclib.inc"

PROC sin.vector( []REAL32 sinv, VAL []REAL32 v )
  VAL s IS SIZE v :
  IF
    s > 1
      VAL bottomhalf IS s/2 :
      VAL tophalf    IS (s/2) + (s\2) :
      []BYTE b.sinv.in.bytes  RETYPES [sinv FROM 0 FOR bottomhalf] :
      []BYTE t.sinv.in.bytes  RETYPES [sinv FROM bottomhalf FOR tophalf] :
      VAL []BYTE b.v.in.bytes RETYPES [v FROM 0 FOR bottomhalf] :
      VAL []BYTE t.v.in.bytes RETYPES [v FROM bottomhalf FOR tophalf] :
      PAR
        --{{{  spawn bottom half
        --{{{  constants
        VAL num.chans IS 0 :
        VAL num.ptrs  IS 2 :
        VAL num.vals  IS 2 :
        VAL data.size IS bottomhalf*2 :
        --}}}
        --{{{  variables
        INT pblock :
        INT target :
        --}}}
        SEQ
          RANDOM.PROC     (target)
          INIT.CALL       (pblock, target, num.chans, num.ptrs, num.vals, data.size)
          PASS.VAR.BY.REF (pblock, b.sinv.in.bytes)
          PASS.BY.VALUE   (pblock, bottomhalf)
          PASS.VAL.BY.REF (pblock, b.v.in.bytes)
          PASS.BY.VALUE   (pblock, bottomhalf)
          PLACED.RUN      (pblock, "sinvec.c8h")
          DONE.CALL       (pblock)
        --}}}
        --{{{  spawn top half
        --{{{  constants
        VAL num.chans IS 0 :
        VAL num.ptrs  IS 2 :
        VAL num.vals  IS 2 :
        VAL data.size IS tophalf*2 :
        --}}}
        --{{{  variables
        INT pblock :
        INT target :
```

```
        --}}}
        SEQ
          RANDOM.PROC     (target)
          INIT.CALL       (pblock, target, num.chans, num.ptrs, num.vals, data.size)
          PASS.VAR.BY.REF (pblock, t.sinv.in.bytes)
          PASS.BY.VALUE   (pblock, tophalf)
          PASS.VAL.BY.REF (pblock, t.v.in.bytes)
          PASS.BY.VALUE   (pblock, tophalf)
          PLACED.RUN      (pblock, "sinvec.c8h")
          DONE.CALL       (pblock)
        --}}}

    s = 1
      sinv[0] := SIN (v[0])
:
```

Bibliography

1 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.

2 Debbage, M., Hill, M. B., and Nicole, D. A. Virtual channel router version 2.0 user guide. Tech. rep., University Of Southampton, June 1990.

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

A Specification Of VCR Version 2.0 Library Routines

This appendix documents the interface between occam and C applications and the VCR kernel. This comprises the libraries `virtual.lib` and `vcr.lib`, and the definitions files `vclib.inc` and `virtual.h`. Both occam and C procedural interfaces are given and their behaviour defined. All routines are valid when called from *either* priority. However, for best communications performance it is strongly recommended that user code executes at low priority.

occam source implicitly references the file `virtual.lib` to expose routines for virtual communication, and hence the communications interface for occam is pure occam and there is no need to reveal procedural equivalents. The other primitives are used by `#USE`ing `vcr.lib`. Any constants referenced in the occam bindings here can be declared by including `vclib.inc`.

C source needs to reveal the prototypes for the primitives and any associated definitions by including `virtual.h`. The virtual communications routines require that at least 50 words of workspace is available on the stack. For all other routines the stack requirement is less than 150 words.

Note that in many cases these routines are actually just stubs which link dynamically to kernel functions which are part of the run-time system. In this ways applications code is disjoint from many of the kernel parameters and its implementation.

A.1 Data Structures

The data structures prototyped in `virtual.h` are,

VCB a generic virtual channel block, this is qualified internally to make it input or output, static or dynamic, internal or external.

SCDB a separately compiled code data block.

PBLOCK a parameter block.

Note that only pointers to these structures are used so that user code is independent of their implementation.

A.2 Communications Library (C only)

This sections defines the communications interface from C, the occam interface being defined in the language itself. Virtual channels in user code are denoted by always having their lowest bit set, as opposed to soft channels which have it clear. Hence virtual communications routines can be applied to channels irrespective of whether they are virtual, soft or hard. These routines can therefore be used as direct replacements for the Inmos routines (with appropriate retyping of `Channels` to `VCBs`). As in the occam model both parties to a communcation must know the length of the transaction. Further information about the use of the communications library can be found in [2].

`EXTERN void VChanIn (VCB *c, void *message, int length) ;`

> Inputs `length` bytes from virtual channel `c` into memory beginning at `message`. `length` may be zero, in which case the participating pair of processes synchronize without data transfer.

`EXTERN char VChanInChar (VCB *c) ;`

> Returns a single character input from virtual channel `c`.

`EXTERN int VChanInInt (VCB *c) ;`

> Returns a single integer input from virtual channel `c`.

`EXTERN void VChanOut (VCB *c, void *message, int length) ;`

> Outputs `length` bytes from memory beginning at `message` to virtual channel `c`. `length` may be zero, in which case the participating pair of processes synchronize without data transfer.

`EXTERN void VChanOutChar (VCB *c, char ch) ;`

> Outputs a single character `ch` to virtual channel `c`.

`EXTERN void VChanOutInt (VCB *c, int i) ;`

> Outputs a single integer `i` to virtual channel `c`.

A.3 Miscellaneous VCR Library Calls

