DYNAMICS AND PRAGMATICS FOR HIGH PERFORMANCE CONCURRENCY

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Frederick R. M. Barnes
June 2003

Abstract

This thesis is concerned with support at all levels for building highly concurrent and dynamic parallel processing systems. The CSP model of concurrency, as (largely) embodied in the occam programming language is used due to its simplicity, expressiveness, architecture-independent nature, and potential for high performance. Additionally, occam provides guarantees regarding freedom from aliasing and race-hazard error.

This thesis addresses one of the grand challenges of present day computer science: providing a software technology that offers the dynamic flexibility and performance of mainstream object oriented environments with the level of safety, formal analysis, modularity and lightweight concurrency offered by CSP/occam. Two approaches to this challenge are possible: do something to make the mainstream languages (e.g. Java, C++) safe, or make occam dynamic — without compromising its existing good properties. This thesis follows the latter route.

The first part of this thesis concentrates on enhancing the occam language and run-time system, on a commodity platform (IBM PC) running the freely available Linux operating system. After a brief introduction to the various components of the KRoC occam system, additions and extensions to the occam programming language and supporting run-time system are examined. These provide a greater degree of programming flexibility in occam (for example, by adding support for dynamic allocation, mobile semantics and dynamic network construction), without compromising the safety of programs which use them. Benchmarks are reported that demonstrate significant improvements in performance (for example, channel communication in tens of nano-seconds).

The second part concentrates on improving the level of interaction between occam programs and the OS environment. Providing easy access to sockets and networking, for example.

This thesis concludes with a discussion of the work presented herein, with consideration given to parallels with object-oriented languages. Also described are details of ongoing and potential future research. The modified language grammar, details of new compiler generated code, and miscellany are provided in the appendices.

Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

This thesis is the result of a little over three years work, carried out full-time in the Computing Laboratory at the University of Kent at Canterbury, funded by an EPSRC Ph.D. studentship.

I would like to acknowledge and thank my supervisor Professor Peter Welch, whose knowledge and motivation for all things parallel has provided me with much inspiration and many ideas over the years. I would also like to thank EPSRC (the U.K. Engineering and Physical Sciences Research Council) for providing the funding that made the bulk of this research possible, carried out within the UKC Computing Laboratory (specifically the Concurrency Research Group), to whom I am grateful for providing office-space, sometimes much equipment, and above all, a friendly academic environment.

In particular, thanks go to David Wood (responsible for the Sparc version of KRoC) for much insightful technical discussion, and to Jim Moores and Michael Poole, who built the initial KRoC/Linux components, and have provided me with much technical information over the years. Many thanks also to the Systems Group within the Computing Laboratory, who have accommodated many requests for system fiddling, hardware and have provided entertaining technical discussions from time to time.

During the pursuit of this Ph.D. I have visited CERN in Switzerland several times, to work within the EP-ATR group, responsible in part for the ATLAS (a particle detector) level-2 trigger system. In particular I would like to thank Bob Dobinson and Brian Martin, for accommodating my many visits there, that have provided much technical fun and excellent skiing.

Thanks also to the WoTUG community, whose technical conferences have been the outlet for various components of this work, and who have been an inspiring source of ideas and knowledge, in addition to providing valuable feedback. In particular, Brian Vinter and Adrian Lawrence.

Lastly, but by no means least, many thanks to my family and friends (Dave and Phill in particular), who have been wholly supportive throughout this endeavour, despite less frequent and shorter visits home.

This document was typeset in LaTeX, with figures and graphs prepared using Xfig and gnuplot.

Chapter 1

Introduction

This thesis, "Dynamics and Pragmatics for High Performance Concurrency" is concerned with the motivation, design and implementation of methods and mechanisms that provide support, at all levels, for highly concurrent dynamic parallel processing within occam/CSP. From a different viewpoint, the promotion of the occam multiprocessing language from rigorously static to highly dynamic, with a minimal impact on performance and preservation of occam's existing good properties (particularly freedom from race-hazard and aliasing errors).

The work within this thesis is undertaken at three levels: firstly, the occam language itself, to enhance the language in general and provide support for dynamic parallelism; secondly, the run-time kernel — the environment in which occam programs are executed — to provide the necessary run-time support for a dynamic occam language; and finally, the underlying operating-system, that can be extended to significantly improve run-time support for concurrent software systems.

1.1 History

The occam [Inm84a, Inm84b, Inm95, Bar92] programming language is a concurrent programming language based on the CSP [Hoa78, Hoa85, Ros97] process algebra. CSP provides a rigorous parallel model, whereby parallel processes engage in synchronous events. A process may also select between several events on offer (external choice). In occam, CSP events appear mostly as channels (where two processes synchronise and data is copied) and at the ends of PARallel blocks — to ensure that all nested parallel processes have terminated.

occam was developed between Oxford University and Inmos, with the Transputer [Inm93] in mind. The Transputer combined an on-chip micro-coded scheduler with on-chip external communications (on DS links), resulting in highly scalable multi-processor platforms for parallel processing. The T9000 Transputer additionally provided a virtual channel capability, allowing several virtual links to be multiplexed into one physical link [MTW93]. Coupled with crossbar routing, namely the C104, this provided a solid platform for the distribution and execution of occam programs. Sadly, the Transputer was shelved not long after the T9000 came into production.

The ideas and technologies underlying the Transputer still persevered in the research community however. The occam For All (oFA) EPSRC-funded project, that ran from February 1995 to May 1997, bought the benefits of the occam language (in the form of KRoC) to a range of architectures and platforms; Sparc, DEC Alpha, Motorola Power PC, Motorola 68HC11, Motorola 68000, Intel 80x86, Analog Devices 21060 SHARC, and MIPS. Like many high-level languages, occam itself is architecture independent. Its simplicity and freedom from side-effects (introduced through aliasing) make it a suitable language for many purposes, including hardware compilation [CP99], where the occam program becomes the architecture. With the reduction in cost and ever increasing size and density of Field Programmable Gate Arrays (FPGAs), building processors like the Transputer is a distinct possibility. Handel-C [APR+96] is one occam-based hardware language, incorporating a C-like syntax, arbitrarily sized integer types and various hardware-related facilities. Hardware compilation is now an active area of research, a good quantity of which is based on the CSP/occam model.

The Kent Retargetable occam Compiler (KRoC) [WW96] started life as a student project [RSS95] and after development in the oFA program of research, was developed into an occam system supporting the range of targets already mentioned. KRoC is comprised of an occam compiler, that generates an intermediate code, a translator to generate native code and a run-time kernel to provide Transputer functionality (at that time written in the target assembly language). KRoC/Linux originated here and was developed extensively at UKC [Poo96, WP97], incorporating a C-based run-time system (CCSP [Moo99]) and various extensions to the occam compiler itself [WM99].

The work presented in this thesis builds on KRoC/Linux as it was in September 1999. The native code-generator in this version of KRoC was '`tranpc`' — written in occam and translating from ETC binary [Poo98] to Intel i386 object-code [Int99] directly. Despite its originality, it turned out that this translator was too rigid to allow easy extensions and modifications, so during the course of this work a new translator was written — '`tranx86`', with ease of experimentation in mind.

1.2 Aims of This Work

The primary objective of this work is to provide support, at the language, kernel and operating-system level, for highly concurrent dynamic parallel systems based on occam/CSP.

This objective is sought in a number of ways. Firstly, by the extension and general enhancement of the occam programming language, largely by adding dynamic capabilities, plus a number of other (often trivial) extensions that bring it closer to languages such as C and Java (which rely heavily on dynamic memory allocation), whilst remaining secure against aliasing and parallel usage errors. Second, to provide support for data, channel and process mobility, using a movement semantics. And finally, by improving the interface between occam programs and the operating-system environment, allowing programmers to make full use of the UNIX/POSIX [Int96] environment.

As a further objective, this work aims to improve the maintainability and safety of occam code, particularly in light of the new facilities added.

1.3 Other Approaches to Compiling occam

This thesis is concentrated on the KRoC occam system. However, other ways of compiling occam programs exist. The most common of these is SPoC [DHWN94] — the Southampton Portable occam Compiler [1].

SPoC takes a different approach to compiling occam programs from KRoC. Instead of using the Inmos occam compiler, SPoC uses a compiler written (from scratch) using the GMD compiler-compiler toolkit [GE90], to produce a compiler that generates portable ANSI C code from the occam

[1] In a Google search for "occam compiler", circa October 2002, SPoC appeared at the top of the list, followed by KRoC in 2nd place. Currently, KRoC appears in first place, followed by SPoC in second. This change of ordering is most likely due to the addition of KRoC to the popular 'FreshMeat' website (`http://freshmeat.net/`), that acts as a public project-management and bulletin-board system, mostly for free software.

sources. The generated code can then be compiled on any suitable system, i.e. one with an ANSI C compiler.

High-Level Compilation

The advantage of using the GMD toolkit to build an occam compiler, besides its own portability, was for its algebraic-based semantic analyser. Given the occam language, this toolkit cannot analyse anything more than the Inmos occam compiler can — everything is pretty much static. However, for large replicated PARs, the job of parallel-usage checking is greatly speeded up when using the algebraic checker (inductive) rather than using static checking (exhaustive).

The version of the GMD toolkit used with SPoC suffered from a rather serious deficiency however — it was unable to correctly analyse the modulo (remainder after division) operator, with particular failings in the parallel usage checker. Mathematically, modulo is quite complicated, being relatively hard to reason about in algebra, but has a common use in occam for setting up fair-ALTs. Static analysis on modulo operators is trivial if the values involved are known. For the place where it matters — usage checking replicated PARs that setup a cycle of processes — broken analysis causes a problem. It should, in theory, be possible to fix this since the mechanics of SPoC's dependency checker (specifically the Omega-Test [Pug92]) can handle modulo to a certain extent; certainly enough to enable the correct checking of replicated PARs that setup a cycle of processes.

Various other failings have also limited SPoC's usefulness[2], many of which are covered by Singleton and Cook in [SC96]. Potentially, SPoC could be hugely successful as a general occam compiler, and it is somewhat unfortunate that this has not been the case.

Low-Level Compilation

Another approach to compiling occam, investigated by Poole in [Poo96] and by Welch and Poole in [WP97], is the direct targeting of the Inmos occam compiler to native code — the DEC Alpha architecture in this case. This proved to be a highly successful approach, and found good industrial use. Unfortunately, the Alpha line of processors has ceased production, with most people's choice of computer these days being either a PC (almost ubiquitous), or a large server solution (e.g. a Sun E450 / V880, SGI Octane-II, IBM RS6000) [3]. The technique of generating target architecture code directly from the occam compiler is still a good one, and removes the need for a native code translator. However, there is an obvious issue of portability, and from the point of software engineering, retargeting the (native-code) translator is likely to be significantly easier than retargeting the compiler.

One alternative technique for occam compilation would be the addition of an occam front-end for the 'gcc' compiler [Sta98]. The generation of an occam front-end for gcc would require a significant investment of time — centrally in order to learn the internals of gcc. Such porting is made easier, however, by the provision of good documentation and many existing front-ends for various languages: 'g++' for C++, 'g77' for Fortran-77, and more recently, 'gcj' for Java. It is envisaged that much of a (C-based) run-time system would still be required, to perform the scheduling of occam processes.

[2] Attempts to build SPoC cleanly (from the various versions on their website), largely failed when tried on a modern Linux system — often realised as an internal (SPoC occam) compiler error when compiling certain library files. Tracking these bugs down and fixing them is certainly possible, but as yet has not been attempted.

[3] There is an almost eerie parallel with the early days of computing — X-terminals and TTYs are now PCs, servers just got bigger and more powerful. The industrial philosophy is once again back in favour with the idea of having centrally managed servers, with clients as largely slave devices (database systems in particular). It made sense in the first place, and it still does now: many real-world interactions are client/server based.

New Approaches

One frequently discussed topic has been the possibility of developing a completely new occam compiler, using modern compiler tools and techniques. However, the existing Inmos occam compiler clearly demonstrates that compiling occam is a non-trivial task. This arises largely from the extensive checking required to ensure the correctness of parallel code, and the desire to limit the number of run-time checks to a minimum.

It is clear that a large amount of time and effort has been spent in developing the Inmos occam compiler, particularly with respect to the correct compilation of occam. For occam, there exists the "CG-tests", consisting of some 46 extensive test programs, that exercise various features of the occam language and compiler. Many of these tests contain occam code that is unlikely to be generated naturally by humans, but which is still legal occam.

For any new occam compiler, that would ideally pass all the CG-tests, the extensive parallel-usage and alias checks must be re-implemented. This is no small task, and one that is hard to get right. Although the Inmos occam compiler is certainly not simple, using it as a core to build upon makes good sense, given the alternatives.

1.4 Motivation

Parallel programming is too often seen as a "hard" discipline, and one area which the majority of programmers try to avoid. This has not come about through the non-availability of parallel hardware, such as the Transputer — current hardware is more than adequate — but through the lack of appropriate software tools and infrastructures to build concurrent systems. Traditional languages like C, and more recently Java, suffer when parallelism is "bolted on", frequently resulting in more harm than good. This arises because there is little or no control over how that parallelism is used — the programmer can write all sorts of race-hazardous code, often without realizing it. The problem of managing parallelism in these languages increases with the size of the system, leading to catastrophic failures in large systems that are almost impossible to pin down. Large non-parallel systems also suffer from such problems, that can easily be caused by variable/pointer aliasing errors.

The occam language, with CSP semantics for parallelism, solves the majority of these problems: occam does not permit uncontrolled aliasing and CSP provides composable semantics for building parallel systems. Despite this clear advantage, occam suffers from an inability to interact fully with the surrounding operating-system and hardware environments. The principal reason for this, in the KRoC occam system, is twofold. Firstly, adding significant amounts of code to hand-coded assembly language, of which the majority of KRoC run-time kernels are implemented in, is a difficult and daunting task. Secondly, the surrounding operating-system environment is often poorly adapted for fine-grain parallelism — for instance, executing a blocking system-call from within a (KRoC) occam process will cause all parallel occam processes to be suspended, while the program is blocked in the OS kernel. This makes writing programs that utilise inter-process communication (IPC) and networking difficult. Difficult to the point where it may be preferable to use C, or another language, and risk race-hazard and aliasing errors.

Another of occam's limiting factors is its lack of dynamic behaviour. Traditional occam programs can be viewed as static process graphs, nodes representing processes and arcs representing channels. Even though parts of a program may come into existence then disappear, all the graphs can be defined statically. One of the reasons for this static model stems from the Transputer, that had real finite memory (as opposed to virtual memory), and by today's standards, a much lower processing capacity — Transputers were designed to be assembled in networks to increase processing yield. Thus the use of dynamic behaviour in these systems would have been limited — a case of cost outweighing gain. Today's commodity computing hardware is massively more high-power by comparison, with processor speeds currently heading upwards of 3 giga-hertz and hundreds of mega-bytes of *real* memory commonplace. On this scale of hardware, the possibilities for dynamic behaviour in occam programs are enormous, providing of course that such dynamic behaviour is securely managed — in order to be free from race-hazards, uncontrolled-aliasing and memory-leak errors.

1.5 Structure of This Thesis

This thesis is divided into three main parts. The first part concentrates on the various additions and modifications made to the occam language and KRoC system to provide for dynamic parallel computing, presented across three chapters. Chapter 2 examines the the KRoC/Linux system, with a discussion of the major components and overall structure. General enhancements and additions to the occam language, that provide some basic dynamic capability, are presented in Chapter 3. Chapter 4 presents 'mobiles', an addition that provides mobility (a movement semantics) for data, channels and processes, and is in many cases dynamic. Chapter 5 examines various mechanisms that provide dynamic parallel process creation, a feature previously lacking in occam, and one that has always been highly desirable.

The second part of this thesis describes various other extensions that increase the level of (system) interaction available to occam program, and add support for facilities such as process priority and debugging, covered in Chapters 6 and 7.

The second part concludes in Chapter 8 with a discussion of the work presented in Chapters 3 through to 7, along with details of ongoing and possible future research.

The third part of this thesis consists of the appendices, which primarily serve as an implementation guide for future work. Appendix A provides an ordered syntax definition for occam, incorporating the various new language features, with cross-references to their descriptions in the thesis. Appendix B covers the various new instructions and translator-directives (ETC specials) which are needed, in part, to support the work presented in the first two parts of this thesis.

Part I

Dynamic Parallel Computing

This part of the thesis is concerned with the enhancement of the occam language, primarily to provide a dynamic capability for both data and processes. Additions to the language and run-time implementation of KRoC/Linux are examined, for Intel i386 based architectures.

