Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

Interfacing C and occam-pi

Fred BARNES

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, England.
F.R.M.Barnes@kent.ac.uk

Abstract. This paper describes an extension to the KRoC occam-π system that allows processes programmed in C to participate in occam-π style concurrency. The uses of this are wide-ranging, from providing low-level C *processes* running concurrently as part of an occam-π network, through to concurrent systems programmed entirely in C. The easily extended API for C processes is based on the traditional Inmos C API, used also by CCSP, extended to cover new features of occam-π. One of the motivations for this work is to ease the development of low-level network communication infrastructures. A library that provides for networking of channel-bundles over TCP/IP networks is presented, in addition to initial performance figures.

Keywords. C, occam-pi, concurrency, processes, networks

Introduction

The occam-π language [1] extends classical occam [2] in numerous ways. Included in these extensions, and supported by the KRoC [3] implementation, are mechanisms that allow occam-π processes to interact with the external environment. Classical occam on the Transputer [4] had a very physical environment — hardware links to other Transputers. In contrast, modern systems support highly dynamic application environments, e.g. file-systems and networking, that occam-π applications should be able to take full advantage of.

In most cases, interaction with anything external to an occam-π program requires interfacing with C — since the environments in which KRoC programs run have C as a common interface (e.g. UNIX). There are a few exceptions, however, such as the mechanism that provides low-level hardware I/O access directly from occam-π using "`PLACED PORT`"s (described in [5]).

The mechanisms currently support by KRoC for interfacing with C are: simple external C calls [6]; blocking external C calls [7]; and a "user defined channels" mechanism that allows C calls (blocking and non-blocking) to be placed behind channel operations, including direct support for `ALT`ing on completion of external calls. These mechanisms, although mostly adequate, lack the level of flexibility that programmers require. For example, it is not immediately clear as to how a low-level network communication infrastructure, such as that required by KRoC.net [8], would be implemented using the existing mechanisms.

All of these existing mechanisms essentially attach 'dead' C function calls to various occam-π operations. Programming interactions between these calls, which would be required if multiplexing channels over IP links, is difficult and prone to error. On the other hand, most of the infrastructure could be programmed in occam-π, with only the lowest-level I/O inside C functions. However, occam-π does not lend itself to the type of programming we might wish to employ at this level — e.g. deliberate pointer aliasing for efficiency (which we know to be safe, but which cannot be checked by the current occam-π compiler).

The C interface mechanism presented here (CIF) attempts to address these issues, by providing a very general framework for the construction of parallel processes and programs in C. In some respects, this mechanism provides exactly what CCSP [9] provided in terms of support for C programs, but with the added benefits of occam-π (e.g. mobiles and extended synchronisations) and the ability to support mixed occam-π and C process networks. The interface presented to applications is based on the original Inmos and CCSP APIs.

The uses for this are wide-ranging. Applications that require only a limited amount of external interaction can encapsulate these in concurrent C processes, avoiding the overheads of repeated external C calls. The CIF mechanism can also be used to migrate existing C code into occam-π systems — e.g. minimal-effort porting of Linux device-drivers to RMoX [5]. At the far end of the scale, the CIF mechanism can be used to program entire concurrent systems in C. In contract with some alternative parallel C environments, CIF offers very low overheads and a reasonable level of control. Unlike occam-π, however, the C compiler — typically 'gcc' [10] — does not perform parallel-usage checks, leaving the potential for race-hazard errors. The opportunity for such error can be minimised by good application design.

Section 1 examines the technical aspects of the C interface, implementation and API. Section 2 presents a specific application of CIF for networking mobile channel-bundles, in addition to a general discussion of potential application areas. Conclusions and initial performance results are presented in section 3, together with plans for future work.

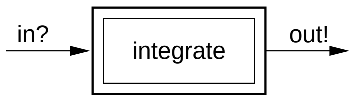
1. Interfacing C and occam-π

The C interface operates by encapsulating C processes such that the KRoC run-time system sees them as ordinary occam-π processes. No changes are required in the KRoC run-time to support these C processes, and no damage is caused to the performance of existing occam-π code. As a consequence, C processes incur a slight overhead each time they interact with the run-time system (switching from a CIF *process-context* to an occam-π one). This overhead is small, however (less than 100 nanoseconds on an 800 MHz Pentium-3).