```
INT FUNCTION MALLOC (VAL INT int.size)
EXTERN void *Malloc (int size) ;
```

> This routines allocates `int.size` words (occam) or `size` bytes (C) from the system heap and returns a pointer to this area. If a full C module is used then it can access local heap with `malloc` and system heap with `Malloc`. Heap failure is indicated by a null pointer for C or a run-time error message for occam .

```
PROC FREE (VAL INT pointer)
EXTERN void Free (void *pointer) ;
```

> This routine frees the previously allocated memory from the system heap.

```
PROC EXCEPTION (VAL INT severity, VAL []BYTE file, VAL INT line, VAL []BYTE msg)
EXTERN void Exception (int severity, char file[], int line, char msg[]) ;
```

This call generates a VCR exception message. `msg` is the desired error string, `file` holds the filename of the code module and `line` the offending line number. In the C Toolset the `file` and `line` parameters are typically macro substituted. However, with occam no predefines are available so the typical usage is to declare a string `FILE` as the name of the module, and to pass in the constant `LINE` from `vclib.inc` to denote an unimplemented feature.

If `severity` is `UPR.warning`, the exception call returns immediately whilst the message is routed back to the root processor and then displayed on `stderr`. `UPR.error` is similar but the exception call never returns. If the severity is `UPR.fatal`, no attempt is made to route the message back, the message pointer is stored into a global location, the error flag set, and the processor halted.

```
PROC ERROR (VAL []BYTE message)
```

Error generating routine for occam provided for compatibility with VCR version 1.8.

```
PROC FIND.CODE (VAL []BYTE file, INT scdb)
EXTERN SCDB* RPC_FindCode (char *filename) ;
```

This routines searches the node database for the code unit denoted by `file` or `filename`. The C routine returns zero if it is not found, or the pointer if it is. For occam , `scdb` is zero if it is not found, or a pointer to the unit if it is.

```
PROC LOAD.CODE (VAL []BYTE file, INT scdb)
EXTERN SCDB* RPC_LoadCode (char *filename) ;
```

This routine can be used to force the code unit denoted by `file` or `filename` onto a particular processor. A pointer to the unit is returned upon success, and a run-time error generated upon failure.

A.4 RPC Library Calls

The remote procedure call library is used to call code on a remote VCR processor. Only occam SC code units may be called; C routines must be wrapped up in a suitable occam harness. Processor targets are identified by a virtual processor number which is mapped to a physical processor by the run-time.

```
PROC INIT.CALL (INT pblock, VAL INT target, num.chans, num.ptrs,
num.vals, data.size)
EXTERN PBLOCK *RPC_InitCall (int target, int num_chans, int num_ptrs,
int num_vals, int data_size) ;
```

Call to initialise a parameter block for a subsequent RPC. The parameters specify the virtual processor id for the target of the RPC, the number of channel ends to be passed, the number of objects to be passed by reference, the number of words to be passed by value and the total copy-out data size in words. These values are used to size the parameter block memory requirements, which is then allocated, initialised and returned as a pointer in `pblock`.