Chapter 2 gives an introduction to the KRoC/Linux occam system, on which the majority of this work is based. The enhancements are split into a number of categories, each presented in a separate chapter.

A number of general extensions to the occam language, that are not specifically KRoC/Linux centric, are presented in Chapter 3. Chapter 4 presents *mobiles*, that provide a very general dynamic capability for occam, in particular a unit-time movement capability for data, channels and processes.

The first part of this thesis finishes with Chapter 5, that presents various enhancements specifically for dynamic process creation, greatly enhancing the expressiveness of occam for creating (dynamic) process networks.

Chapter 2

Introduction to KRoC/Linux

This chapter provides an introduction to the KRoC/Linux system, on which the work in this thesis has been realized. The major components of the system are described here, along with a brief history of its development.

2.1 Origins of KRoC/Linux

The current version of KRoC/Linux started life as a deliverable of the EPSRC funded oFA project [Pro95]. Michael Poole developed the native-code translator for the Intel 80x86 architecture (written in occam), as well as making several modifications and improvements to the Inmos occam compiler (which KRoC in general uses) [Poo96]. James Moores subsequently developed CCSP [Moo99], a run-time kernel supporting both C and occam, with was combined with the 80x86 translator, the occam compiler, various libraries, and a wrapper script, producing the KRoC/Linux system (largely work done by Dave Beckett and Jim Moores). The occam compiler work done by Wood and Moores [WM99] is also used in the Sparc version of KRoC (which also happens to be the first KRoC target).

The work presented in this thesis starts from the KRoC/Linux system as described above, and affects all components of the system — from the occam compiler and translator, to the run-time kernel. Further to this, consideration is also given to modifying the Linux operating-system to better support the occam run-time environment.

2.2 Packaging

The KRoC/Linux system is comprised of a number of separate programs, bound together by the '`kroc`' script, that performs the necessary steps to turn an occam source file into a target executable (linking in libraries and the run-time kernel along the way).

Figure 2.1 shows the pipeline of programs involved from turning an occam source file into a target executable. Source files are first compiled by '`occ21`', the Inmos occam 2.1 [Inm95] compiler. This has been modified by Poole to generate a binary intermediate language, ETC code [Poo98], which is code for the '*virtual transputer*' provided by KRoC. The compiler is described more fully in section 2.3.

The compiler-generated ETC is then translated into native object-code by the translator. In the original KRoC/Linux system, this was done using '`tranpc`', a translator written in occam that

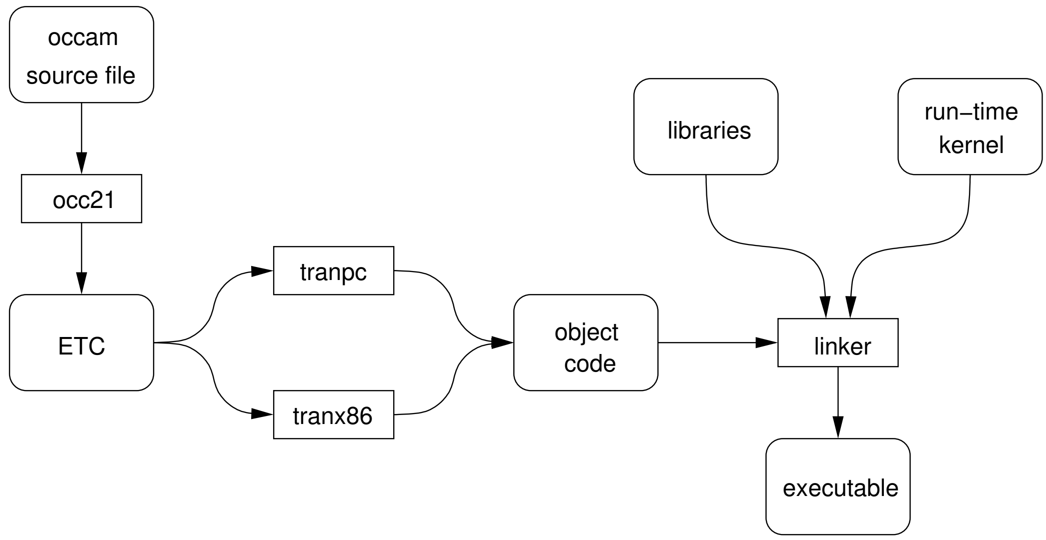


Figure 2.1: The KRoC/Linux compilation sequence

generated i386 object-code directly. This has some portability and maintainability problems, one of which is the inability to provide '`tranpc`' in a source-only distribution — due to need for a working `occam` system in order to build it. As such, a new translator, '`tranx86`' was written during the course of this work, providing many new features in the process. The new translator is described more fully in section 2.4.

Once compiled and translated, the object code is linked with various libraries and the run-time kernel to produce the target (native) executable. The run-time kernel used is a modified CCSP [Moo99], written mostly in C with some in-line native assembler for interfacing with `occam`. This run-time system is introduced more fully in section 2.5. Rather than calling the linker directly to create executables, KRoC uses '`gcc`' [Sta98] to invoke the linker, since the run-time kernel has a C '`main()`' (compiled with `gcc`). Additionally, `gcc` will adjust linker parameters where necessary for compatibility between Linux systems.

2.3 The occam Compiler — occ21

The `occam` compiler used in KRoC is adapted from the Inmos ToolSet compiler, "`occ21`". It is comprised of some 100,000 lines of C code, organised into the main compiler 'components' – front-end, tree-transformations, back-end and the compiler harness. Rather than using lexer and parser tools such as '`flex`' and '`bison`', the compiler is completely hand-written. Once parsed, the code inside the compiler operates on an *abstract tree* representing the code being compiled, which is modified as the various stages of compilation happen. The large amount of C `switch` statements, mostly on parse-tree node types, make the compiler rather tricky to maintain — a small change in the tree-structure requires many small changes throughout the compiler.

The non-use of tools such as `bison` is due in part to the difficulty in parsing `occam` in an LALR1 fashion. Additionally, a rather clever lexer is required in order to extract indent and outdent information from the `occam` code. As it turns out, incorporating the lexer and parser directly in C produce some nice performance advantages, especially when handling strings (the compiler maintains a global pool of *unique* words, making byte-by-byte string comparisons largely unnecessary).

The occam compiler targets a '*virtual-transputer*', based on the T800 Transputer [HMSS87, Inm88]. The output from the compiler is the previously mentioned ETC [Poo98]: binary code based on the T800 and T9000 Transputer instruction encodings, with various ETC '*specials*' that provide non-code information (such as entry-points, call-stubs, PROC memory usage information and constant data).

This output code has been extended to handle the new features in KRoC, particularly those associated with dynamic allocation. A description of the ETC additions can be found in Appendix B.

2.4 The Translator — tranx86

The tranx86 translator was developed during the course of this work [Bar01], replacing the original tranpc translator written by Poole. The function of the translator in KRoC is to turn the intermediate compiler output (ETC) into native code that interfaces with the run-time kernel (CCSP, discussed in section 2.5).

The new translator was designed with the Intel x86 family of processors in mind, hence the name. During the course of development, however, this target dependency has been localised with the intention of allowing easy modification to support alternative target architectures. An alternative code target for the MIPS architecture [ea82, Swe99] is currently underway — primarily supporting 32-bit MIPS targets, which Linux supports reasonably well [Bc01][1].

tranx86 maintains its own intermediate representation of code, with instructions based on the Intel Pentium-II (i686). These instructions carry significantly more information than can be expressed in assembly language, taking into account instructions that affect registers not specified in the instruction encoding. The internal code is build from lists of these instructions, using *virtual registers* — excessively — that are later allocated into real target registers, satisfying any constraints imposed by a particular target. In this back-end stage of the translator, CISC code generation (the originally intended target being the Intel i386 family) involves a reduction in the number of real registers used, and constraints for many instructions. RISC code generation, on the other hand, is more a reduction of the instruction set into simpler sequences, with fewer constraints on target register allocation.

Optimisation in tranx86 is performed largely in a *peephole* style, a common optimisation, and one used in a port of KRoC to the Power-PC [SARW98]. Such optimisations are somewhat target dependant, and may be affected by variations in processor revision. As such, they form part of the target-dependant part of tranx86, increasing the work required in porting. However, for experimental ports, optimisations can be skipped — in the interest of producing a functional KRoC system quickly.

As well as offering some architecture independence, tranx86 is geared to supporting different run-time kernels. Currently, support only exists for the original run-time kernel (CCSP), its modified successor (covered in the following section), and an experimental interface for RMoX [BJV03]. However, adding support for other run-time kernels is designed to be as simple as possible.

[1] Linux support for the MIPS architecture is in on-going development, but is both functional and usable. To date, the 32-bit support is largely complete, with 64-bit on the way. It turns out that a lot of the development time spent on Linux/MIPS is in handling uncommon processor variants — many such variants have been created over the years, often specific to their destined (typically embedded-systems) use.

2.5 The Run-Time Kernel — CCSP

The run-time kernel used by KRoC/Linux is a modified version of Jim Moores' CCSP run-time kernel [Moo99, Moo00b], that originally supported both C and occam (mutually exclusive). The C support has been largely removed from the version of CCSP in KRoC/Linux, with interfaces to C provided through different mechanisms (including one that allows the concurrent execution of both C and occam processes).

The primary function of the run-time kernel is to provide an implementation of the Transputer scheduling and communication mechanisms. Run-time kernels for other KRoC ports have traditionally been written in native assembly. This hinders maintenance somewhat, but in some — nowadays rare — cases, this may be unavoidable. The original CCSP contained a large amount of architecture-dependant in-line assembler code. This is still present in the current version, but has been separated out of the main kernel itself into architecture-specific header files (as pre-processor macros). In theory, porting CCSP to a different architecture requires only the writing of new architecture-dependant assembly macros, and that a '`gcc`' compiler suite (targeting the desired architecture) is available. Fortunately, architectures for which '`gcc`' does not exist are few and far between.

There are some advantages to writing run-time kernels in assembler, rather than writing them in C. For example, they are not at the mercy of C compiler bugs. During the development of KRoC/Linux, to its current state, multiple versions of `gcc` have been and gone. Along the way, versions have been encountered (particularly on old systems), that fail to compile the run-time kernel correctly. These bugs have typically appeared around the various in-line assembler macros, that break the logic of the C code quite substantially in places. Some (old) versions of the assembler encountered also caused problems, mainly in the non-support for certain instructions (typically those added to the architecture after the assembler was written). The current solution is to have the various configuration scripts in the KRoC source distribution attempt to identify and work-around any such deficiencies (before building the KRoC system components). Thus far, this approach has worked, resulting in a system that builds correctly on the majority of Linux configurations. OpenBSD [Reg02] is also supported by KRoC/Linux, with some of the particularly Linux-specific extensions missing.

Chapter 3

Extending occam

This chapter describes the various additions and extensions to the occam language and compiler, which in part facilitate the aims of making KRoC/Linux a more suitable language platform for deploying parallel systems.

The extensions are divided into three main groups. Sections 3.1 through 3.5 present extensions which are limited to changes in the occam compiler only — i.e. they do not generate new instructions. As such, many of these extensions are instantly portable to other KRoC targets — providing the necessary ETC-reading code is installed in the target code-generator.

The second group, sections 3.6 through 3.8, present the replicator `STEP`, the extended-rendezvous, and dynamic memory allocation extensions. The `STEP` and the extended rendezvous alter the occam language, and require support in the translator and run-time kernel for new instructions generated. The dynamic allocation extension (section 3.8) does not affect the occam language directly, requiring most changes in the translator and run-time kernel — largely for new instruction support.

The remaining extensions are described in sections 3.9 and 3.10. These cover a variety of minor compiler extensions and enhancements, plus a discussion of a C-style syntax for occam.

3.1 Channel Direction Specifiers

occam programs have traditionally been designed as process-networks, which provide a good (simple and intuitive) abstraction for reasoning about concurrent systems. Figure 3.1 shows an example '`integrate`' process network — for a running-sum integrator.

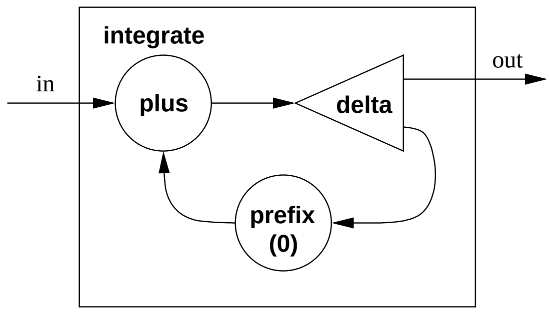


Figure 3.1: Process network for a running-sum integrator

From the diagram, the implementation is clear — create three internal channels and run the three sub-processes in parallel. In standard occam, this would be written:

```
PROC integrate (CHAN OF INT in, out)
  CHAN INT a, b, c:
  PAR
    plus (in, c, a)
    delta (a, b, out)
    prefix (0, b, c)
:
```

However, some information has been lost in the translation from design to implementation — the directions associated with the channels (shown using the arrow-heads in figure 3.1). The primary implication of this is that it prevents the easy translation from implementation back to design — in order to draw a diagram from the above code, one needs look at the implementation of the '`plus`', '`delta`' and '`prefix`' processes in order to determine the directions in which channels are used. This problem scales with the system too, given a simple diagram of '`integrate`' (not showing the internal construction), the code for both '`integrate`' and its sub-processes would need to be examined in order to deduce (fully correctly) the channel-direction usage.

Channel-direction specifiers present a simple solution to this, by including the "arrow-head" information in the code. This is certainly not a new idea however. Syntactically, the idea evolved from discussions in [Wel85]. In actuality, the channel-direction usage information is always known — but only internally to the compiler[1].

This extension exists in two main parts. Firstly, the addition of the necessary syntax within the language and secondly, the implementation of compile-time checks. The change in syntax is trivial — channel names are given an input ('`?`') or output ('`!`') postfix. The earlier '`integrate`' code, for example, now becomes:

```
PROC integrate (CHAN OF INT in?, out!)
  CHAN INT a, b, c:
  PAR
    plus (in?, c?, a!)
    delta (a?, b!, out!)
    prefix (0, b?, c!)
:
```

Whilst implementing this, a fairly minor change was made to channel declaration parsing: the '`OF`' keyword is now optional — as it serves no useful purpose in the language. Similarly, the '`AT`' and '`IN`' keywords are also now optional. Section 3.5.1 covers this slightly more formally.

3.1.1 Syntax Changes

The above example provides a good demonstration of the channel-direction specifier syntax. Within the (standard) language, there are three places where channel-direction specifiers may be used:

- on formal parameters in `PROC` headers:

```
PROC n.merge ([]CHAN INT in?, CHAN INT out!)
```

[1] Disassembling the compiler output will also provide this information — used by the compiler for handling separate compilation.

- on the left-hand-side of channel abbreviations:

```
CHAN INT c? IS ...:
```

- and in channel expressions:

```
CHAN INT local.out! IS out[0]!:
n.merge (chans?, local.out!)
```

The specifier can be used on both individual channels and on arrays/slices of channels. Where the whole channel name or a subscript of a channel array is used, the specifier is simply appended. For example:

```
[n]CHAN INT chans:
CHAN INT c:
PAR
  n.merge (chans?, c!)
  generate (chans[0]!)
  generate (chans[1]!)
  ...
```

Slices have a slightly different syntax. The most obvious syntax for these, following on from the above, would be:

```
PAR
  ...
  n.generate ([chans FROM 2]!)
```

However, this syntax is not supported by the compiler. Instead, this syntax is used:

```
PAR
  ...
  n.generate ([chans! FROM 2])
```

The reasoning behind this is that our syntax for array-*subscriptions* is actually wrong — it really ought to be:

```
PAR
  ...
  n.generate (chans![0])
  ...
```

When an array of channels is declared, such as with the '`[n]CHAN INT chans:`' declaration above, two arrays of channel-ends are conceptually declared — one of input ends and one of output ends. Thus the entire array of output-ends would be referred to as '`chans!`', and to select a single channel ('`i`') from within that, we *should* write '`chans![i]`'. Slices are technically correct: a range of ends from the appropriate array '`[chans!  FROM s FOR l]`'.

The array-subscription form is different since parsing '`out![0]`' as a *channel expression* is hard — not least because it complicates outputs/inputs. By placing the direction specifier at the end of the channel expression, it can be 'merged' into the action where input/output is performed directly — effectively omitting the specifier. For example, we can write:

```
SEQ
  chans[0] ! 42
  generate.int (chans[0]!)
```

Compare this with the technically more correct, but less natural, alternative:

```
SEQ
  chans![0] ! 42
  generate.int (chans![0])
```

Array-slices do not need to be considered for direct input/output, since an input or output process may only operate on a single channel, and slices always generate arrays (even if they contain a single element). However, array-slice expressions can be immediately followed by a subscription, for example:

```
[chans! FROM 3 FOR 2][1] ! 42
```

3.1.2 Compiler Checks

The processing of channel-direction specifiers in the compiler happens in two places. Early on in the compiler front-end (lexer and parser), channel-direction specifiers are extracted and turned into '`mopnode`'s (monadic operator nodes), using the tags '`ASINPUT`' and '`ASOUTPUT`' as appropriate.