C processes are managed through a variety of API calls, the majority of which require a *C process context*. Some do not, however, including those used for initial creation of C processes. Creation and execution of the first C process in a system is slightly complicated, requiring the use of the basic C calling mechanism. For example, using the C interface, the standard '`integrate`' component could be written as:

```
void integrate (Process *me, Channel *in, Channel *out)
{
    int v, total = 0;

    for (;;) {
        ChanInInt (in, &v);
        total += v;
        ChanOutInt (out, total);
    }
}
```



The '`me`' parameter given to CIF processes gives the process a handle on itself. The CIF infrastructure always knows which particular C process is executing, however, raising questions about the necessity of this extra (and automatically provided) parameter. The above process shows examples of the '`ChanInInt`' and '`ChanOutInt`' API calls, whose usage is mostly obvious.

1.1. Starting C Processes

To create an instance of the above '`integrate`' process requires a call to either '`ProcAlloc`' or '`ProcInit`'. To do this from occam-π requires the use of an external C call:

```
void real_make_integrate (Channel *in, Channel *out, Process **p)
{
    *p = ProcAlloc (integrate, 1024, 2, in, out);
}

void _make_integrate (int *ws)
{
    real_make_integrate ((Channel *)(ws[0]), (Channel *)(ws[1]),
                         (Process **)(ws[2]));
}
```

that can be called from an occam-π program after declaring with:

```
#PRAGMA EXTERNAL "PROC C.make.integrate (CHAN INT in?, out!, RESULT INT p) = 0"
```

The usage of this in occam-π is slightly peculiar since the call will return providing a process address in '`p`', but having already consumed its '`in?`' and '`out!`' parameters. An in-line occam-π procedure is provided by CIF that executes the C process, returning only when the C process has terminated — at which point it could be freed[1] using '`ProcAllocClean`'. For example:

```
#INCLUDE "cifccsp.inc"

PROC external.integrate (CHAN INT in?, out!)
  INT proc:
  SEQ
    C.make.integrate (in?, out!, proc)
    cifccsp.startprocess (proc)
:
```

Creating and executing C processes inside a CIF process is much simpler. Processes are created in the same way using '`ProcAlloc`', but are executed using '`ProcPar`' (or one of its variants).

It should be noted that the above two C functions, the entry-point '`_make_integrate`' and '`real_make_integrate`', could be made into a single function. Separating them out gives the parameters passed explicit names, however, instead of using indices into the '`ws`' array. The 'real' function can be declared '`inline`' to get equivalent performance if desired.

1.2. Masquerading as occam

In order to present themselves as occam-π processes, CIF processes need a valid occam-π process workspace. This is a fixed-size block that contains the state of the CIF process, in addition to the 'magic' workspace fields used for process control. Figure 1 shows the layout of this structure, with word-offsets relative to the '`Process`' pointer (equivalent to an occam process's *workspace-pointer*).

The workspace below offset 0 is that normally associated with suspended occam-π processes. These are used only when the CIF process is inactive, e.g. blocked on channel communication. The workspace offsets from 0 to 2 are used by CIF processes that have gone parallel and are waiting for their sub-processes to terminate, in the same way that occam-π processes do. The workspace offsets from 4 to 12 hold the CIF-specific process state, including the stored state of the run-time system when a CIF process is executing (held in processor registers for occam-π processes).

When a CIF process is initially created, its *entry-point* is set to the C function specified in the call to '`ProcAlloc`'. The *iptr* field is set to point at an assembler routine that starts the

[1]There seems little point in cleaning up after this '`integrate`' process, since it is not expected to terminate.