```
PROC PLACED.RUN (VAL INT pblock, VAL []BYTE file)
EXTERN void RPC_PlacedRun (PBLOCK *pb, char *filename) ;
```

This call executes a remote procedure call invoking the code unit referenced by file-name with the required parameter block. The file unit must be an occam (or equivalent) separately compiled unit. The call returns upon completion of the RPC.

```
PROC DONE.CALL (VAL INT pblock)
EXTERN void RPC_DoneCall (PBLOCK *pb) ;
```

This call returns any copy-in data back to the user's workspace and deallocates the parameter block.

```
PROC NEW.CHANNEL (INT local.out, local.in)
EXTERN void RPC_NewChannel (VCB **local_out, VCB **local_in) ;
```

This creates a unidirectional local virtual channel connecting the channel referenced by `local.out` to `local.in`.

```
PROC DISPOSE.CHANNEL (VAL INT local.out, local.in)
EXTERN void RPC_DisposeChannel (VCB *local_out,  VCB *local_in) ;
```

This destroys the unidirectional local virtual channel connecting the channel referenced by `local.out` to `local.in`.

```
PROC PROC.ID (INT proc.id)
EXTERN int RPC_ProcId (void) ;
```

Returns the *physical* processor id.

```
PROC NUM.PROCS (INT num.procs)
EXTERN int RPC_NumProcs (void) ;
```

Returns the number of physical processors in the network, *excluding* the root.

```
PROC RANDOM.PROC (INT proc)
EXTERN int RPC_RandomProc(void) ;
```

Returns a random processor number. If the network contains only a single processor it returns 0, else it returns a random number between 1 and the highest physical processor id (inclusive). The implementation uses a feedback shift register seeded differently across the network.

```
PROC PASS.BY.VALUE (VAL INT pblock, value)
EXTERN void RPC_PassByValue (PBLOCK *pb, int value) ;
```

Builds a single word value parameter into the parameter block.

```
PROC PASS.VAL.BY.REF (VAL INT pblock, VAL []BYTE vector)
EXTERN void RPC_PassValByRef (PBLOCK *pb, char *pointer, int num_bytes) ;
```

Builds a constant array of bytes into the parameter block.

```
PROC PASS.VAR.BY.REF (VAL INT pblock, []BYTE vector)
EXTERN void RPC_PassVarByRef (PBLOCK *pb, char *pointer, int num_bytes) ;
```

Builds a variable array of bytes into the parameter block.

```
PROC PASS.CHANNEL (VAL INT pblock, chan.ptr)
EXTERN void RPC_PassChannel (PBLOCK *pblock, VCB *channel) ;
```

Builds a channel into the parameter block. The channel should be a pointer to a local dynamic virtual channel since it will be moved in an RPC.

```
PROC PASS.CHANNEL.ARRAY (VAL INT pblock, VAL []INT chan.ptr.array)
EXTERN void RPC_PassChannelArray (PBLOCK *pblock, VCB **channel, int num_chans) ;
```

Builds a one-dimensional array of channels into the parameter block. The array should contain pointers to local dynamic virtual channels since they will be moved in an RPC.

```
PROC RPC.TARGET (VAL INT pblock, target)
EXTERN void RPC_Target (PBLOCK *pblock, int target) ;
```

Changes the target virtual processor associated with the parameter block.

A.5 Low-Level Dynamic Virtual Channel Primitives

These primitives can be used to create, destroy, move and return dynamic virtual channels. Processor targets are identified by a virtual processor number which is mapped to a physical processor by the run-time. Some of the routines return a remote channel pointer; this can only be made use of by an RPC to that remote node or a communication to an existing process on that remote node.

```
PROC DVC.CREATE.CHANNEL.OUT (INT local.out, remote.in, VAL INT remote)
EXTERN void DVC_CreateChannelOut (VCB **local_out, VCB **remote_in, int remote) ;
```

This routine creates a dynamic virtual channel end, with an outputting end on the calling processor and an inputting end on the specified remote virtual processor. The case of the remote virtual processor also being local is handled efficiently. The call returns a local output channel pointer and a remote input channel pointer.

```
PROC DVC.CREATE.CHANNEL.IN (INT local.in, remote.out, VAL INT remote)
EXTERN void DVC_CreateChannelIn (VCB **local_in, VCB **remote_out, int remote) ;
```

This routine creates a dynamic virtual channel end, with an inputting end on the calling processor and an outputting end on the specified remote virtual processor. The case of the remote virtual processor also being local is handled efficiently. The call returns a local input channel pointer and a remote output channel pointer.