Assuming the input parsed correctly, the compiler moves into the middle section of the front-end (scope and check). Here any '`ASINPUT`' and '`ASOUTPUT`' nodes in channel expressions are removed, checking direction compatibility in the process. As noted previously, there are only three places where direction specifiers may be placed: on formal channel parameters to a `PROC`, on the left-hand-side of a channel abbreviation, and in channel expressions ('channel' here includes channel arrays).

Direction specifiers in `PROC` *formal* parameters are simply ignored at this stage, since they are only meaningful when actual parameters are supplied.

For channel and channel-array abbreviations, the compiler checks that the direction specifier on the LHS matches any direction specifier on the RHS, either explicit or implicit. Implicit channel direction specifiers are found on the *declarations* of channels in the RHS. For example:

```
PROC foo ([]CHAN INT chans?)
  CHAN INT out! IS chans[0]!:
  ...  body of "foo"
:
```

This code is clearly wrong, but just by examining the LHS and RHS of the abbreviation it is not obvious that there is an incompatibility.

The checking of channel expressions occurs in two places. Firstly, when a channel is used (directly) for input or output, and secondly when a channel (or channel array/slice) is used as an actual parameter to a `PROC`. Checking in the first case is trivial — if a channel is used for output, it must be marked with `ASOUTPUT` and similarly, if a channel is used for input, it must be marked with `ASINPUT`.

Checking in `PROC` actual parameters is similar to checking in channel abbreviations — semantically they are the same thing (a renaming). Involved in the check is the direction-specifier in the `PROC`s corresponding formal parameter, any explicit direction specifier on the actual parameter and any implicit direction specifier on that actual parameter.

The compiler checks these three variables, each of which may have three states ('`ASINPUT`', '`ASOUTPUT`' or unspecified), then either allows the check to succeed (possibly generating a warning) or stops with an error. Table 3.1 shows the various combinations and their outcomes.

Formal	*input*	*output*	*unspecified*
Explicit			
input	(✔, ✘, ✔)	(✘, ✘, ✘)	(✔, ✘, ✔)
output	(✘, ✘, ✘)	(✘, ✔, ✔)	(✘, ✔, ✔)
unspecified	(✔, ✘, ✔)	(✘, ✔, ✔)	(✔, ✔, ✔)
	Implicit (*input*,*output*,*unspecified*)		

Table 3.1: Outcomes for channel-direction specifier compatibility checks

Some of the tests marked as successful (✔) may actually fail later on in the compiler, mainly when direction specifiers are missing (*unspecified*) and the *actual* usage of channels is incorrect — just as they would have failed previously.

Additionally, in strict-mode (section 3.9) any missing '*formal*' or '*explicit*' specifiers result in compiler errors — forcing channel-direction specifiers to be used wherever possible. It should be noted that separately compiled `PROC`s are not subject to the missing formal direction-specifiers check — since the programmer may not be in a position to alter them.

3.2 Extending PROTOCOLs

`PROTOCOL`s form the type-system for communication in occam. At the simplest level they may be simple data-types (such as `INT` or `BYTE`), user-defined types, or counted-arrays. Counted array protocols are used for communicating arrays, specified as '`DType::[]AType`', where '`DType`' represents the type associated with the size of the array (commonly `INT`) and '`AType`' represents the array element-type. At this level, encapsulating the protocol information in a `PROTOCOL` declaration is un-necessary, since channels can be declared to carry one of these basic protocols. For example:

```
PROC checksum (CHAN INT::[]BYTE in?, CHAN INT out!)
  ...
:
```

At the next level are sequential protocols, which are comprised of a series of simple protocols. These need to be placed in `PROTOCOL` definitions, if the sequence contains more than one simple protocol. For example:

```
PROTOCOL MESSAGE IS INT::[]BYTE:
PROTOCOL PACKET IS INT; INT; INT::[]BYTE:
```

At the most complex level are variant protocols. These allow a channel to carry an arbitrary number of sequential protocols, each one prefixed with a tag in order that the inputting process can select between the various sequential protocols on offer — or `STOP` if a particular case is not handled. Syntactically, these are defined with the `PROTOCOL` and `CASE` keywords (and as such are also known as "case protocols"), for example:

```
PROTOCOL LINK
  CASE
    message; INT::[]BYTE
    packet; INT; INT; INT::[]BYTE
    quit
:
```

The one feature this lacks is the ability to re-use an existing `PROTOCOL` definition in a later `PROTOCOL` definition — a feature which has been desirable for some time. There are three main parts to supporting this: sequential-in-sequential protocol inclusion, sequential-in-variant protocol inclusion, and variant-in-variant protocol inclusion. These are described in section 3.2.1.

As well as this, a new protocol *extension* (inheritance) mechanism has been implemented — which provides a rigid sub-typing mechanism for variant protocols. This feature is described in section 3.2.2.

3.2.1 Protocol Inclusion

Protocol inclusion is a relatively trivial extension which allows the re-use of user-defined `PROTOCOL`s within other `PROTOCOL` definitions. For sequential-in-sequential and sequential-in-variant inclusion, the name of previously defined protocol is used to represent its contents. For example:

```
PROTOCOL STRING IS INT::[]BYTE:
PROTOCOL PACKET IS INT; INT; STRING:

PROTOCOL LINK
  CASE
    packet; PACKET
    quit
:
```

This defines the protocol '`PACKET`' to be a sequence of two integers and a counted-array of bytes (the '`STRING`' protocol), and the '`LINK`' protocol to be a choice between the expanded '`PACKET`' (with the '`packet`' tag) and nothing (with the '`quit`' tag). The implementation of this inside the compiler is trivial — simply replace the name with the contents of the sequential protocol it represents. After processing, the compiler representation of the above is:

```
PROTOCOL STRING IS INT::[]BYTE:
PROTOCOL PACKET IS INT; INT; INT::[]BYTE:

PROTOCOL LINK
  CASE
    packet; INT; INT; INT::[]BYTE
    quit
:
```

Variant-in-variant protocol inclusion involves deeper analysis, in order to check that any tag *name* does not appear more than once after expanding inclusions. The syntax for this type of inclusion is slightly different, and uses the '`FROM`' keyword — in order to distinguish between an included protocol and an empty tag. For example:

```
PROTOCOL FOO
  CASE
    tag0; INT; INT
    tag1
:
```

```
PROTOCOL BAR
  CASE
    tag2; BYTE
    FROM FOO
:
```

Protocol inclusion, in all its forms, does not introduce any type-compatibility between the PROTOCOLs involved — they remain distinct. Protocol inheritance, described in the following section, provides a mechanism for sub-typing variant PROTOCOLs only.

3.2.2 Protocol Inheritance

Consider the following occam variant protocol declaration, for a generic GUI event handler:

```
PROTOCOL EVENT
  CASE
    expose; GADGET              -- redraw request
    resize; GADGET; INT; INT    -- resize notification
    mouse.click; GADGET; INT    -- mouse click
    mouse.enter; GADGET         -- mouse enter
    mouse.leave; GADGET         -- mouse leave
    key.press; GADGET; INT      -- key-press
    scroll; GADGET; INT         -- scroll event for scroll-bars
:
```

A protocol such as this can be used to describe the events sent between a GUI component and an event handler (either user-written or supplied in a library). Individual GUI components, however, might only be able to generate a small subset of these events. For example, a button component, rather than using the 'EVENT' protocol in its entirety, might just use a particular subset in a new protocol definition:

```
PROTOCOL BUTTON.EVENT
  CASE
    expose; GADGET              -- redraw request
    mouse.click; GADGET; INT    -- mouse click
:
```

As it stands, the 'EVENT' and 'BUTTON.EVENT' protocols have no type compatibility — i.e. we cannot connect a button component generating 'BUTTON.EVENT's to a generic event handler which takes 'EVENTS'. In single component, single event-handler systems, this will not normally be a limitation (the event handler can be restricted to a 'BUTTON.EVENT' interface), but when components *share* a common connection to a generic GUI event handler, it becomes an issue. Currently, there are two solutions to this problem. Firstly, drop component specific event protocols (such as 'BUTTON.EVENT') and modify component interfaces to output 'EVENT's. Or second, introduce a protocol-conversion component which transforms 'BUTTON.EVENT's into 'EVENT's — simply by copying input to output, as shown in Figure 3.2 [2].

[2] The aspect of channel sharing shown in figure 3.2 is not of primary interest here — a fair multiplexor process could be used instead. Channel sharing, and mechanisms for the automatic management of such are described in section 4.3.

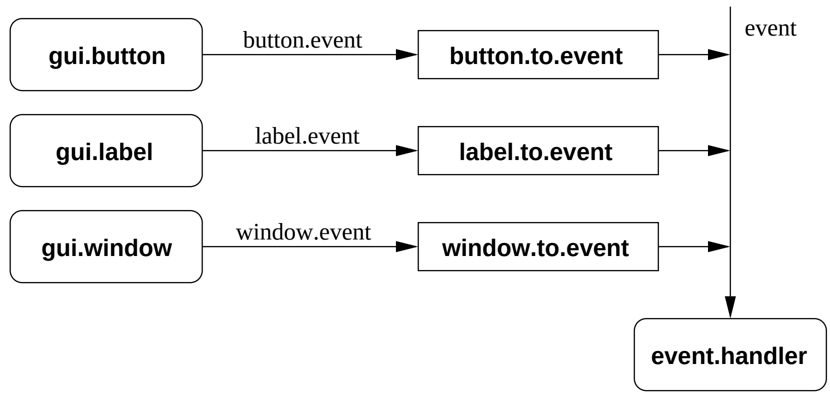


Figure 3.2: Using protocol conversion components to wire up a GUI

Neither solution is particularly attractive however. The former makes it possible for components to generate meaningless events (for example a label generating a '`scroll`' `EVENT`). The second requires the engineering (and connection) of specific protocol conversion components. Additionally, these converters must be modified whenever the component-specific protocol is altered.

A mechanism for protocol inheritance has therefore been introduced, that provides a limited type-compatibility between explicitly related variant protocols. The '`EVENT`' protocol declaration, above, can be modified such that it `EXTENDS` the '`BUTTON.EVENT`' protocol:

```
PROTOCOL EVENT EXTENDS BUTTON.EVENT
  CASE
    expose; GADGET              -- redraw request
    resize; GADGET; INT; INT    -- resize notification
    mouse.click; GADGET; INT    -- mouse click
    mouse.enter; GADGET         -- mouse enter
    mouse.leave; GADGET         -- mouse leave
    key.press; GADGET; INT      -- key-press
    scroll; GADGET; INT         -- scroll event for scroll-bars
:
```

This creates two distinct relationships in the protocol type-system. Firstly allowing a '`CHAN EVENT !`' actual parameter to be used where a '`CHAN BUTTON.EVENT !`' formal parameter is expected, and secondly, allowing a '`CHAN BUTTON.EVENT ?`' actual parameter to be used where a '`CHAN EVENT ?`' formal parameter is expected. The channel-direction restriction in these relationships prevents the potential non-handling of variant tags (which result in a run-time error). For example, the following is allowed:

```
PROC gui.button (CHAN BUTTON.EVENT out!)
  ...  body of gui.button
:

PROC event.handler (CHAN EVENT in?)
  ...  body of event.handler
:
```

```
CHAN EVENT events:
CHAN BUTTON.EVENT btn.events:
PAR
  gui.button (events!)          -- outputs BUTTON.EVENTs
  event.handler (events?)

  gui.button (btn.events!)
  event.handler (btn.events?)   -- inputs EVENTs
```

but this is not:

```
PROC gui.button (CHAN BUTTON.EVENT out!)
  ...  body of gui.button
:

PROC event.handler (CHAN EVENT in?)
  ...  body of event.handler
:

PROC gui.mystery (CHAN EVENT out!)
  ...  body of gui.mystery
:

PROC btn.event.handler (CHAN BUTTON.EVENT in?)
  ...  body of btn.event.handler
:

CHAN EVENT events:
CHAN BUTTON.EVENT btn.events:
PAR
  gui.button (events!)          -- outputs BUTTON.EVENTs
  btn.event.handler (events?)   -- INVALID: inputs BUTTON.EVENTs

  gui.mystery (btn.events!)     -- INVALID: outputs EVENTs
  event.handler (btn.events?)   -- inputs EVENTs
```

Protocol inheritance, when used in this way, appears to be a natural and intuitive technique in occam. This is reinforced both by the simplicity of the implementation (described in the following section) and a soundness in CSP.

3.2.3 Implementation Aspects of Protocol Inheritance

The for most part, the implementation of protocol inheritance is trivial. The internal representation of variant protocols inside the compiler needs to be modified slightly, in order to indicate those protocols which it extends. The nature of scoping in occam prevents cyclic inheritance, in the same way it prohibits mutual recursion[3].

In the most trivial cases, when one protocol extends another (singular), the compiler only needs to modify the tag values of the 'extended' protocol (the one which `EXTENDS` another). The compiler ensures that the sequential protocols attached to each tag are consistent, in cases where a tag is

[3]Self-recursion, on the other hand, is now supported by the compiler (described in section 5.1)

explicitly mentioned in the extended protocol. Any tags not explicitly mentioned in the extended protocol are automatically inserted by the compiler, keeping the tag values consistent.

Multiple protocol inheritance is also permitted, requiring only slightly more work in the compiler. In single inheritance, there is the guarantee that we can label the new protocol (with tag values) without affecting the inherited protocol. With multiple inheritance, this will generally not be the case (tag labelling in variant protocols starts at 0 and increases as tags are processed). To handle this, the compiler may need to re-assign tag values in one of the original protocols.

To avoid potentially serious problems, any variant protocol declaration requiring changes in another must reside in that same file. The re-assigning of tag values in one protocol may also require the re-assigning of tag values in any inherited protocols. In an attempt to minimise potential problems, the compiler, when presented with a choice of protocols to re-assign tag values in, will choose the least deep (in inheritance) and most local (in scope and defined in the same input file). Once a (locally declared) protocol has been extended (and possibly re-assigned), it is flagged such that it can never be re-assigned again. This prevents the 'hidden' changing of the tag values in an inherited protocol. If the compiler gets into a situation where it cannot satisfy the inheritance relations, it simply reports an error and stops.

Here is a short example showing multiple protocol inheritance (and the tag values assigned, then re-assigned where necessary):

```
PROTOCOL GRAPHIC.EVENT
  CASE
    expose; GADGET              -- tag = 0
    resize; GADGET; INT; INT    -- tag = 1
:

PROTOCOL MOUSE.EVENT
  CASE
    mouse.click; GADGET; INT    -- tag = 0 -> 3
    mouse.enter; GADGET         -- tag = 1 -> 4
    mouse.leave; GADGET         -- tag = 2 -> 5
:

PROTOCOL SCROLL.EVENT EXTENDS GRAPHIC.EVENT
  CASE
    scroll; GADGET; INT         -- tag = 2
:

PROTOCOL EVENT EXTENDS GRAPHIC.EVENT, MOUSE.EVENT, SCROLL.EVENT
  CASE
    key.press; GADGET; INT      -- tag = 5, MOUSE.EVENT re-assigned
:
```

3.3 RESULT Parameters and Abbreviations

Result parameters and abbreviations were first suggested by Barrett for occam3 [Bar92]. They are used to specify that a parameter or abbreviation is intended as a result, and as such need not be initially 'defined' — similar to the '`out`' parameter specifier in Ada [US 83]. The checks for 'definedness' states of `RESULT` parameters and abbreviations are implemented in the 'undefinedness-checker' (described in section 4.6).

Two implementations of RESULT abbreviations are supported by the compiler — one uses a reference to the original, the other a temporary and a copy. For example, the abbreviation:

```
RESULT INT r IS x:
P (r)
```

is turned in to an abbreviation by the first implementation:

```
INT r IS x:
P (r)
```

and into a temporary and a copy by the second implementation:

```
INT r:
SEQ
  P (r)
  x := r
```

The first implementation (a plain abbreviation) is used by default in the compiler, unless a special compiler flag '-zrv' is given. The second implementation has a potential advantage in cases where 'x' is a reference and the type is small — 'x' requires two instructions to load or store, 'r' requires only 1. Additionally, and perhaps more importantly, the second implementation causes 'r' to be *really* undefined initially — reading from it will not produce the value of 'x'. The checks are the same whichever implementation is used — the undefinedness checker will always consider 'r' to be initially undefined, regardless of whether it is an abbreviation or a fresh variable.