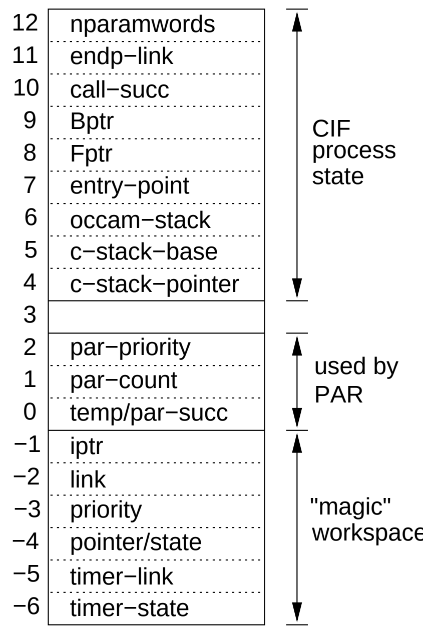


Figure 1. CIF process workspace

process for the first time and handles its shutdown. When a CIF process is blocked, the *entry-point* field holds the real 'return' address in the user's C code, whilst the *iptr* field points to an assembler routine that resumes the process. Figure 2 shows the life-cycle of a CIF process.

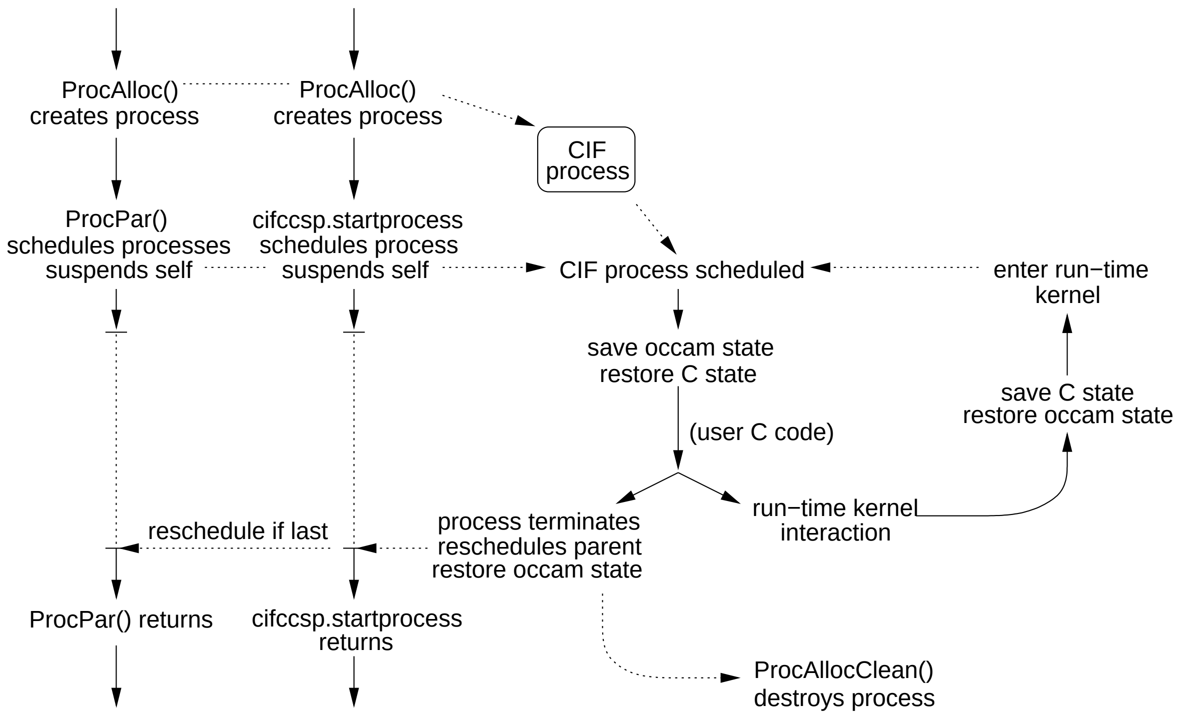

ProcAlloc() creates process
ProcAlloc() creates process
CIF process
ProcPar() schedules processes suspends self
cifccsp.startprocess schedules process suspends self
CIF process scheduled
enter run−time kernel
save occam state restore C state
save C state restore occam state
(user C code)
run−time kernel interaction
reschedule if last
process terminates reschedules parent restore occam state
ProcPar() returns
cifccsp.startprocess returns
ProcAllocClean() destroys process

Figure 2. CIF process life-cycle

When entering the run-time kernel, a CIF process must set up its workspace in the same way that an occam-π process would. Furthermore, it must also use the correct calling convention for the particular entry-point. In-line assembler macros are used to achieve this, containing code very similar to that generated by the KRoC translator, 'tranx86' [11]. The return-address (in *iptr*) is always to a pre-defined block of assembler, however, that restores the CIF process correctly when it is rescheduled.