```
INT FUNCTION DVC.MOVE.CHANNEL (VAL INT target, local.chan)
EXTERN VCB *DVC_MoveChannel (int target, VCB *c) ;
```

The local *dynamic* channel end (either input or output) referenced by `local.chan` is moved to the required `target` virtual processor id and the remote channel pointer returned.

```
PROC DVC.RETURN.CHANNEL (VAL INT chan)
EXTERN void DVC_ReturnChannel (VCB *c) ;
```

The dynamic channel end referenced by the local channel pointer is returned to its parent.

```
PROC DVC.DESTROY.CHANNEL.END (VAL INT chan)
EXTERN void DVC_DestroyChannelEnd (VCB *c);
```

State for the dynamic channel end referenced by the local channel pointer is destroyed.

A.6 Routines For Version 1.8 Compatibility

The following routines are provided for compatibility with the version 1.8 set of dynamic occam primitives. Only occam interfaces are given here since their inclusion is purely for old code support.

`PROC PASS.VECSPACE (VAL INT plock)`

Provided only as an empty stub to resolve references.

`PROC PASS.BY.REF (VAL INT pblock, pointer, num.bytes)`

Calls equivalent routine `PASS.VAR.BY.REF`.

`PROC MOVE.CHANNEL (VAL INT pblock, source.chan, INT destn.chan)`

Calls move channel routine directly. This cannot be used in the same parameter build as `PASS.CHANNEL` or `PASS.CHANNEL.ARRAY`.

`PROC RETURN.DATA (VAL INT pblock, pointer, num.bytes)`

Returns data to user's workspace. Invocation inhibits implicit returning in `DONE.CALL`.

`PROC SKIP.DATA (VAL INT pblock, num.bytes)`

Skips data in the copy-in field. Invocation inhibits implicit returning in `DONE.CALL`

B Compatibility With Version 1.8 RPC Primitives

The version 2.0 primitives set have been designed to be compatible with the version 1.8 set. They are indeed, fully compatible with the code sequences generated by the first version of the Syseca full occam compiler.

However, one potential problem exists in the coding of `NEW.CHANNEL` calls. In the old set the call created a undirected channel but did not differentiate between the two ends (much like a conventional occam channel word). The new set does differentiate and the call is now,

```
NEW.CHANNEL (out.end, in.end)
```

and misuse of the channel ends results in run-time errors. The order chosen for the new set is compatible with the Syseca compiler, but potentially incompatible with some user code (e.g. the old VCR examples).

C Modifying Version 1.8 Codes To New Primitives

Upgrading old codes to the new style parameter passing is relatively straight-forward,

- If the old parameter list contains a `PASS.VECSPACE`, delete it and reduce `num.refs` by 1.
- If the old parameter list contains `RETURN.DATA`s and/or `SKIP.DATA`s delete them.
- For each `PASS.BY.REF (pblock, object.ptr, size)` in the old parameter list, replace it by,

  ```
  []BYTE object.in.bytes RETYPES object :
  PASS.VAR.BY.REF (bpblock, object.in.bytes)
  ```

  if the object is a `VAR` parameter, or

  ```
  VAL []BYTE object.in.bytes RETYPES object :
  PASS.VAL.BY.REF (bpblock, object.in.bytes)
  ```

  if the object is a `VAL` parameter. Delete any other code used to obtain pointers.
- Replace each occurrence in the old parameter list of,

  ```
  SEQ
    MOVE.CHANNEL (pblock, chan.ptr, move.chan.ptr)
    PASS.BY.VALUE (pblock, moved.chan.ptr)
  ```

  with,

  ```
  PASS.CHANNEL (pblock, chan.ptr)
  ```
- Replace each occurrence in the old parameter list in the form of,

  ```
  [n]INT moved.chan.array :
  INT moved.chan.array.ptr :
  SEQ
    GUY
      LDLP moved.chan.array
      STL  moved.chan.array.ptr

    SEQ i = 0 FOR n
      MOVE.CHANNEL (pblock, chan.array[i], move.chan.array[i])

    PASS.BY.REF (pblock, moved.chan.array.ptr, n*4)
  ```

  with,

  ```
  PASS.CHANNEL (pblock, chan.array)
  ```
- Check sizing of parameter block constants with rules in Section 3.8, although no modifications should be necessary.