Result parameters are somewhat more natural than result abbreviations, as we often want to write PROCs which use some parameters specifically for output. Traditionally, this was not an issue, since the compiler did not perform undefinedness checks on the code — if you used an undefined variable, you got undefined results. Result parameters allow the compiler to check this behaviour safely, for example:

```
PROC populate (VAL INT n, RESULT []BYTE r)
  ...  populate r[0] through r[(SIZE r) - 1]
:
```

This indicates that the procedure 'populate' should expect its 'r' parameter to be undefined on entry and defined on exit. The compiler checks this and generates warnings as appropriate. For example, if the body of 'populate' does not actually populate 'r'.

3.4 Array-Constructors

Array-constructors are a feature commonly found in functional languages, such as Haskell [HPJW92, Tho99] and Miranda [Tur85, Tho95], the exception being that in these languages the constructors produce lists rather than arrays.

Concisely, array-constructors provide a method for building an array, based on *start*, *length* and *step* values, plus an expression which is computed for each element of the array. For example:

```
[10]INT vals:
SEQ
  vals := [i = 5 FOR SIZE vals | i + some.func(i)]
  ...  process using "vals"
```

This populates the whole 'vals' array, starting with the value '5 + some.func(5)' and finishing with '14 + some.func(14)'. In this example, the created array is simply assigned to the target variable ('vals'), and as such could be re-written (without the array-constructor) as:

```
[10]INT vals:
SEQ
  SEQ i = 5 FOR SIZE vals
    vals[i-5] := i + some.func(i)
  ...  process using "vals"
```

The underlying implementation of the array-constructor uses an occam value-process (VALOF, RESULT) — i.e. it generates an expression. This allows array-constructors to be used wherever an array (or slice) may be used. Newly added parts of the compiler test-suite exercise this to a pathologically complex level, for example:

```
[8]REAL64 vals:
SEQ
  vals := [[i = 15 FOR 16 STEP -1 | (REAL64 TRUNC i)]
               FROM [j = 0 FOR 8 | j][4] FOR 8]
  ...  process using "vals"
```

It should be noted that the 'STEP' is not specific to the array-constructor — it is a general enhancement to replicators and described in section 3.6.

The array constructor finds a natural use with the 'INTIIAL' specifier — that combines declaration and initialisation. For example, in:

```
INITIAL [16]REAL64 data IS [i = 0 FOR 16 | 0.0]:
...  process using "data"
```

Note that the size must be specified explicitly in the array-constructor. "SIZE data" would be invalid, since 'data' does not come into scope until the colon at the end of the INITIAL declaration.

3.4.1 Syntax and Transformation

The formal syntax definition for the array-constructor is as follows:

array.constructor = [*replicator* | *expression*]

With the two available forms of *replicator*, the RHS expands to:

= [*name* = *base* FOR *count* | *expression*]
| [*name* = *base* FOR *count* STEP *stride* | *expression*]

Semantically, this builds an array of constant length '*count*', whose elements are built from '*expression*' (which may involve '*name*'). The values of '*base*' and '*stride*' only affect '*name*', and have no effect on the size or order of construction. If '*stride*' is not given, its value is assumed to be 1.

Internally, the compiler transforms array-constructors into value-process (VALOF) expressions. These types of expressions are rarely used directly in occam programs, as they have a very peculiar syntax. Their most common occurrence is internally in the occam compiler from the inlining of FUNCTIONs (which are effectively named value processes). For example, the following declaration and array constructor assignment:

```
[20]INT vals:
SEQ
  vals := [i = start FOR (SIZE vals) STEP stride | func(i)]
  ...  process using "vals"
```

is transformed internally into:

```
[20]INT vals:
SEQ
  vals := ([SIZE vals]INT tmp:
           VALOF
             SEQ i = 0 FOR SIZE vals
               VAL INT tmp2 IS start + (i * stride):
               tmp[i] := func (tmp2)
             RESULT tmp
           )
  ...  process using "vals"
```

The first point to note is that '`start`' and '`stride`' have been moved out of the replicator and into the replicated expression. In this example a temporary is created to hold what would have been the value of '`i`' in the original expression. To handle this, occurrences of the replicator name ('`i`') in the original expression are substituted for either the temporary ('`tmp2`' as in this example), or the expression '`start + (i * stride)`' (where '`start`' and '`stride`' are simple[4]).

Array-constructors may also be used to create multi-dimensional arrays, formed in the expected manner, for example:

```
[32][32]INT64 matrix:
SEQ
  matrix := [i = 0 FOR SIZE matrix | [j = 0 FOR SIZE matrix[i] | gen.2d (i, j)]]
  ...  process using "matrix"
```

The resulting transformation for this code is remarkably unpleasant however — a nested `VALOF` process, but still legal occam:

```
[32][32]INT64 matrix:
SEQ
  matrix := ([SIZE matrix][SIZE matrix[0]]INT64 tmp:
             VALOF
               SEQ i = 0 FOR SIZE matrix
                 tmp[i] := ([SIZE matrix[i]]INT64 tmp:
                            VALOF
                              SEQ j = 0 FOR SIZE matrix[i]
                                tmp[j] := gen.2d (i, j)
                              RESULT tmp
                            )
               RESULT tmp
            )
  ...  process using "matrix"
```

[4]Simple expressions are those which can be evaluated using at most 1, 2 or sometimes 3 levels of the Transputer stack — the exact limit depends on the context, i.e. how deep the current evaluation stack already is.

The restriction that the replicator lengths must be constant simplifies code-generation somewhat, since all the generated 'SIZE' expressions will reduce to constants.

Where array-constructors are involved in subscriptions or slices, the compiler will simplify to produce new array-constructors or simple expressions. For example, the array slice:

```
[[i = 1 FOR 16 | 1.0 / (REAL64 i)] FROM 4 FOR 8]
```

is simplified to:

```
[i = (1 + 4) FOR 8 | 1.0 / (REAL64 i)]
```

with a run-time check to ensure that an array created in the original would have been large enough (since the original array-constructor length is removed from the expression).

The transformation for array-constructor subscriptions is somewhat simpler, removing the array-constructor entirely. For example:

```
[i = 1 FOR 16 | 1.0 / (REAL64 i)][x]
```

is simplified to:

```
1.0 / (REAL64 (x + 1))
```

with a similar run-time check to ensure that 'x' would have been within the bounds of the original array.

After such modifications, the affected part of the parse tree is processed again, to further reduce slices and subscriptions, eventually creating the resulting VALOF process.

3.5 Other Language Extensions

This section describes other, relatively small, extensions to the occam language. Section 3.5.1 describes, briefly, the optional 'OF' (and other similar keywords). Finally, section 3.5.2 describes the addition of zero-sized array support — something that occam traditionally bans.

None of the extensions presented in this section require work anywhere other than in the compiler. Extensions which are more widely-affecting (i.e. those requiring translator or run-time support) are presented in subsequent sections.

3.5.1 Optional OF

One of the endearing qualities of occam is its simplicity, both in syntax and semantics. On inspection of the occam grammar, most keywords serve a useful purpose. Some do not, however — most notably 'OF', 'AT' and 'IN'. As such, these have now been made optional, allowing a more succinct declaration of channels, for example 'CHAN INT c:' instead of 'CHAN OF INT c:'. The 'AT' and 'IN' keywords are used much less frequently — typically for allocating a variable 'AT' a particular address, or for forcing variable allocation 'IN' either workspace or vectorspace.

The compiler changes required to support this are minor. The 'OF', 'IN' and 'AT' keywords are always *expected* — and not used to determine choices in parsing. For example:

```
CHAN REAL64 c:              -- omitted OF
PLACE c VECTORSPACE:        -- omitted IN
CHAN REAL32 d:
PLACE d addr:               -- omitted AT
```

3.5.2 Empty Array Support

One slightly odd feature of occam is that it does not permit zero-sized arrays to be specified directly. Empty array literals '`[]`' and zero-sized array declarations '`[0]TYPE var:`' are rejected by the compiler. Whilst it is unclear why support for the latter would be useful, support for empty arrays as actual parameters would be useful. For example, a recently added `PROC` in the occam file-library (part of the *blocking system-calls* support — section 6.3), has the (partial) signature:

```
PROC file.select ([]INT read.set, write.set, except.set, ...)
```

This `PROC` provides access to the UNIX/POSIX '`select()`' system-call, used for IO multiplexing. Often, the application may only be interested in multiplexing reads, but the other arrays must still be specified. For this `PROC` in particular, which takes the arrays as reference (non-`VAL`) parameters, the programmer cannot simply write:

```
INITIAL [3]INT read.set IS [accept.fd, client[0], client[1]]:
SEQ
  file.select (read.set, [-1], [-1], ...)
```

Since the parameters must be *variables*, the constant array constructors are rejected by the compiler. Instead, and as well for cases where the array must actually be zero-sized (not consisting of a single ignored value, as in the above), the programmer must resort to something of the form:

```
INITIAL [3]INT read.set IS [accept.fd, client[0], client[1]]:
[1]INT write.set, except.set:
SEQ
  file.select (read.set, [write.set FOR 0], [except.set FOR 0], ...)
```

Having to resort to such things does not, in general, inspire confidence in programmers.

The compiler has now been modified to permit the use of '`[]`', for explicitly specifying empty array actual parameters — both `VAL` and non-`VAL` (reference). For example:

```
INITIAL [3]INT read.set IS [accept.fd, client[0], client[1]]:
SEQ
  file.select (read.set, [], [], ...)
```

As well as supporting this, the compiler needs also to allow the use of '`[]`' on the right-hand side of abbreviations (that may happen through `PROC` inlining). For example:

```
[]INT dummy IS []:
VAL []BYTE dummy2 IS []:
SEQ
  ...  inlined PROC body
```

Allowing '`[]`' for reference parameters may seem a little inconsistent — it is technically a constant, not a variable. However, '`[]`' is literally nothing, and any attempt to access its (non-existent) elements would result in a run-time error. This is divergence in CSP — and a divergent process can behave as *anything*. In effect, the elements of '`[]`' are $\perp$, therefore the type of '`[]`' is somewhat undefined — it can be every array type, including `VAL` and non-`VAL`.

This type compatibility extends to multiple dimensions, but specifying a constant array of one (empty) element is also valid. For example:

```
REAL64 FUNCTION weight (VAL [][]REAL64 m)
  ...
:

INITIAL REAL64 v IS 0:
SEQ
  v := weight ([])
  v := v + weight ([[]])
```

However, the use of '[[]]' would *not* be valid for a non-VAL parameter, since the outermost array contains one element.

3.6 STEP in Replicators

The ability to have a *stride* (step) in replicators has been a feature long missing from occam. In many cases, the same effect can be achieved using a VAL abbreviation, but providing it allows for a more natural specification of replicator strides.

The syntax used is simply the addition of the 'STEP' keyword and a '*stride*' expression to the replicator. For example:

```
SEQ i = 0 FOR n STEP s
  Q (i)
```

will replicate the instance 'Q(i)' 'n' times, starting with 'i' as 0 and incrementing by 's' each time. STEP expressions are valid in any of the four standard replicators (SEQ, PAR, ALT and IF) and in array-constructors (section 3.4). Formally, the syntax for a replicator is now:

replicator	=	*name* = *base* FOR *count*
	|	*name* = *base* FOR *count* STEP *stride*

A common place where having STEP is useful is when processing arrays, for example:

```
[N]REAL64 data:
SEQ
  ...  populate data
  PAR i = 0 FOR 4
    SEQ j = i FOR (N / 4) STEP 4
      ...  process element data[j]
```

which uses 4 parallel processes to work on the array 'data'. The first process works (sequentially) on elements $0, 4, 8, ...$, the second on $1, 5, 9, ...$ and so on[5]. If N is not exactly divisible by 4, the last 'N \ 4' elements will be ignored.

3.6.1 Supporting STEP at Run-Time

The compiler handles replicators by generating a 'LOOPEND' instruction. Inside the workspace of the process executing the replicator, two consecutive words are allocated, one for the replicator '*count*' and one for the value of the replicator '*name*'. The count-field is initialised to the number of

[5]Such code however, if run on multiprocessor hardware, would not be very efficient — due to the false sharing of cache lines (if the cache-line size is larger than occam's REAL64, which is certainly the case on Intel processors).

replications. The value-field is initialised to the replicator *base*. Each time the LOOPEND instruction is called, it subtracts 1 from the count field and adds 1 to the value field. If the count field has *not* reached zero, LOOPEND jumps to the start of the loop, otherwise it continues execution after the replicator.

To support a STEP at this level, the only change required is the ability to add or subtract an arbitrary value from the value-field — the count field is unaffected. To provide support for this, two new LOOPEND instructions have been added. The first of these, 'LOOPENDR', is used to implement STEPs of −1. This somewhat special case is often generated by the compiler when it reverses the processing order of a replicator (specifically for the modified ALT disabling — section 3.9.2).

The second new instruction, 'LOOPEND3', is for arbitrary STEP values. This is handled by placing the evaluated STEP expression alongside the count and name-value fields in the process' workspace. All the loop-end instructions take the workspace offset of this structure as an argument (along with the start-of-loop label and end-of-loop label). Figure 3.3 shows this workspace layout, and the code generated for a typical 'LOOPEND3' instruction.

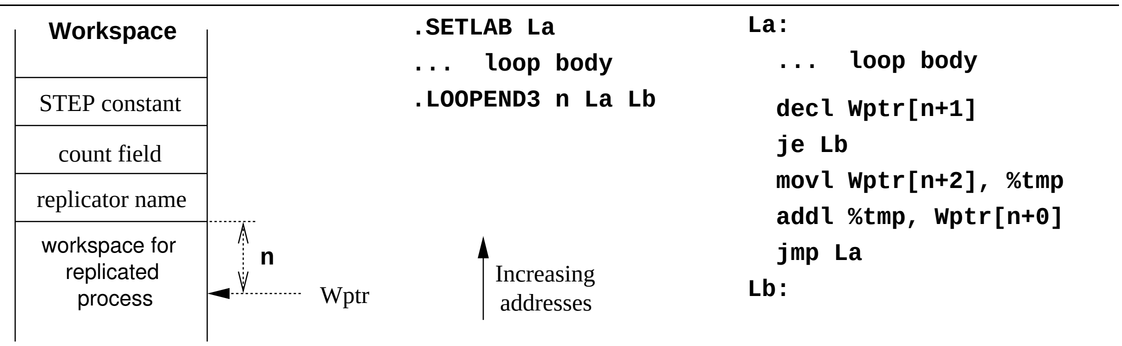


Figure 3.3: Workspace layout and code-generation for supporting arbitrary replicator STEP values

3.6.2 Loop-End Optimisation

One slightly frustrating feature of the loopend instructions is that they always generate code that modifies the replicator name[6], even if the name is never used in the replicated process. For example:

```
SEQ i = 0 FOR 10000
  INT x:
  c ? x
```

This generates code that uses the standard loop-end ETC special ('LOOPEND'). For the above, this (annotated) code is:

```
  LDC 10000              -- load replicator count
  STL 2                  -- store in count slot
  LDC 0                  -- load start value
  STL 1                  -- store in 'i'
L1:
  ...  code for declaration of "x" and input

  .LOOPEND 1 L1 L2       -- implicitly defines "L2"
```

[6]The replicator name is strictly not a variable — it cannot be assigned to

The translator generates a simple sequence of instructions for this form of ‘LOOPEND’, that takes as arguments: the workspace offset for the replicator specials (name slot and count slot); the label where the loop begins; and the label to define at the end of the loop. The (annotated) i386 code for this would typically be:

```
    movl $10000, 8(%ebp)        ; LDC 10000, STL 2
    movl $0, 4(%ebp)            ; LDC 0, STL 1
  L1:
    ...  translated for for declaration of "x" and input

    decl 8(%ebp)                ; decrement count
    je L2                       ; jump if finished
    incl 4(%ebp)                ; increment name "i"
    jmp L1                      ; jump to the loop start
  L2:                           ; end-of-loop label
```

Such generated code unnecessarily increments the name-value field (at an offset of 4 bytes for the above code). For very tight loops, being able to suppress this behaviour is highly desirable. Additionally, avoiding the name-value initialisation before the loop would represent a small saving.

Changes in the compiler and translator implement these optimisations. For cases where the replicated name is not used, initialisation code for it (evaluating and storing the start expression) is skipped. Preventing the update of the name-value is somewhat trickier — the loop-end instruction is not engineered for handling this, and adding more would be excessive. Instead, the compiler inserts a special comment before the loop-end instruction. The translator recognises this and avoids generating the increment. Furthermore, the conditional jump is inverted, reducing the loop-end code from 4 instructions to 2. The end-of-loop label (‘L2’ in the above example) might also be optimised away (if no other references to it remain).