As an example, the following shows the pseudocode for the ‘`ChanInInt()`’ assembler routine (placed in-line within the C code):

```
1: // chan : channel address (in register)
2: // ptr : destination pointer (in register)
chan-in-int (chan, ptr):
3: push (frame-pointer)
4: save-c-state
5: restore-occam-state
6: wptr[iptr] ⇐ global-resume-point
7: jump (_Y_in32, chan, ptr)
local-resume-point:
8: pop (frame-pointer)
```

There is a certain degree of unpleasantness in the actual assembler code. Much of it due to subtle differences in the way that different GCC versions handle in-line assembler macros such as these[2]. The actual kernel call here ‘`_Y_in32`’ expects to be called with the channel-address in the EAX register and the destination address in the EBX register. These are handled using *register constraints* (a GCC feature) in the assembler-C interface.

The assembler macros represented by ‘save-c-state’ and ‘restore-occam-state’ are implemented respectively with:

```
1: frame-pointer ⇐ wptr
2: wptr[c-stack-pointer] ⇐ stack-pointer
3: wptr[entry-point] ⇐ local-resume-point
```

and:

```
1: stack-pointer ⇐ wptr[occam-stack]
2: Fptr ⇐ wptr[fptr]
3: Bptr ⇐ wptr[bptr]
```

The first of these saves the globally visible ‘`cifccsp_wptr`’ variable (containing the workspace-pointer for the CIF process, ‘wptr’) in the EBP register, that holds the workspace-pointer of occam-π processes. The current stack pointer is saved inside the CIF workspace, along with the address at which the C process should resume. The second of these macros restores the occam run-time state, consisting of its stack-pointer (which is the actual C stack-pointer of the run-time system), and the current run-queue pointers (that are held in the ESI and EDI registers). Strictly speaking, the copying of ‘`cifccsp_wptr`’ to the EBP register is part of restoring the occam run-time state, but since these macros typically always follow each other, restoring EBP early results in more efficient code.

The actual return address of the CIF process, as seen by the run-time system, is the address of the ‘*global-resume-point*’. This is a linked-in assembler routine that performs, effectively, the inverse of these two macros, before jumping to the stored resume point.

1.3. Providing the API

The application interface and user-visible types are contained in the header file “`cifccsp.h`”. Files containing CIF functions need only include this to access the API. The various functions that make up the API are either preprocessor macros that expand to blocks of in-line

[2]This is not so much the fault of GCC, but rather certain distributions that included development (and potentially unstable) versions of GCC.

assembler (as shown above), or for some more complex operations (e.g. ‘`ProcPar()`’ and ‘`ProcAlt()`’), actual C functions provided by the CIF library.

The API includes the majority of functions available in the original Inmos C API and the CCSP API. Additional functions are provided specifically for new occam-π mechanisms, again a mixture of assembler macros and C functions. These include, for example, ‘`ProcFork()`’ to fork a parallel process (following the occam-π ‘`FORK`’ mechanism) and ‘`DMemAlloc()`’ to dynamically allocate memory.

A complete description of the supported API, and some basic examples, can be found on the CIF web-page [12].

In addition to the standard and extended API functions, four additional macros are provided to make external C calls. The first two of these are used to make blocking C calls, i.e. that run in a separate *thread* with the expectation that they will block in an OS system-call. The second pair of macros are used to make ordinary external C calls, but only for certain functions. For each macro pair, there is one that is used to call functions with no arguments, and a second to call functions with an arbitrary number of arguments. For example:

```
void do_write (int fd, const void *buf, size_t count, int *result)
{
    *result = write (fd, buf, count);
}

void my_process (Process *me, Channel *in, Channel *out)
{
    for (;;) {
        void *mobile_array[2];
        int fd, result;

        /* input INT descriptor followed by a MOBILE []BYTE
         * array of data.
         */
        ChanInInt (in, &fd);
        ChanMIn64 (in, mobile_array);

        BLOCKING_CALLN (do_write, fd, mobile_array[0],
                        (size_t)(mobile_array[1]), &result);

        DMemFree (mobile_array[0]);

        ChanOutInt (out, result);
    }
}
```

This process inputs an integer file-descriptor, followed by a dynamic mobile array from the ‘`in`’ channel, then writes that data to the given file-descriptor (typically a network socket). After the call the dynamic mobile array is freed, followed by communication of the underlying ‘`write`’ result on the ‘`out`’ channel.

The corresponding occam-π interface for ‘`my_process`’ would be:

```
PROTOCOL FD.DATA IS INT; MOBILE []BYTE:
PROC my.process (CHAN FD.DATA in?, CHAN INT out!)
```

It should be noted that ordinary CIF routines may not be used inside an external C call. For blocking calls (e.g. ‘`do_write()`’ in the above), code executes with a thread stack, not in the CIF process’s stack. For ordinary (non-blocking) external C calls, code may or may not execute in a thread stack. For example, the ‘`BLOCKING_CALLN`’ in the above could be replaced with:

```
EXTERNAL_CALLN (do_write, fd, mobile_array[0],
                (size_t)(mobile_array[1]), &result);
```

The decision of whether to run ‘do_write’ in the CIF process’s stack, or the occam-π run-time’s stack, depends on whether POSIX threads [13] are enabled. Where POSIX threads are *not* enabled (and the run-time system uses Linux’s native ‘clone’ thread mechanism), the above call will be reduced to just:

```
do_write (fd, mobile_array[0], (size_t)(mobile_array[1]), &result);
```

When POSIX threads are enabled, the call is redirected to a linked-in assembler routine, that performs the call on the occam-π run-time’s stack. This stack-switch is actually only required when the POSIX threads implementation stores thread-specific information in the stack, rather than in proessor registers. In this case it is relevant since the ‘write()’ call sets the global ‘errno’ value; however, the standard C library, in the presence of POSIX threads, re-directs this to a thread-specific ‘errno’ (so that concurrent system-calls in different threads do not race on ‘errno’). In cases where the POSIX threads implementation is built to store the thread-identifier in processor registers, locating this thread-specific ‘errno’ is no problem — and can be done safely when code is executing in a C stack. However, if POSIX threads are configured to use the stack to store thread-specific data, making the call from a CIF stack results in a crash (as the ‘pthreads’ code walks off the top of the CIF stack whilst looking for thread-specific data). Linux distributions vary in their handling of this, but it is arguably better to use spare processor registers for holding the thread identifier (avoiding the chance of false-positives in a stack search).

2. Applications

CIF has a potentially huge range of application. Generally speaking, it allows the programmer to interface C with occam-π in a naturally compatible way, i.e. channel communication and other CSP-style concurrency mechanisms [14]. Despite the safety and practicality of occam-π, there are some things which are still more desirable to program in C — particularly low-level interface code that typically deals with *pointers*, which occam-π does not support natively. Explicit pointer types (such as those found in C) create the potential for aliasing and race-hazard errors, requiring care on the programmer’s part.

One of the original motivations for CIF was in order to ease implementation of the ‘ENCODE.CHANNEL’ and ‘DECODE.CHANNEL’ compiler built-ins [15]. These transform occam channel communications into *address*,*size* pairs, using extended inputs to block the process outputting whilst the resulting address and size are handled. These “protocol converters” are necessary for implementing the KRoC.net infrastructure [8][3] — as well as other similar infrastructures — transforming application-level communications into something suitable for network communication. The standard implementation of ‘ENCODE.CHANNEL’ and ‘DECODE.CHANNEL’ is by means of tree re-writing inside the compiler, necessary because different channel protocols require different handling, for which run-time information is generally not available. Although the mechanism is fully sufficient for its intended uses, making it compatible with new occam-π types, e.g. a ‘MOBILE BARRIER’ [16], is non-trivial and time-consuming.

A generic implementation of ‘ENCODE.CHANNEL’ and ‘DECODE.CHANNEL’ in C is relatively simple, provided that information about the structure of the channel-protocol is available. Recent versions of the KRoC system have the option of including this information in generated code. In practice, this is only supported for mobile channel-types, since they pro-

[3]KRoC.net will be known as “pony” when released, to avoid confusion with a .net targeting KRoC.

vide a convenient place to store a pointer to the generated type-description block. Figure 3 shows an example of how a generic protocol decoder could be used with an occam-π application.