Only the original ‘LOOPEND’ instruction is affected by this optimisation — in cases where the replicator name is not used, the compiler will always generate ‘LOOPEND’, regardless of any ‘STEP’. The compiler-generated and translated code for the above, with the loop-end optimisation, is:

```
    LDC 10000                                 movl $10000, 8(%ebp)
    STL 2                                   L1:
  L1:                                         ...  translated loop body
    ...  loop body
                                              decl 8(%ebp)
    .COMMENT .MAGIC UNUSED LOOPVAR            jne L1
    .LOOPEND 1 L1 L2                        L2:
```

Although the workspace slot for the name-value is completely un-referenced, it cannot be removed — other translators are free to ignore this magic comment (as described in section B.2.3). The ‘commstime’ benchmark is particularly affected by this optimisation, resulting in a reduction of up to 15% in the measured communication/context-switch time.

3.7 The Extended Rendezvous

The extended rendezvous is a powerful mechanism for *extending* channel synchronisation. Without requiring any changes in an outputting process, it allows an inputting process to execute code (the *extended process*), with the communicated data, whilst the outputting process remains blocked.

The extended rendezvous was an idea from Peter Welch originally [WB01], subsequently refined and investigated.

In order to accommodate the extended process, a new (extended) input operator is provided, '`??`'. In a similar way to `ALT` constructs, the input (guard) is followed by an indented process. There are two indented processes following the extended input operator: an extended process run whilst the outputting process is blocked, then a process run after the outputting process has resumed. This second indented process is optional, and assumed to be '`SKIP`' if not present. It is, however, required when the extended input operator is used in `ALT` guards or for `CASE` inputs — where an indented process is used normally. The various forms of extended-input syntax are covered in detail in section 3.7.1.

The key feature of the extended rendevous, and its implementation for occam, is that no modification of the outputting process is required. An inputting process may substitute a normal input for an extended input (and an extended process), without the outputting process's knowledge — i.e. an outputting process is unable to distinguish between either normal inputs or extended inputs (and the execution of the extended *input* process) on the other side. Semantically however, the outputting process has changed, as discussed in section 3.7.4. The modified semantics for the outputting process are handled transparently in the implementation, discussed in section 3.7.2.

3.7.1 Syntax

The extended rendezvous uses a new operator for extended input, '`??`'. This is followed by two indented processes — the first executed whilst the outputting process is blocked, and the second after the outputting process has resumed (and in parallel with it). Formally:

$$
\begin{array}{lcl}
\textit{extended.input} & = & \textit{channel}\ \texttt{??}\ \{_1;\ \textit{input.item}\} \\
 & & \quad \textit{process} \\
 & & \quad \textit{process} \\
 & | & \textit{channel}\ \texttt{??}\ \texttt{CASE}\ \textit{tagged.list} \\
 & & \quad \textit{process} \\
 & & \quad \textit{process} \\
 & | & \textit{channel}\ \texttt{??}\ \texttt{CASE} \\
 & & \quad \{\textit{extended.variant}\}
\end{array}
$$

Where the '*extended.variant*' is defined as:

$$
\begin{array}{lcl}
\textit{extended.variant} & = & \textit{tagged.list} \\
 & & \quad \textit{process} \\
 & & \quad \textit{process} \\
 & | & \textit{specification} \\
 & & \textit{extended.variant}
\end{array}
$$

An obvious example for the extended rendezvous is a *tap* process. Like '`delta`', it can be wired into an existing process network, but without affecting the synchronisation of that network — synchronisation is extended through the tap.

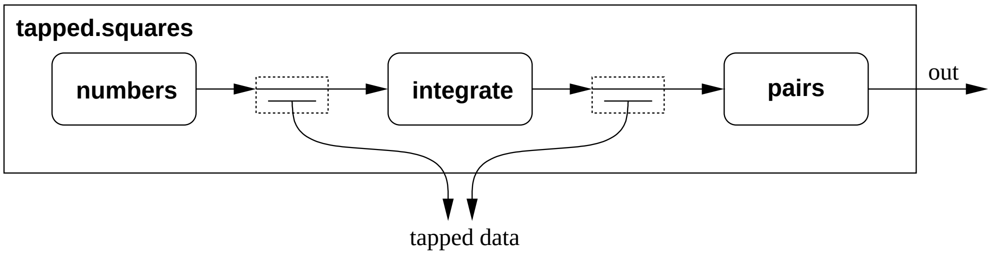


Figure 3.4: A tapped ‘`squares`’ process pipeline

Figure 3.4 shows the ‘`tap`’ process used to inspect the data flowing through the classic ‘squares’ process pipeline. Two equivalent implementations for this integer ‘`tap`’ are:

```
PROC tap (CHAN INT in?, out!, tapped!)
  WHILE TRUE
    INT v:
    in ?? v
      out ! v
      tapped ! v
:
```

```
PROC tap (CHAN INT in?, out!, tapped!)
  WHILE TRUE
    INT v:
    SEQ
      in ?? v
        out ! v
      tapped ! v
:
```

The syntax for extended `CASE` inputs follows suit — two indented processes following the extended input operator. For example, a tagged protocol type and ‘`proto.tap`’ process to handle it could be:

```
VAL INT max.message.size IS 256:

PROTOCOL MESSAGE
  CASE
    data; INT; INT; INT::[]BYTE
    error; INT
:

PROC proto.tap (CHAN MESSAGE in?, out!, tapped!)
  WHILE TRUE
    in ?? CASE

      [max.message.size]BYTE data:
      INT src, dst, d.len:
      data; src; dst; d.len::data
        out ! data; src; dst; d.len::data
        tapped ! data; src; dst; d.len::data

      INT e.code:
      error; e.code
        out ! error; e.code
        tapped ! error; e.code
:
```

Extending synchronisation for the purpose of tapping data in a process network is only one example of the extended rendezvous. Another potential use is in the creation of extended ‘`delta`’ and 'multiplex' processes. To demonstrate the use of the extended rendezvous in ALTs, a simple INT fair extended multiplex process could be:

```
PROC ext.plex ([]CHAN INT in?, CHAN INT out!)
  INITIAL INT fav IS 0:              -- current favourite
  WHILE TRUE
    PRI ALT i = fav FOR SIZE in
      VAL INT i IS i \ (SIZE in):    -- wrap around array
      INT v:
      in[i] ?? v
        out ! v
        fav := i + 1
:
```

3.7.2 Implementing the Extended Rendezvous

The primary reason for being able to add an extended input to occam is that the implementation requires no change in an outputting process. This is acheived by having an extended input masquerade as an ALTing process. When a channel output occurs, the run-time implementation checks the channel word for activity. If there is no inputting process waiting — the channel word is 'NotProcess.p' (defined as 'NULL' in KRoC/Linux) — then the outputting process places itself in the channel and reschedules. If the channel word is not 'NotProcess.p', then either an inputting or ALTing process is waiting. If an ALTing process is found in the channel word, it is placed on the run-queue, and the outputting process blocks in the channel and reschedules. This means that when a purely channel-based ALTing process wakes up, there will be an outputting process blocked in one of the channels. The standard channel output algorithm is shown in algorithm 3.1.

The algorithm used to implement the extended input consists of three main parts: an extended enable (which waits for the outputting process), the extended input (which copies the data without rescheduling the outputting process), and finally the extended end (which reschedules the outputting process and clears the channel word). A number of new virtual-Transputer instructions are required to support the extended rendezvous, listed in table B.3 and Appendix B.1.3.

The algorithm for the extended enable is shown in algorithm 3.2, implemented in the new 'XABLE' instruction. This is essentially a reduced form of a single-guarded ALT. If the outputting process is already blocked in the channel word, this simply does nothing. Otherwise, the process is disguised as a *waiting* ALTer and placed in the channel. When the outputting process arrives at the channel, it will find what it believes to be a waiting ALTer and reschedule it (at line 13 in algorithm 3.1)

After the 'XABLE' instruction has returned, the data is copied from the outputting process to the inputting process. This is trivial and is implemented by the 'XIN' instruction which, like the 'IN' instruction, takes the channel-address, destination-address and count as arguments. Once the data has been copied, the extended process is executed — whilst the outputting process remains blocked in the channel.

Once the extended process has finished, the outputting process is resumed. This is implemented by the 'XEND' instruction, whose operation is shown in algorithm 3.3.

The following (un-numbered) sections describe the different implementations for the extended input. Four seperate implementations are described: simple input, sequential protocol input, tagged protocol input, and the implementation for ALTs.

Algorithm 3.1: Standard channel output algorithm

// Wptr : *invoking workspace pointer*
// Iptr : *return address in occam process*

channel.output (chan.addr, src.addr, count):
$T_c \Leftarrow$ *chan.addr
if $T_c =$ NotProcess.p **then**
    // *channel empty*
    Wptr[Ptr] $\Leftarrow$ src.addr
    Wptr[$Iptr$] $\Leftarrow$ Iptr
    *chan.addr $\Leftarrow$ Wptr
    reschedule
else if $T_c[Ptr] \leqslant$ Ready.p **then**
    // *alting input process*
    if $T_c[Ptr] =$ Waiting.p **then**
        queue T_c
    end if
    $T_c[Ptr] \Leftarrow$ Ready.p
    Wptr[Ptr] $\Leftarrow$ src.addr
    Wptr[$Iptr$] $\Leftarrow$ Iptr
    *chan.addr $\Leftarrow$ Wptr
    reschedule
else
    // *committed input process*
    copy count **bytes from** src.addr **to** $T_c[Ptr]$
    queue T_c
    *chan.addr $\Leftarrow$ NotProcess.p
end if

Generating code for simple input

Generating code for handling extended simple inputs is relatively trivial, by contrast with its powerful operation. The following code fragments show example occam code and the compiler output generated, for ordinary input (left) and extended input (right):

```
INT x:
SEQ
  in ? x
  P (x)
```

```
LD ADDRESSOF x
LD ADDRESSOF in
LDC 4
IN
LD x
CALL P
```

```
INT x:
SEQ
  in ?? x
    P (x)
```

```
LD ADDRESSOF in
XABLE
LD ADDRESSOF x
LD ADDRESSOF in
LDC 4
XIN
LD x
CALL P
LD ADDRESSOF in
XEND
```

As can be seen, the code generated for an extended input is not much beyond that generated for an ordinary input — certainly less than would be generated for an `ALT` with one guard. Any following indented process (after the call to '`P`') is generated beyond the '`XEND`' instruction that resumes the outputting process.

Counted array input is not, strictly speaking, simple — it consists of two inputs, the count first then the actual array data. This is covered in the following section.

Algorithm 3.2: Extended rendevzous enabling algorithm

// Wptr : *invoking workspace pointer*
// Iptr : *return address in occam process*

extended.enable (chan.addr):
if *chan.addr = NotProcess.p **then**
  Wptr[*Ptr*] ⇐ Waiting.p
  Wptr[*Iptr*] ⇐ Iptr
  *chan.addr ⇐ Wptr
  reschedule
end if

Algorithm 3.3: Extended rendezvous end algorithm

extended.end (chan.addr):
queue *chan.addr
*chan.addr ⇐ NotProcess.p

Generating code for sequential protocol input

During the 'trans' phase in the compiler, sequential protocol communication is split into a sequence of individual communications. This is done to simplify code generation in the back-end of the compiler, and also to insert temporaries and run-time checks where needed. For example:

```
PROTOCOL S.PROTO IS INT; [2]REAL64:

PROC read (CHAN S.PROTO in?)
  [10]REAL64 data:
  INT v:
  SEQ
    c ? v; [data FROM v FOR 2]
    ...  compute with 'data'
:
```

It transformed, internally, into the (illegal) occam:

```
[10]REAL64 data:
INT v:
SEQ
  SEQ                              -- compiler inserted SEQ
    c ? v
    ASSERT ((v >= 0) AND (v < 9)   -- run-time check for valid 'v'
    c ? [data FROM v FOR 2]
  ...  compute with 'data'
```

When faced with an extended input on a sequential protocol channel, the compiler must generate code that perserves the semantics. Clearly, the only way is to have the last part of the communication extended, leaving the others as ordinary simple inputs. For example, the extended input:

```
c ?? v; [data FROM v FOR 2]
  ...  extended process
```

Is transformed into (the illegal):

```
SEQ
  c ? v
  ASSERT ((v >= 0) AND (v < 9)      -- run-time check for valid 'v'
  c ?? [data FROM v FOR 2]
    ...  extended process
```

A corresponding sequential-protocol output process will be transformed in a similar way, into a sequence of simple outputs. At run-time, all but the last item in the protocol will be communicated in the standard way (i.e. non-extended), with the last protocol item performed using the extended input — thus an outputting process will be blocked whilst the extended process runs.

Slight errors in this transformation can result in undesired deadlocks, or improperly extended communications. New programs added to the compiler test suite (CG tests) extensively test the extended rendezvous. In particular, tests exist that ensure that the outputting process remains blocked where it ought to be. Failure of such tests is indicated by deadlock, rather than explicit failure.

Generating code for tagged protocol input

Tagged protocol input is handled by first inputting the tag (as a `BYTE`[7]), then by using `CASE` selection on the tag-value, with the variant sequential protocols and corresponding process as processes in the '`CASE`' body. For example:

```
PROTOCOL TAGGED
  CASE
    empty
    some.int; INT
    more.ints; INT; INT
:

PROC read (CHAN TAGGED in?)
  in ? CASE

    empty
      ...  handle "empty"

    INT x:
    some.int; x
      ...  handle "some.int" using "x"

    INT x, y:
    more.ints; x; y
      ...  handle "more.ints" using "x" and "y"
:
```

The `CASE` input in the above is transformed into the following (illegal) occam:

[7] If the tagged protocol contains more than '`MOSTPOS BYTE`' tags, an `INT` will be used to handle the tag value instead.

```
BYTE $tag:
SEQ
  in ? $tag
  CASE $tag

    empty
      ...  handle "empty"

    some.int
      INT x:
      SEQ
        in ? x
        ...  handle "some.int" using "x"

    more.ints
      INT x, y:
      SEQ
        in ? x
        in ? y
        ...  handle "more.ints" using "x" and "y"
```

This transformation is substantially more complicated when the extended input is used, especially when handling tags without trailing sequential protocols (such as '`empty`' in the above example). The code generated for an extended-input version of the above (whose handling processes are extended) must perform the tagged input using an extended input, running the "`...  handle "empty"`" process if the tag was '`empty`'. However, for tags that have sequential-protocols, the outputting process must be released early and the extended-input performed on the last sequential protocol item (as described previously).

This is engineered in the compiler by the use of special '`FIRST.HALF`' and '`SECOND.HALF`' operators, internal to the compiler, that affect the code generated for a simple extended input. The '`FIRST.HALF`' operator generates code for the extended enable, data copy and the extended process, but stops the outputting process being resumed with the extended end — and will also skip any indented *after* (following) process. The '`SECOND.HALF`' operator performs the extended end, to resume the outputting process, but does not generate the code for any 'following' process (that must be done explicitly). For example, the following extended `CASE` input process:

```
  in ?? CASE
    empty
      ...  handle "empty"

    INT x:
    some.int; x
      ...  handle "some.int" using "x"

    INT x, y:
    more.ints; x; y
      ...  handle "more.ints" using "x" and "y"
:
```

Is transformed into the simpler (and illegal):

```
BYTE $tag:
FIRST.HALF in ?? $tag
  CASE $tag
    empty
      SEQ
        ...  handle "empty"
        SECOND.HALF in ??

    some.int
      INT x:
      SEQ
        SECOND.HALF in ??
        in ?? x
          ...  handle "some.int" using "x"

    more.ints
      INT x, y:
      SEQ
        SECOND.HALF in ??
        in ? x
        in ?? y
          ...  handle "more.ints" using "x" and "y"
```

When the input is extended in this way, an un-handled tag communicated by an outputting process will leave the inputting process `STOP`ed (at the '`CASE`' selection on the tag value). The outputting process will remain blocked, since the corresponding '`SECOND.HALF`' of the tag value input is never reached. This fixes a long-standing problem with unhandled protocol-less tags, where the outputting process would previously have been (incorrectly) rescheduled — technically, the inputting process never participated in the event. A generic implementation for this (i.e. for unhandled empty protocol tags on non-extended `CASE` inputs) is introduced in section 3.9.5.