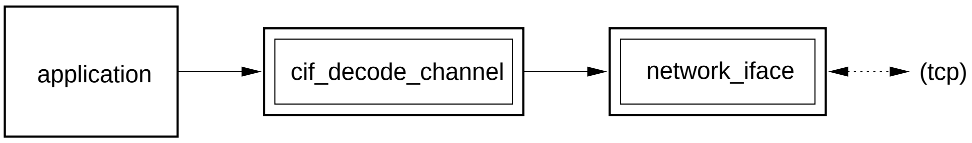


Figure 3. Generic protocol decoding in C

Unlike the compiler built-in versions of these protocol converters, the C implementations are substantially simpler. In the case of figure 3, the two C routines could be combined to a certain degree, providing a single CIF process that deals with networking of occam-π channels directly — such a mechanism would be non-transparent, unlike KRoC.net where transparency is key.

The following section presents a library that uses CIF processes to provide networked mobile channels. Each *channel-bundle* networked results in multiple encode/decode processes and the necessary infrastructure to support them.

2.1. Networking Mobile Channels

A simple mobile channel-type networking mechanism for occam-π is currently being developed. In particular it aims to facilitate the multiple-client/single-server arrangement of communication, of an arbitrary mobile channel-type. For example:

```
PROTOCOL REQUEST IS MOBILE []BYTE:
PROTOCOL RESPONSE IS MOBILE []BYTE:

CHAN TYPE APP.LINK
  MOBILE RECORD
    CHAN REQUEST req?:
    CHAN RESPONSE resp!:
:
```

Figure 4 gives an idea of what such a networked application might look like. New clients can connect to a server, and "plug-in" a client-end of the desired channel-type, provided they know where it is — i.e. host-name and TCP port. Unlike the KRoC.net infrastructure, this "application link layer" is unable to cope with the communication of mobile channel ends, that could alter the TCP 'wiring', and is beyond its scope in any case.

The implementation under development allows the user to specify different behaviours for the networked "virtual mobile-channel". In this example, and in order to operate as we intend, the infrastructure needs to know how communications on '`req?`' correspond with those on '`resp!`' — if at all. To a certain extent, this is related to how the shared client-end '`CLAIM`' gets handled. For the network shown in figure 4, application nodes will compete internally for access to the server, or will delegate that responsibility to the server. Which behaviour is chosen can affect performance significantly.

For instance, if each communication on '`req?`' is followed by a communication on '`resp!`', the client-end semaphore claim can remain local to application nodes — the server knows that whichever client communicated on '`req?`' will be expecting a response on '`resp!`', or rather, to which client the communication on '`resp!`' should be sent. However, if the application behaviour is such that communications on '`resp!`' can happen independently of those on '`req?`', the server needs to be aware of client-end claims, so that it knows which client to send data output on '`resp!`' to.

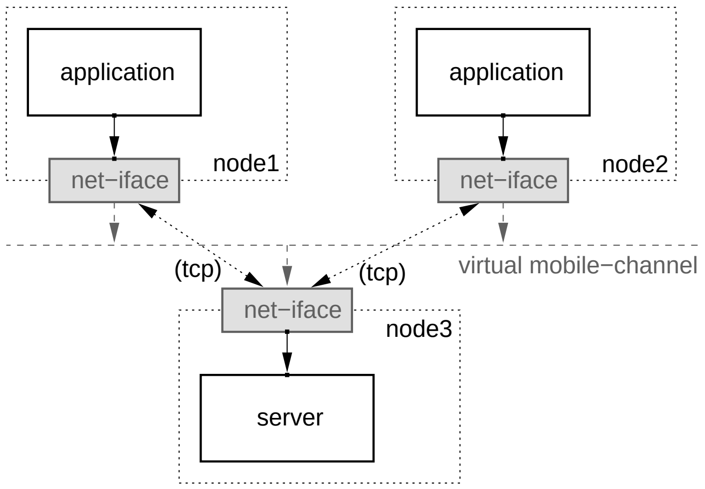


Figure 4. Networking any-to-one shared mobile-channels

The primary aims of this link-layer are simplicity and efficiency. To connect to a server using the above protocol, a client will use code such as:

```
SHARED APP.LINK! app.cli:
APP.LINK? app.svr:
INT result:
SEQ
  app.cli, app.svr := MOBILE APP.LINK
  all.client.connect (app.svr, "korell:3238", result)
  IF
    result = 0
      SKIP                   -- else STOP
  ...  code using "app.cli"
```