Generating code for `ALT`ing inputs

The handling of extended inputs inside `ALT`ing constructs is only slightly more complicated than handling simple inputs. The implementation of the `ALT` is done such that when a blocked `ALT`ing process reschedules, one of the guards will be *ready*. For ready channel guards, this means that the outputting process is blocked in the channel word and waiting.

The code generated for an `ALT` is distinct from the code generated for any channel input within the `ALT`, so the `ALT` itself is unaffected by extended inputs. When an extended input is used as an `ALT` guard, it is known that the channel will be ready when the guard is selected. This allows for a slight optimisation — the extended enable ('`XABLE`', that waits for the outputting process), can be omitted, since the outputting process is already blocked in that channel.

3.7.3 Further Uses of the Extended Rendezvous

With the ability to extend synchronisation comes the ability to dramatically alter the way in which a network of communicating parallel processes behaves. Extended inputs can be *chained* together to extend synchronisation through a whole communication pipeline, which may be desirable for some applications. In some cases, such modifications will result in deadlock — so care must be taken in application design to avoid such conditions. Such deadlocks, resulting from the use of the extended rendezvous, can be analysed using standard tools such as FDR [For00] — given a suitable formal model of the extended rendezvous (section 3.7.4).

Multi-Way Synchronisation

One long-standing feature missing from occam is the *multi-way* ALT. Multi-way ALTs enable simultaneous synchronisation between multiple parties, as described in [WL98], that provides an implementation in Transputer assembly language[8].

Figure 3.5 shows a network of four processes, where 'meeting' performs a multi-way ALT. For example:

```
PROC meeting (CHAN INT in.p?, in.q?, in.r?)
  WHILE TRUE
    INT p, q, r:
    PRI ALT
      in.p, in.q, in.r ? p, q, r
        do.meeting ([in.p?, in.q?, in.r?], [p, q, r])
      in.q, in.r ? q, r
        do.meeting ([in.q?, in.r?], [q, r])
:
```

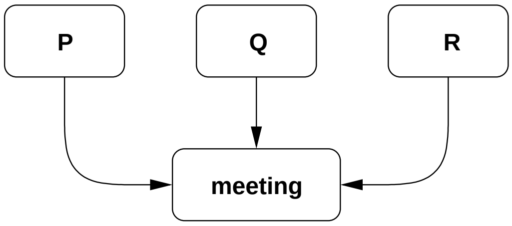


Figure 3.5: Multi-way synchronising processes

This ALT either synchronises with all three processes, or with just 'Q' and 'R', if 'P' is not ready. The semantics are effectively the merging of events involved, resulting in one for each multi-way synchronisation. Thus, either all four processes synchronise, or all but 'P' synchronise, with priority given to the former.

Without explicit language support for a multi-way ALT — as shown in the (illegal) occam fragment above — such synchronisations are only possible using user-provided code. The most obvious candidate is the 'BARRIER' user-defined type, described in [WW97]. However, there is no support for ALTing (or communication) using the BARRIER type. The crucially limiting factor is that processes, once they begin synchronisation on a BARRIER, are committed. More recent are ideas and a draft implementation for a 'SYNC' type [Wel03], that can support ALTs — but that still requires extensive compiler support for such.

The extended rendevous can also provide a mechanism for supporting multi-process synchronisation. However, support for ALTing is restricted. The proposed 'SYNC' type is symmetric — i.e. all

[8] Such functionality could be engineered in the current compiler, given suitable development time.

parties can ALT if desired[9]. The extended rendevous can simulate this multi-process synchronisation by extending communication between those processes. This can either be done by *chaining* or *nesting*, or a combination, of extended communications.

The earlier 'manager' process can be implemented using the extended rendevzous alone, without changing its operation. The following example uses a combination of both chaining and nesting, with the former requiring the addition of internal parallelism:

```
PROC meeting (CHAN INT in.p?, in.q?, in.r?)
  PROTOCOL TWO.INT IS INT; INT:
  CHAN TWO.INT c:
  PAR
    WHILE TRUE
      INT q, r:
      in.q ?? q
        in.r ?? r
          c ! q; r

    WHILE TRUE
      INT p, q, r:
      PRI ALT
        in.p ?? p
          c ? q; r
          do.meeting ([in.p?, in.q?, in.r?], [p, q, r])
        c ? q; r
          do.meeting ([in.q?, in.r?], [q, r])
:
```

A PAR-free implementation is possible too, but requires a moderately extensive ALT construct. The first parallel process in the above code extends communications from 'Q' and 'R' into a local channel. The second parallel process simply ALTs between a communication from 'P' and the local channel 'c'. If data arrives from 'P' before data from 'Q' *and* 'R', the first guard is selected — that then performs an extended input to collect the data from 'Q' and 'R' (via the local process), before completing the synchronisation. If 'Q' and 'R' communicate before 'P', the second guard will be selected. This synchronisation only completes when the (non-extended) input completes — since the corresponding output is within an extended input (in the first parallel process).

3.7.4 Formal Semantics

Two simple processes, P and Q, which synchronise on an event $e.i$ (typically a channel communication in occam), then perform some other actions (P' and Q'), could be defined with:

$$P = e!0 \rightarrow P'$$
$$Q = e?x \rightarrow Q'(x)$$

Let E be the set of events $\{e.i \mid i \in \mathbb{N}\}$, and D be another set of events on which P' and Q' synchronise, with $D \cup E = \emptyset$. The parallel composition of P and Q is then:

[9] The cost of a multi-process 'SYNC' is $O(n)$ when no ALTs are involved. The cost for ALTing SYNCs depends on the number of *false synchronisations* — where an ALTing process offers to synchronise then subsequently retracts that offer.

$$\begin{aligned}
&\big(P \mathbin{\underset{E\cup D}{\parallel}} Q\big) \setminus E \\
=&\big(e!0 \rightarrow P' \mathbin{\underset{E\cup D}{\parallel}} e?x \rightarrow Q'(x)\big) \setminus E \\
=&\big(P' \mathbin{\underset{D}{\parallel}} Q'(0)\big) \setminus E
\end{aligned}$$

The use of '$e?x$' and '$e!0$' serves to remind that this event $(e.i \mid i \in \mathbb{N})$ is channel communication from occam (with e representing the channel-name and i representing the different values communicated), and only involves two (or fewer) parallel processes. The hiding of '$e.i$' around the parallel processes is most accurately represented by a channel declaration in occam. This could be expressed as:

$$\begin{aligned}
&\mathit{CHANDECL}(e, P) = P \setminus \{e.i \mid i \in \mathbb{N}\} \\
\Longrightarrow &\mathit{CHANDECL}\big(e, (e!0 \rightarrow P' \mathbin{\underset{E\cup D}{\parallel}} e?x \rightarrow Q'(x))\big)
\end{aligned}$$

The same process can be expressed directly in occam, assuming P' and Q' are suitable PROCs, requiring only the addition of types:

```
CHAN INT e:
PAR
  SEQ
    e ! 0
    P.prime (D)
  SEQ
    INT x:
    e ? x
    Q.prime (D)
```

When executed, the pair of processes will synchronise on $e.0$, then continue as: $P' \mathbin{\underset{D}{\parallel}} Q'(0)$.

Modelling The Extended Rendezvous

Semantically, the use of an extended input in occam causes two synchronisations with the corresponding outputting process, with the extended process executed by the inputting process between the two synchronisations. The outputting process just synchronises twice.

Consider the same occam fragment, but where the inputting process performs an extended input, with the extended process '$Z(x)$', where x is the value communicated:

```
CHAN INT e:
PAR
  SEQ
    e ! 0
    P.prime (D)
  SEQ
    INT x:
    e ?? x
      Z (x)
    Q.prime (D)
```

The CSP representation of this code requires changes to both the inputting and outputting processes, in order to incorporate the extra synchronisation:

$$\left(e!0 \rightarrow e!0 \rightarrow P' \underset{E\cup D}{\parallel} e?x \rightarrow Z(x) \,\text{;}\, e?y \rightarrow Q'(x)\right) \setminus E$$

A way of capturing this is through a variant of the earlier '*CHANDECL*', using substitution to alter the processes. To explicitly specify an extended input, a new type of event interaction is added: "$e??_Z x$". Meaning: perform an extended input on the event '$e.x$' with '$Z(x)$' as the extended process. Thus:

$$\mathit{XCHANDECL}(e, P) = P' \setminus \{f.i \mid i \in \mathbb{N}\}$$

where P' is derived from P by applying all of the following substitutions:

$$\begin{aligned} e!i \rightarrow A &\implies f!i \rightarrow f!0 \rightarrow A \\ e?x \rightarrow B(x) &\implies f?x \rightarrow f?y \rightarrow B(x) \\ e??_Z x \rightarrow B(x) &\implies f?x \rightarrow Z(x) \,\text{;}\, f?y \rightarrow B(x) \end{aligned}$$

If P contains no extended inputs on e, then $\mathit{XCHANDECL}(e, P) = \mathit{CHANDECL}(e, P)$.

The original event is renamed in '*XCHANDECL*', primarily for clarity. The previous occam example, with an extended input, can be expressed in CSP (using the new extended-input syntax) as:

$$\mathit{XCHANDECL}\big(e, (e!0 \rightarrow P' \underset{D\cup\{e.i \mid i\in\mathbb{N}\}}{\parallel} e??_Z x \rightarrow Q'(x))\big)$$

Substituting for the declaration gives:

$$\left(f!0 \rightarrow f!0 \rightarrow P'' \underset{D\cup\{f.i \mid i\in\mathbb{N}\}}{\parallel} f?x \rightarrow Z(x) \,\text{;}\, f?y \rightarrow Q''(x)\right) \setminus \{f.i \mid i \in \mathbb{N}\}$$

that simplifies to:

$$\left(f!0 \rightarrow P'' \underset{D\cup\{f.i \mid i\in\mathbb{N}\}}{\parallel} Z(0) \,\text{;}\, f?y \rightarrow Q''(0)\right) \setminus \{f.i \mid i \in \mathbb{N}\}$$

Where P'' and Q'' are derived from P' and Q' respectively, using the given substitutions.

This model, although slightly crude, provides enough to reason about the extended rendezvous — allowing the construction of proofs that involve the extended rendezvous. The behaviour of substitution on CSP expressions is as expected, following the semantics of substitution within the Lambda-Calculus [Bar84]. The obvious care must be taken to avoid capture of names bound through hiding, specifically that:

$$P \setminus \{x\}[e/x] \longrightarrow P \setminus \{x\}$$

and:

$$P \setminus \{e\}[e/x] \longrightarrow (P[e'/e])[e/x] \setminus \{e'\}$$

Examples

A useful facility is the ability to prove that synchronisation is (conceptually) extended using this mechanism. Figure 3.6 shows a simple network in which the '`link`' process is implemented simply

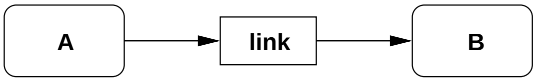


Figure 3.6: Example extended synchronising processes

by:

$$LINK(in, out) = \mu\, L \bullet \big(in??_{(out!x)}x \to L\big)$$

The processes represented by '`A`' and '`B`' are assumed to be trivial here, but could be quite complex:

$$A(out) = out!0 \to A'$$
$$B(in) = in?x \to B'(x)$$

Assuming positive integer communication, the whole network, as a closed system, can be described as:

$$XCHANDECL\Big(c, XCHANDECL\big(d, A(c) \underset{\{c.i|i\in\mathbb{N}\}}{\|} LINK(c,d) \underset{\{d.j|j\in\mathbb{N}\}}{\|} B(d)\big)\Big)$$

If the synchronisation is extended correctly, this network should be equivalent to a similar one where '`A`' and '`B`' are directly connected:

$$CHANDECL\big(c, A(c) \underset{\{c.k|k\in\mathbb{N}\}}{\|} B(c)\big)$$

Starting from the first and expanding out the internal processes gives:

$$XCHANDECL\Big(c, XCHANDECL\big(d, c!0 \to A' \underset{\{c.i|i\in\mathbb{N}\}}{\|} \mu\, L \bullet (c??_{(d!x)}x \to L) \underset{\{d.j|j\in\mathbb{N}\}}{\|} d?y \to B'(y)\big)\Big)$$

Taking the innermost '*XCHANDECL*' and substituting for its definition gives:

$$\Big(c!0 \to A' \underset{\{c.i|i\in\mathbb{N}\}}{\|} \mu\, L \bullet (c??_{(d'!x \to d'!0)}x \to L) \underset{\{d'.j|j\in\mathbb{N}\}}{\|} d'?y \to d'?z \to B'(y)\Big) \setminus \{d'.j \mid j \in \mathbb{N}\}$$

Placing this back into the '*XCHANDECL*' for 'c' and substituting gives:

$$\Big(c'!0 \to c'!0 \to A' \underset{\{c'.i|i\in\mathbb{N}\}}{\|} \mu\, L \bullet \big(c'?x \to (d'!x \to d'!0) \,;\, c'?v \to L\big) \underset{\{d'.j|j\in\mathbb{N}\}}{\|} d'?y \to d'?z \to B'(y)\Big)$$
$$\setminus \{c'.i \mid i \in \mathbb{N}\} \cup \{d'.j \mid j \in \mathbb{N}\}$$

Removing the sequential composition inside the recursive process gives:

$$\Big(c'!0 \to c'!0 \to A' \underset{\{c'.i|i\in\mathbb{N}\}}{\|} \mu\, L \bullet \big(c'?x \to d'!x \to d'!0 \to c'?v \to L\big) \underset{\{d'.j|j\in\mathbb{N}\}}{\|} d'?y \to d'?z \to B'(y)\Big)$$
$$\setminus \{c'.i \mid i \in \mathbb{N}\} \cup \{d'.j \mid j \in \mathbb{N}\}$$

Returning to the simplified network, without the middle '`link`' process, substituting '*CHANDECL*', '`A`' and '`B`' gives:

$$\begin{aligned}
&\mathit{CHANDECL}\big(c, A(c) \mathop{\|}_{\{c.i|i\in\mathbb{N}\}} B(c)\big)\\
=&\mathit{CHANDECL}\big(c, c!0 \to A' \mathop{\|}_{\{c.i|i\in\mathbb{N}\}} c?x \to B'(x)\big)\\
=&\big(c!0 \to A' \mathop{\|}_{\{c.i|i\in\mathbb{N}\}} c?x \to B'(x)\big) \setminus \{c.i \mid i \in \mathbb{N}\}
\end{aligned}$$

The laws of CSP allow the arbitrary addition of parallel processes. Furthermore, the concept of *subordination* allows the addition of a *slave* process, whose interactions are hidden from the common environment. Viewed independently, the left and right-hand parallel processes produce the same initial trace: $\langle c'.0\rangle$, before continuing as $(A' \parallel B'(0))$.
From the simple rule that:

$$R \mathbin{;} S = \big(R \mathbin{;} e \mathop{\|}_{\{e\}} e \to S\big) \setminus \{e\}$$

it can be deduced that the following processes are also equivalent (by letting R and S be $(c?x)$ and $P(x)$ respectively):

$$c?x \to P(x) = \big(c?x \to e!x \mathop{\|}_{\{e.i|i\in\mathbb{N}\}} e?y \to P(y)\big) \setminus \{e.i \mid i \in \mathbb{N}\}$$

and also:

$$c?x \to c?y \to P(x) = \big(c?x \to e!x \to e!0 \to x?v \mathop{\|}_{\{e.i|i\in\mathbb{N}\}} e?y \to e?w \to P(y)\big) \setminus \{e.i \mid i \in \mathbb{N}\}$$

Using this, the simpler network can be expanded internally into:

$$\begin{aligned}
&\Big(c'!0 \to c'!0 \to A' \mathop{\|}_{\{c'i|i\in\mathbb{N}\}}\\
&\quad\big(\mu\, L \bullet (c'?x \to d'!x \to d'!0 \to c'?v \to L) \mathop{\|}_{\{d'.j|j\in\mathbb{N}\}} d'y? \to d'?w \to B'(y)\big) \setminus \{d'.j \mid i \in \mathbb{N}\}\\
&\hspace{20em}\Big) \setminus \{c'.i \mid i \in \mathbb{N}\}
\end{aligned}$$

This can be re-arranged (with event renaming) to give an equivalent expression to the one that included the explicit link process:

$$\begin{aligned}
&\Big(c'!0 \to c'!0 \to A' \mathop{\|}_{\{c'.i|i\in\mathbb{N}\}} \mu\, L \bullet \big(c'?x \to d'!x \to d'!0 \to c'?v \to L\big) \mathop{\|}_{\{d'.j|j\in\mathbb{N}\}} d'?y \to d'?z \to B'(y)\Big)\\
&\hspace{20em}\setminus \{c'.i \mid i \in \mathbb{N}\} \cup \{d'.j \mid j \in \mathbb{N}\}
\end{aligned}$$

Thus, the two processes are equivalent — the extended rendezvous really does extend synchronisation.