The call to '`all.client.connect`' dynamically spawns the necessary processes to handle communication, connecting to the server and verifying the protocol before returning. It is the server that specifies how communication is handled, for example:

```
SHARED APP.LINK! app.cli:
APP.LINK? app.svr:
INT result:
SEQ
  app.cli, app.svr := MOBILE APP.LINK
  all.server.listen (app.cli, "**:3238", "**(0 -> 1)", result)
  IF
    result = 0
      SKIP                   -- else STOP
  ...  code using "app.svr"
```

The string "`*(0 -> 1)`" is given as the usage-specification, stating that each communication on channel 0 ('`req?`') is followed by a communication on channel 1 ('`resp!`'), repeated indefinitely. These usage-specifications are essentially regular-expression style *traces* (for that channel-type only), and like the direction-specifiers are specified from the server point-of-view. Table 1 gives an overview of the supported specification language, in order of precedence.

The usage specification, in addition to controlling the behaviour of client-side '`CLAIM`'s, is used to build a state-machine. This state machine is used by client and server nodes to keep track of the current *trace* position. In particular, the infrastructure will not allow a communication to proceed if it not 'expected'.

Table 1. Supported usage-specification expression syntax

Syntax	Description	
`(X)`	sub-expression, where `X` is an expression	*most binding*
`*X`	`X` repeated zero or more times, where `X` is an expression	
`+X`	`X` repeated one or more times, where `X` is an expression	
`X | Y`	`X` or `Y`, where `X` and `Y` are expressions	
`n -> X`	`n` followed by `X`, where `n` is a channel index and `X` is an expression	
`n`	communication on `n`, where `n` is a channel index	*least binding*

The infrastructure comprising this "application link layer" is dynamically created behind the relevant client and server calls. Figure 5 shows the infrastructure created at the server-end, for the above '`APP.LINK`' channel type.

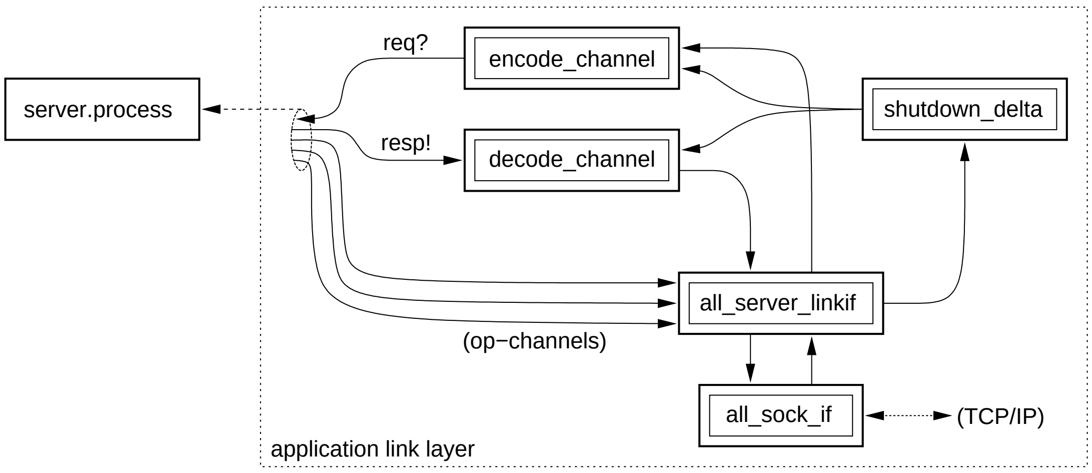


Figure 5. Server-side channel-type networking infrastructure

The three 'op-channels' emerging from the channel-bundle are specially inserted by the compiler, that generates communications on entry and exit from a '`CLAIM`' block, and when the channel-end is freed by the application (i.e. when it leaves scope). Programming this infrastructure in C makes easier the handling of dynamically created 'encode' and 'decode' processes. Internally, '`all_server_linkif`' `ALT`s across its input channels and processes them accordingly. The '`all_sock_if`' process is responsible for network communication and operates by waiting in a '`select()`' system-call, that allows it to be interrupted without side-effects, before reading or writing data.