3.8 Dynamic Memory Support for occam

Dynamic parallel computing is one of the primary aims of this work. In all current mainstream languages, dynamic behaviour is made available through dynamic memory management. For the C language, dynamic memory is available through the use of '`malloc()`' and '`free()`', largely governed by POSIX [Int96]. C++ [Str97a] provides specific operators for allocating and releasing objects — the '`new`' and '`delete`' operators. Traditional C '`malloc()`' and '`free()`' dynamic memory handling is still available, but the two are not compatible — e.g. '`delete`' should not be used to free memory allocated using '`malloc()`'. Such mixing can have disastrous effects.

Java [JGS96] provides dynamic memory management through the '`new`' operator only (to create new objects), leaving it up to the garbage collector to recover lost memory (objects). For C and C++, memory leaks represent a serious threat — especially for programs that remain active for a long time[10].

The dynamic memory provisions for occam strive to overcome some of these problems. Perhaps the most significant is that the occam programmer is not given direct access to dynamic memory management. Instead, the occam programmer has available numerous dynamic paradigms, provided through specialised syntaxes and semantics. The language mechanisms that *require* the availability of dynamic memory management are covered in Chapters 4 and 5.

Dynamic memory management in occam is provided by an allocator integrated into the run-time kernel. This is accessed at run-time using various new Transputer instructions. The allocator is based on one described by Brinch-Hansen in [Han95] and [Han96], pooled using (approximately) half-power of two grouping, the smallest being 4 bytes (then, $6, 8, 12, 16, 24, \ldots, 2048, 3072, 4096, \ldots$), up to a maximum of 1.5 giga-bytes. A similar memory allocator was previously implemented by Wood (in [Woo00]) to support recursion in the Sparc version of KRoC.

For occam programs, dynamic memory management is either implicit (from the language enhancements which require it), or explicit (in the form of in-line Transputer `ASM`).

The new instructions for allocation are '`MNEW`' and '`MALLOC`', and for freeing, '`MFREE`' and '`MRELEASE`'/'`MRELEASEP`' respectively. Appendix B.1.1 covers these instructions in detail.

These instructions are available to programmers writing inline `ASM` blocks — for use in highly specialised code, such as that implementing the semaphore, bucket, barrier and CREW user-defined types [WW97]. As a general rule, programmers should restrict themselves to the '`MALLOC`' and '`MRELEASE`' instructions. The '`MNEW`' and '`MFREE`' instructions offer a greater potential for error (if the slot number is specified incorrectly). The '`MRELEASEP`' instruction is a combination of '`MRELEASE`' and '`STOPP`', used for freeing a dynamic workspace — and this should be left to the compiler.

As a small example, the following shows part of an arbitrary user-defined type that uses dynamic memory, along with initialisation and release code (that would normally reside in a '`#INCLUDE`'d file):

```
DATA TYPE MYTYPE
  RECORD
    INT addr:          -- dynamic address
    INT size:          -- size in bytes
    ...  other fields
:
```

[10] It is possible, in UNIX, to engineer applications that can overcome memory leak problems. This is achieved by having the application periodically spawning a new copy of itself (in a suitably fresh environment), then transferring the active state of the current process into the new process. A clever trick to achieving this involves using a pipe to transfer the state between the two processes (file-descriptors can be communicated through a pipe), giving the file-descriptor of the pipe's reading-end on the command line to the new process. Such application engineering is quite rare however — it is very application specific and relatively complex to program.

```
INLINE PROC init.mytype (RESULT MYTYPE m, VAL INT size)
  IF
    size <= 0
      m[addr], m[size] := 0, 0
    TRUE
      SEQ
        ASM
          LD size
          MALLOC         -- allocate memory
          ST m[addr]
        m[size] := size
        ...  initialise other fields
:

INLINE PROC free.mytype (MYTYPE m)
  IF
    m[size] = 0
      SKIP
    TRUE
      SEQ
        ...  perform any cleanup
        ASM
          LD m[addr]
          MRELEASE       -- release memory
:
```

3.9 Other Extensions

This section describes the remaining extensions to the occam compiler. These extensions do not make any changes to the occam language syntax directly, but new instruction support is required in the translator and run-time kernel for some of them.

3.9.1 Modified SKIP in ALT Checking

The use of a `SKIP` guard within `ALT` and `PRI ALT` explicitly requires a pre-condition, in the majority of cases this is simply '`TRUE`'. The reasoning behind this is stems from the fact that `SKIP` guards are usually a bad thing — polling in a `PRI ALT`, or very explicit non-determinism for a non-prioritised `ALT`. For example:

```
ALT
  TRUE & SKIP
    P ()
  PRI ALT
    in ? x
      Q ()
    TRUE & SKIP
      R ()
```

This is a process whose behaviour can be expressed in $\mathcal{CSPP}$ as:

$$(Skip \rightarrow P) \sqcap ((in?x \rightarrow Q) \overleftarrow{\Box} (Skip \rightarrow R))$$

When offered no events, this process may behave like either P or R, a choice that is non-determinable. However, when offered some event '$in.x$', the process may behave as either P or Q, also a non-deterministic choice. For most practical applications, such behaviour is undesirable.

The processing of such ALTing constructs has been modified slightly in the compiler, such that any explicit 'TRUE & SKIP' guards in an ALT are warned about (or an error in strict mode). Furthermore, the processing of these in PRI ALT constructs has also been modified, such that a warning is generated if a 'TRUE & SKIP' guard does not occur last in the PRI ALT. Additionally, to capture the polling idiom more accurately, the explicit 'TRUE' pre-condition can be omitted (in this case only). For example:

```
PRI ALT
  in ? x
    Q ()
  SKIP
    R ()
```

Generally speaking, SKIP guards are discouraged[11] — unless polling is absolutely required (or used in an acceptable way, for example signalling process termination [Wel89]).

3.9.2 Reversed ALT Disabling

The ALT is implemented by *enabling* and *disabling* instructions for each type of *guard*. When the ALT is entered, its guards are enabled sequentially, using the 'ENBC', 'ENBT' and 'ENBS' instructions[12] — for channel (input) guards, timeout guards and SKIP guards respectively. After enabling, if none of the guards are ready, the ALTing process is descheduled — to be rescheduled either by an outputting process on channel communication, or on a timeout. Once rescheduled, or if any ready guard was found during the enabling sequence, the disabling sequence takes place. This is done in the same order as the enabling sequence using similar instructions: 'DISC', 'DIST' and 'DISS'.

The inclusion of ALT enabling and disabling instructions for SKIP guards may at first appear a little excessive — especially for polling, where the guard is always ready (explicit or implicit TRUE pre-condition). The enabling and disabling instructions are always called for each guard however, passing the pre-condition as one of the arguments (which inside the compiler, is confusingly referred to as the 'guard' — the actual guard is called the 'input'). This is required to detect, at run-time, ALTs without any enabled guards — such a process is STOP[13].

The disabling sequence examines each guard in turn, and if *fired*, places a pointer to the *guarded-process* in the ALTing process' workspace — *but only* if no guard had previously been selected (determined by checking the guarded-process pointer in the ALTer's workspace).

The same ALTing sequences are generated for both the 'ALT' and 'PRI ALT', with no visible difference between them — 'ALT' is simply implemented as 'PRI ALT'. A deterministic implementation is a valid refinement of a non-deterministic specification [Law02].

There is a slight flaw with this approach to disabling, however. It is perfectly legal to have the same guard appear twice in a PRI ALT construct (possibly as a result of nested ALTing), and if that guard fires, the first of the guarded processes should be selected. For example:

[11] Students, in particular, tend to be under the impression that polling is required for many things — possibly from a previous training in Java, where the native language tools for managing concurrency are woefully inadequate.

[12] Strictly speaking, this is no longer the case, as an extension to the enabling sequence causes the compiler to generate alternative instructions, described in section 3.9.3

[13] In 'stop' error mode, where STOP deschedules a process rather than causing a run-time error, the need to check for enabled guards is not necessary since an empty ALT will deschedule without leaving any references to itself.

```
-- environment is: CHAN INT in?, []CHAN INT others?
INITIAL BOOL active IS FALSE:
WHILE TRUE
  INT v:
  PRI ALT
    active & in ? v
      SEQ
        P (v)
        active := FALSE
    PRI ALT i = 0 FOR SIZE others
      others[i] ? v
        O (v)
    in ? v
      SEQ
        active := TRUE
        Q (v)
```

Such a process would be expected to cycle between active and non-active states when given data on '`in`'. However, if '`in`' is not ready at the start of the disabling sequence, then ready at its second occurrence, the second guarded process will be selected. This behaviour breaks the logic of the above code. Although such cases may be potentially rare, they could still happen. The above code aggravates this — a reschedule could occur in the replicated `PRI ALT` loop, causing the process outputting to '`in`' to run and block on the communication. To provoke this bug in a uniprocessor KRoC, the option to reschedule on loop-ends must be enabled. A multiprocessor KRoC [WP97, SARW98, VW99] could easily suffer, regardless of the multi-processor synchronisation mechanisms — unless the run-time kernel was locked for the whole duration of the disabling sequence, but this requires some help from the compiler.

The remedy for this is to run the disabling sequence in reverse, such that if '`in`' becomes ready between the first and second disabling calls, the top-most guarded process is selected. Historically, the Transputer implementations '`DISC`', '`DIST`' and '`DISS`' would have prevented this — since they only select a guard if one was not previously selected. Additionally, running a loop in reverse for a replicated `PRI ALT` is somewhat tricky with a forwards-only loop-end. This particular issue has been addressed with the addition of '`STEP`' for replicators (section 3.6), where a new loop-end instruction is provided.

Three new disabling instructions have been added here, to provide implementations that do not check for a previously selected guard. For software emulations of the Transputer (such as KRoC), these new instructions incur less overhead than their checking counterparts.

The new instructions, '`NDISC`', '`NDIST`' and '`NDISS`' are detailed in section B.1.5, implemented as described. Algorithm 3.4 describes, formally, the implementation of '`NDISC`'. '`NDIST`' follows along similar lines, possibly manipulating the timer queue. '`NDISS`' is trivial — if the pre-condition is true, the guarded process will always be selected.

To add slight variation, the code generated by the compiler for *replicated* `ALT`s and `PRI ALT`s now differs. A plain replicated `ALT` disables the guards in the program-specified order, using the new instructions, whereas a replicated `PRI ALT` disables the guards in reverse (also using the new instructions). This difference in the behaviour of replicated `ALT`s and `PRI ALT`s will hopefully help programmers to locate errors caused by these. In particular, where a fair-`ALT` is mis-programmed using a replicated `ALT` instead of a replicated `PRI ALT`.

Algorithm 3.4: New channel `ALT` disabling algorithm

```
//Wptr: invoking workspace pointer (of ALTer)
//chan.addr: channel address
//guard: evaluated pre-condition
//proc.addr: address of guarded process
ndisc (chan.addr, guard, proc.addr):
if guard = false then
   return false
else
   if chan.addr[+0] = Wptr then
      // ALTer is still in the channel, clear it
      chan.addr[+0] ⇐ NotProcess.p
      return false
   else if chan.addr[+0] = NotProcess.p then
      return false
   else
      Wptr[Temp] ⇐ proc.addr
      return true
   end if
end if
```

3.9.3 Enhanced `ALT` Enabling

The `ALT` construct in occam has always been a fairly expensive operation, incurring an $O(n)$ cost, where n is the number of guards. In cases where a ready guard is found during the `ALT` enabling sequence, this cost is excessive — remaining guards will also be enabled, followed by the whole disabling sequence. Although the existing enabling instructions, '`ENBC`', '`ENBT`' and '`ENBS`', *do not* provide a direct indication of whether the guard is ready, the `ALT`ing process' *State* (also pointer) field is set to 'Ready.p' when a guard is found to be ready during the enabling sequence. This could be checked after enabling with a conditional jump to the disabling sequence. The compiler does not do this however. When targeting the Transputer for embedded systems, such checking would have probably been considered excessive, given the hardware implementation of the `ALT`.

The primary reason why the compiler cannot check, however, is because it does not know the layout of the process' workspace at run-time — all the 'magic' locations (which lie below the workspace pointer and include the *State* field) are reserved in fixed amounts. This provides a certain degree of flexibility (abstraction) between the compiler output and a run-time system that provides real implementations for workspace-manipulating instructions.

A new set of enabling instructions have been added to the compiler, that take as a parameter an address to jump to if the guard is ready, named '`ENBC3`', '`ENBT3`' and '`ENBS3`' (they take three parameters instead of the two used for the original enabling instructions — hence the '3'). These three new instructions are described fully in section B.1.5 in the appendices. The implementation for the '`ENBC3`' instruction is shown in algorithm 3.5; the others follow along similar lines.

The basic use for these new instructions in the compiler is to jump to the start of the disabling sequence — skipping the other enables (if any), and the '`ALTWT`' or '`TALTWT`' ('alternative wait') call. Code such as this is generated for a standard `ALT`. The `PRI ALT` can potentially be more efficient, however, given that the disabling sequence is generated in reverse (discussed in the previous section 3.9.2). Rather than jumping to the start of the disabling sequence, an enabling instruction can

Algorithm 3.5: Enhanced channel `ALT` enabling algorithm

```
//Wptr: invoking workspace pointer (of ALTer)
//Iptr: return address in occam process (of ALTer)
//chan.addr: channel address
//guard: evaluated pre-condition
//jump.addr: target address for ready guard
enbc3 (chan.addr, guard, jump.addr):
if guard = false then
  return false
else
  if chan.addr[+0] = NotProcess.p then
    chan.addr[+0] ⇐ Wptr
  else if chan.addr[+0] ≠ Wptr then
    Wptr[State] ⇐ Ready.p
    Iptr⇐ jump.addr
    return
    // nothing is returned when the jump is taken
  end if
  return true
end if
```

jump directly[14] to its corresponding disabling instruction, thereby only disabling this and previously enabled guards.

In cases where no guards are ready, the modified enabling sequence does not cause any additional overhead (for either `ALT` or `PRI ALT`) — all of the guards will be enabled before the alternative wait, followed by a complete disabling sequence afterwards — an $O(n)$ cost, where n is the number of guards. In cases where a ready guard is found during the enhanced enabling sequence, the cost for an `ALT` is *reduced* to $O(\frac{n+m}{2})$, where m is the index of the first ready guard, and $m \leqslant n$. For the `PRI ALT`, the overheads are reduced further to $O(m)$.

To maintain a difference in implementation for `ALT` and `PRI ALT`, the compiler generates the `ALT` enabling sequence backwards and the disabling sequence forwards (program order). This causes a replicated `ALT` to behave like '`ALT PRI`' — priority given to the last guard. The `PRI ALT` is coded the other way around — enabling sequences are generated in the forward (program-specified) order, whilst disabling sequences are generated backwards.

`ALT` Pre-Enabling

A further enhancement to the `ALT` enabling, suggested by Welch [WB02] is to have an ultra-fast pre-enabling sequence, that just tests the guards without actually enabling them. If a ready guard is found, the corresponding guarded process is jumped to immediately — without the need to disable any guards. If the pre-enabling sequence completes, then the enabling sequence proper takes place (no channels were ready during the pre-enabling). Such a modification will add some overhead in the case where no guards are ready (and if and only the `ALT` would have blocked — i.e. no available `SKIP` guards). For standard polling code, this enhancement represents a significant increase in performance. For example:

[14] When a replicated `ALT` or `PRI ALT` is involved, the jump out from the enabling sequence goes via some special code inserted to *reverse* the replicator values — such that the disabling loop happens the opposite way around.

```
PRI ALT
  c ? x
    ...  process "x"
  SKIP
    ...  no data yet, do something else
```

If the plain enhanced `ALT` enabling sequence is generated, this code will always enable then disable '`c`' — if that channel ('`c`') is not ready. The further enhancement removes this, testing the channel '`c`' only once — no enabling or disabling code is ever generated[15].

To be efficient, the pre-enabling of timer-guards is done by storing the current time in the '*Time*' workspace slot (typically at offset -5). The pre-enable timeout instruction then just compares this stored time with the timeout specified in the enabler.