The low-level protocol used by the current implementation *does not* respect occam-π channel semantics. Instead, the individual channels transported behave as buffered channels, where the size of the buffer is determined by the network and operating-system. This will be addressed in the future, once confidence in the basic mechanism has been established — i.e. successfully using CIF to transport occam-π channel-communications over an IP network. The current implementation is reliable, however.

A future implementation will likely use UDP [17] instead of TCP [18], giving the link-layer explicit control over acknowledgements, timeouts and packet re-transmission. Having available a description of channel usage enables some optimisations to be made in the underlying protocol, that are currently being investigated.

3. Conclusions and Future Work

The C interface mechanism presented in this paper has a wide range of uses, from providing low-level C functionality to occam-π applications through to supporting entire CSP-style applications written in C. Although CIF processes incur additional overheads (saving and restoring the C and occam states), these are not significantly damaging to performance.

The '`commstime`' benchmark is traditionally used to measure communication overheads in occam-π; it has been rewritten using CIF in order to get a practical measurement of the CIF overheads. On a 3.2 GHz Pentium-4, each loop for the occam-π commstime takes approximately 89 nanoseconds, 396 nanoseconds for CIF. This corresponds to a complete save/restore overhead of 26 nanoseconds, which will be an acceptable overhead for the majority of applications.

The current CIF implementation is not intended to be excessively efficient (i.e. in-lining of certain run-time kernel calls, as 'tranx86' optionally does). These will gradually appear in future releases of KRoC, as the C interface matures.

The one major drawback of the CIF interface is the inability of the C compiler to guarantee correct usage. This particularly applies to the handling of dynamic mobile types, whose internal reference-counts must be correctly manipulated. Incorrect handling can lead to memory-leaks, deadlocks and/or undefined behaviour (chaos). Despite this, it is hoped that users will find this C interface useful, for both its use with occam-π and as a software-engineering tool to apply CSP concurrency in C applications (e.g. migrating threaded C applications to a more compositional, and predictable/provable, framework).

References

[1] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[2] Inmos Limited. *occam2 Reference Manual*. Prentice Hall, 1988. ISBN: 0-13-629312-3.

[3] P.H. Welch, J. Moores, F.R.M. Barnes, and D.C. Wood. The KRoC Home Page, 2000. Available at: `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.

[4] M.D. May, P.W. Thompson, and P.H. Welch. *Networks, Routers and Transputers*, volume 32 of *Transputer and occam Engineering Series*. IOS Press, 1993.

[5] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 269–288, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

[6] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent at Canterbury, August 1998.

[7] F.R.M. Barnes. Blocking System Calls in KRoC/Linux. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures*, volume 58 of *Concurrent Systems Engineering*, pages 155–178, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[8] M. Schweigler, F.R.M. Barnes, and P.H. Welch. Flexible, Transparent and Dynamic occam Networking with KRoC.net. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 199–224, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

[9] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, The Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[10] Free Software Foundation inc. Using the GNU Compiler Collection (GCC), version 3.3.5, 2003. Available at: `http://gcc.gnu.org/onlinedocs/gcc-3.3.5/gcc/`.

[11] F.R.M. Barnes. tranx86 – an Optimising ETC to IA32 Translator. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 265–282, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[12] F.R.M. Barnes. The occam-pi C interface, May 2005. Available at: `http://www.cs.kent.ac.uk/projects/ofa/kroc/cif.html`.

[13] International Standards Organization, IEEE. Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language], 1996. ISO/IEC 9945-1:1996 (E) IEEE Std. 1003.1-1996 (Incorporating ANSI/IEEE Stds. 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995).
[14] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.
[15] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, June 2003.
[16] P.H. Welch and F.R.M. Barnes. Mobile Barriers for occam-pi: Semntics, Implementation and Application. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*. IOS Press, September 2005.
[17] J. B. Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, August 1980.
[18] J. B. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.