In effect, this pre-enabling sequence behaves much like an `IF`, with the original `ALT` or `PRI ALT` as the '`ELSE`' case. The above `occam` fragment, for example, could be represented as (unoptimised):

```
IF
  CHANNEL.READY (c)
    SEQ
      c ? x
      ...  process "x"
  TRUE
    ...  no data yet, do something else
  TRUE
    PRI ALT
      c ? x
        ...  process "x"
      SKIP
        ...  no data yet, do something else
```

For this code, with a '`SKIP`' guard, it is easy to see where the optimisation happens — the second '`TRUE`' guard containing the `PRI ALT` will never be executed. A more complex `occamALT` construct, including timeouts, might be:

```
ALT
  have.0 & in[0] ? x
    ...  process "x"
  have.0 & tim ? AFTER timeout[0]
    ...  first timeout process
  have.1 & in[1] ? y
    ...  process "y"
  have.1 & tim ? AFTER timeout[1]
    ...  second timeout process
```

To add some more significant variation between the implementations of '`ALT`' and '`PRI ALT`', the pre-enabling sequence processes '`ALT`' guards in reverse order, for straight `ALT`s only — replicators are processed in forward order. This causes plain `ALT`s (un-replicated) to behave like a reversed `PRI ALT` (often referred to as '`ALT PRI`'). The run-time behaviour for this code, with the pre-enabling sequence, would be:

[15] This is somewhat of a special case, and only applies to cases where `SKIP` guards have no, or an explicitly `TRUE`, pre-condition.

```
INT $now:
SEQ
  tim ? $now
  IF
    have.1 AND ($now AFTER timeout[1])
      ...  second timeout process
    have.1 AND CHANNEL.READY (in[1])
      SEQ
        in[1] ? y
        ...  process "y"
    have.0 AND ($now AFTER timeout[0])
      ...  first timeout process
    have.0 AND CHANNEL.READY (in[0])
      SEQ
        in[0] ? x
        ...  process "x"
    TRUE
      ALT
        ...  original ALT construct
```

Implementation of Pre-Enabling

The pre-enabling sequence is implemented using the three existing enhanced `ALT` enabling instructions, prefixed with a special compiler comment in the output code ("`.MAGIC PREENABLE`", covered in detail in section B.2.3). The translator intercepts these and generates the checking code in-line. The label given as an operand to the enabling instruction is that of the guarded process, rather than the corresponding part of the disabling sequence.

As shown above, the code generated for pre-enabling takes place completely outside the `ALT`ing process. However, this causes some additional headaches of its own, particularly where replicated `ALT`s are involved. For replicated `ALT`s, the replicator name is allocated in the process' workspace, kept separate from workspace used to handle the replicator (value, count, and possibly step fields). Ordinarily, replicator values are copied from the replicator value-fields to the `ALT`er's workspace when a ready guard is found during disabling (the disabling instructions return a result indicating whether the guard fired). With the enhanced `ALT` enabling sequence, this is taken care of automatically, since the enabler jumps directly to its corresponding disabler. For the *pre-enabling* sequence, special code is inserted into the program to recover replicator values if needed — i.e. the jump from the pre-enable to the guarded process goes via code that loads any necessary replicator values.

On the other hand, where replicated `ALT`s are present, the enhanced `ALT` enabling sequence will insert code to reverse the loop mid-cycle. It is not hard to imagine, then, trivial occam programs that generate a large amount of code, to handle incrementally: pre-enabling, enhanced enabling, then reversed disabling. For example:

```
PRI ALT
  c ? x
    P (x)
  ALT i = 0 FOR SIZE in
    in[i] ? x
      Q (i, x)
```

generates a sizeable amount of code, most of which is to handle the nested replicated `ALT`. However, the potential performance gains, as discussed in the following section, make it a worthwhile optimisation. As an example, the (annotated) code generated for the pre-enabling of the above is:

```
  LD ADDRESSOF c
  LDC 1                  -- pre-condition
  LD ADDRESSOF L5        -- address of guarded process
  .MAGIC PREENABLE
  ENBC3                  -- pre-enable channel
  .TSDEPTH 0             -- ignore (non-generated) result

  ...  initialise loop out at L7: $Rbase, $Rcount := 0, SIZE in
L6:
  LD $Rbase
  LD ADDRESSOF in
  WSUB                   -- get address of channel pointer
  LDNL 0                 -- get channel pointer
  LDC 1                  -- pre-condition
  LD ADDRESSOF L8        -- address of value selector process
  .MAGIC PREENABLE
  ENBC3                  -- pre-enable channel

  .LEND $Rbase, L6       -- loop end
L7:
```

The label 'L5' is the start of a slightly modified guarded process — the call to 'P':

```
SEQ
  c ? x
  RESCHEDULE ()
  P (x)
```

The 'RESCHEDULE' is necessary to ensure that the ALTing process reschedules at least once, and is inserted for the enhanced-enabling sequence, as well as for pre-enabling. This protects against an unfortunate effect that occurs when an outputting process communicates continuously into an ALTing input process, that is always ready to accept that communication. In such a case, with pre-enabling and/or enhanced-enabling, the ALTing and outputting processes could end up rescheduling each other directly indefinitely[16] — partly a consequence of performance enhancements in the run-time kernel's implementation of channel input and output.

The other label ('L8') referred to in the above compiler output holds the code that selects the current value of 'i' (the replicator name), before jumping to the guarded process proper.

3.9.4 Benchmarking the ALT Enhancements

In order to gauge the performance of the various ALT enhancements (reversed disabling, enhanced enabling, and pre-enabling), a simple benchmark program is used that measures the overhead for a replicated ALT or PRI ALT. The benchmark program consists of two processes, connected by an array of channels. The consumer process ALTs over the channel array, while the producer process communicates values down those channels.
Three forms of ALT are investigated; a plain replicated ALT:

[16] Until an event occurs that *interrupts* the run-time kernel.

```
ALT i = 0 FOR SIZE chans
  INT any:
  chans[i] ? any
    SKIP
```

A prioritised replicated `ALT`:

```
PRI ALT i = 0 FOR SIZE chans
  INT any:
  chans[i] ? any
    SKIP
```

And a fair `ALT`:

```
-- before benchmark: INITIAL INT f IS 0:
PRI ALT i = f FOR SIZE chans
  VAL INT i IS i \ (SIZE chans):
  INT any:
  chans[i] ? any
    SEQ
      SKIP
      f := i + 1
```

For each type of `ALT`, three tests are run. One where communication occurs on channel 0 (first), one where communication occurs on channel $n - 1$ (last), and one where each communication occurs on a random channel between 0 and $n - 1$ inclusive (the '`course.lib`' pseudo-random-number generator is used, implementing the "minimal standard" described in [PM88], and seeded from the microsecond clock).

Figures 3.7, 3.8 and 3.9 shows the results for these benchmarks. The number of channels is shown on the X-axis, and ranges from 1 to 99 for each test, incremented by 2 each time. The Y-axis shows the time per replicated `ALT` — for the whole replicated `ALT`. Each test is run 10 000 times and the results minimised over 3 complete runs in an attempt to reduce the occurrence of any spurious results (where other activity occurred on the benchmark machine, for example).

Note that these graphs only show results for *replicated* `ALT`s, that incur an additional overhead when compared with non-replicated `ALT`s. That extra overhead represents the cost of the replicator itself, and the code required to reverse the replicator (when transferring from the enhanced enabling sequence to the reversed disabling sequence). The pre-enabling sequence also incurs a similar overhead, but typically half, or less — since (in cases where a guard is ready) the disabling sequence is never required, although code for it may be generated.

Behaviour of `ALT`s

Figure 3.7 shows the total cost for a whole replicated `ALT`, as the number of channels is varied. As the results show, the cost of an `ALT`, without any enhancements, is $O(n)$, regardless of which channel within the `ALT` is ready. The "enhanced" results (reversed disabling, and enhanced enabling), show that the resulting cost of the `ALT` depends on where the first ready channel appears within the `ALT` — as is to be expected. When the first channel (in the code) is ready, the `ALT` still examines the other guards, examining the ready guard last. Correspondingly, when the *last* channel in a replicated `ALT` is ready, it will used immediately, since it is the first channel examined; resulting in the constant-code for the `ALT`, regardless of the number of channels.

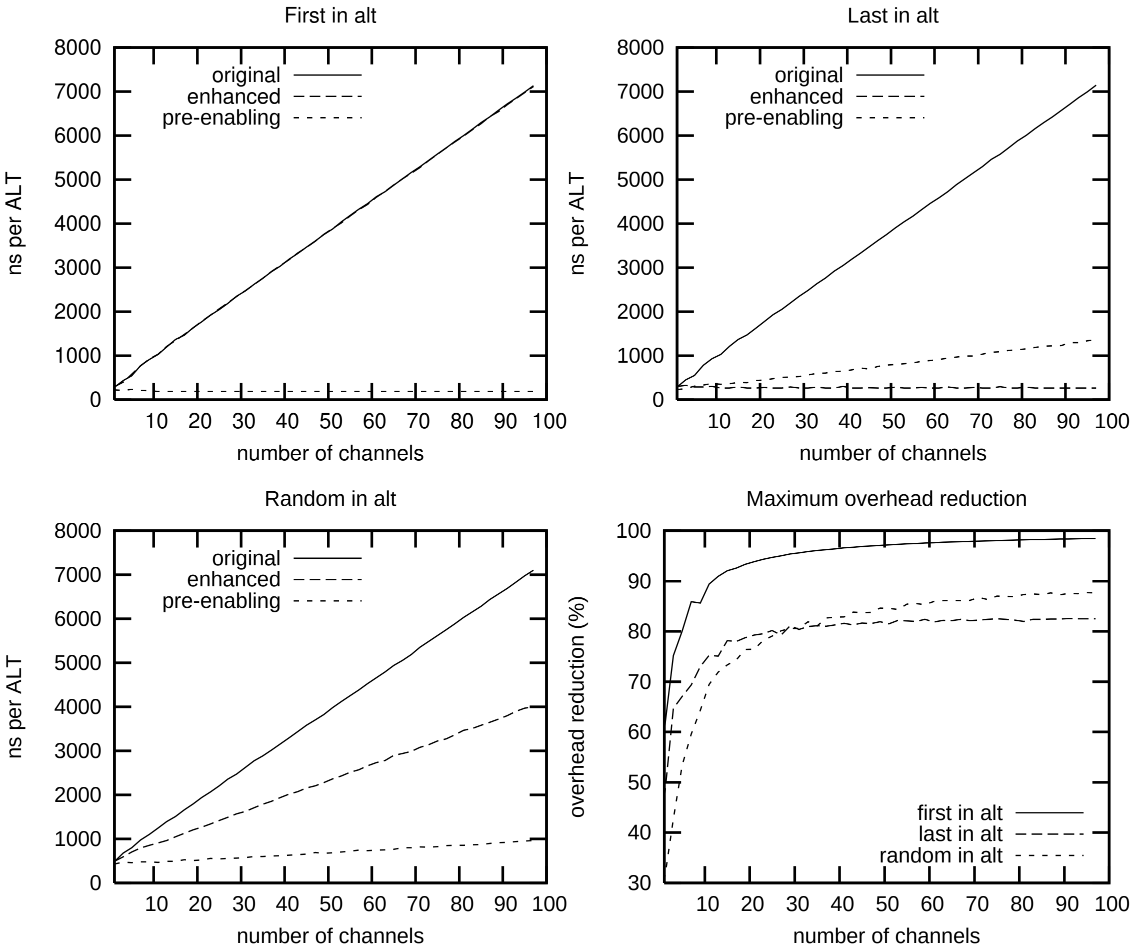


Figure 3.7: Benchmark results for a replicated `ALT`. Results are shown for first-in-alt, last-in-alt and random-in-alt communication, as well as the best-case reduction in overheads

For `ALT`s where an arbitrary (random) channel is ready, enhanced enabling represents a reasonable saving, typically a little over half the cost of a non-enhanced `ALT`.

The addition of pre-enabling has the most pronounced effect — a significant reduction in the cost of an `ALT`. The pre-enabling sequence for replicated `ALT`s is generated in program-order (i.e. forwards). The 'first' and 'last' graphs here represent the best and worst cases for pre-enabling respectively, with 'random' somewhere between.

The "overhead-reduction" graph shows the maximum possible performance increase (overhead reduction), from the original `ALT` to optimised code with pre-enabling. For small numbers of channels, these enhancements represent a 30–60% increase in performance. For 10 or more channels, these enhancements produce a 65–85% reduction in overheads, a potentially significant saving. At around 30 channels and above, this saving is always 80% or above.

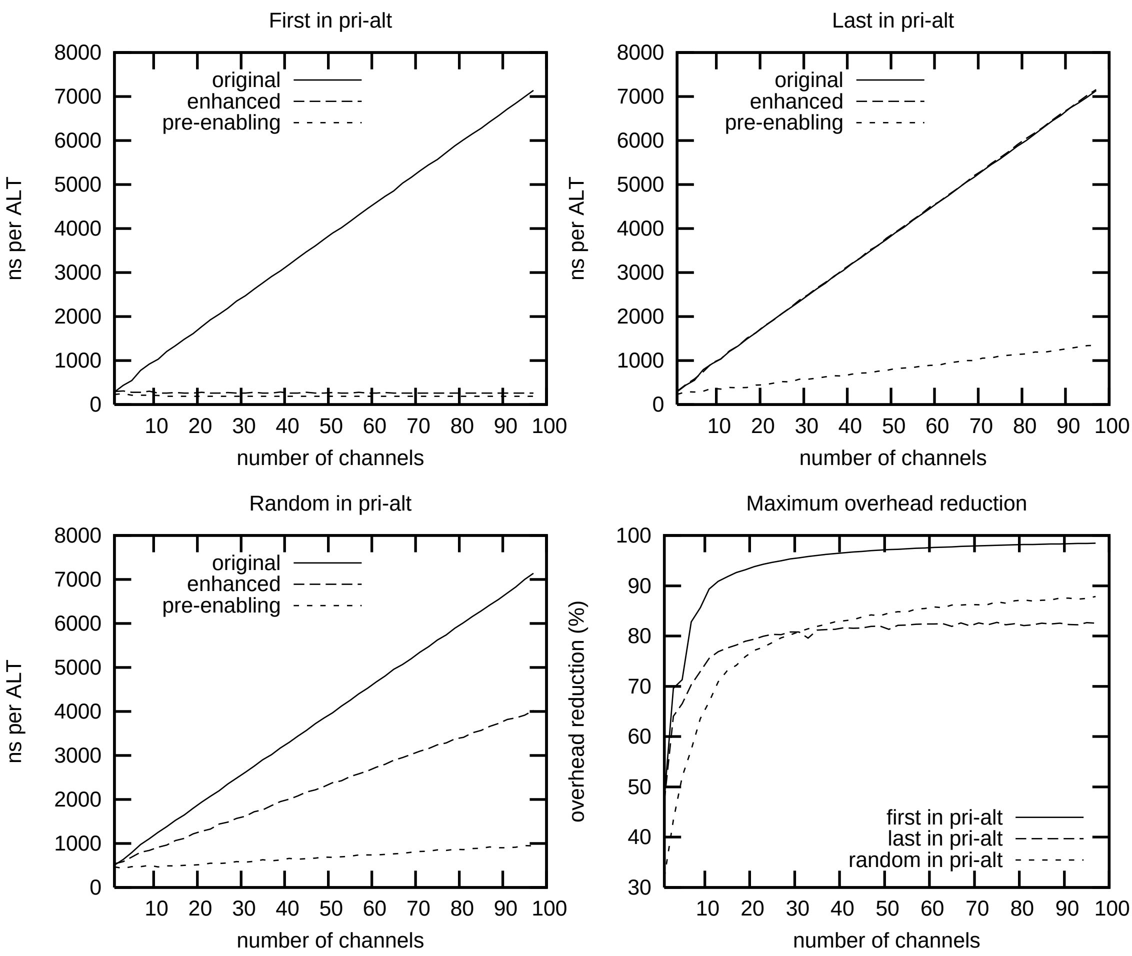


Figure 3.8: Benchmark results for a replicated `PRI ALT`. Results are shown for first-in-alt, last-in-alt and random-in-alt communication, as well as the best-case reduction in overheads

Behaviour of `PRI ALT`s

The `PRI ALT` incurs similar (or exactly) the same overheads as for `ALT`s, as shown in figure 3.8. The primary difference is a change in the processing order for guards in the enhanced-enabling and reversed-disabling sequences.

The run-time behaviour of a `PRI ALT` for random ready guards is unchanged from the `ALT`, given that each channel has the same probability of being ready (and therefore selected). Similar behaviour would be expected if the benchmark were re-written to include n parallel output processes, that output a fixed number of times, one for each of the n channels. However, the run-time memory footprint would change significantly, and cache-effects along with it.

The best-case overhead reduction is slightly different than that of the `ALT`. Most notably there is a 'step' in both graphs, where performance levels slightly. This is undoubtedly an effect of the cache behaviour between the 'original' and 'pre-enabling' (with inlining). However, the size of this effect is small enough not to be of significant concern.

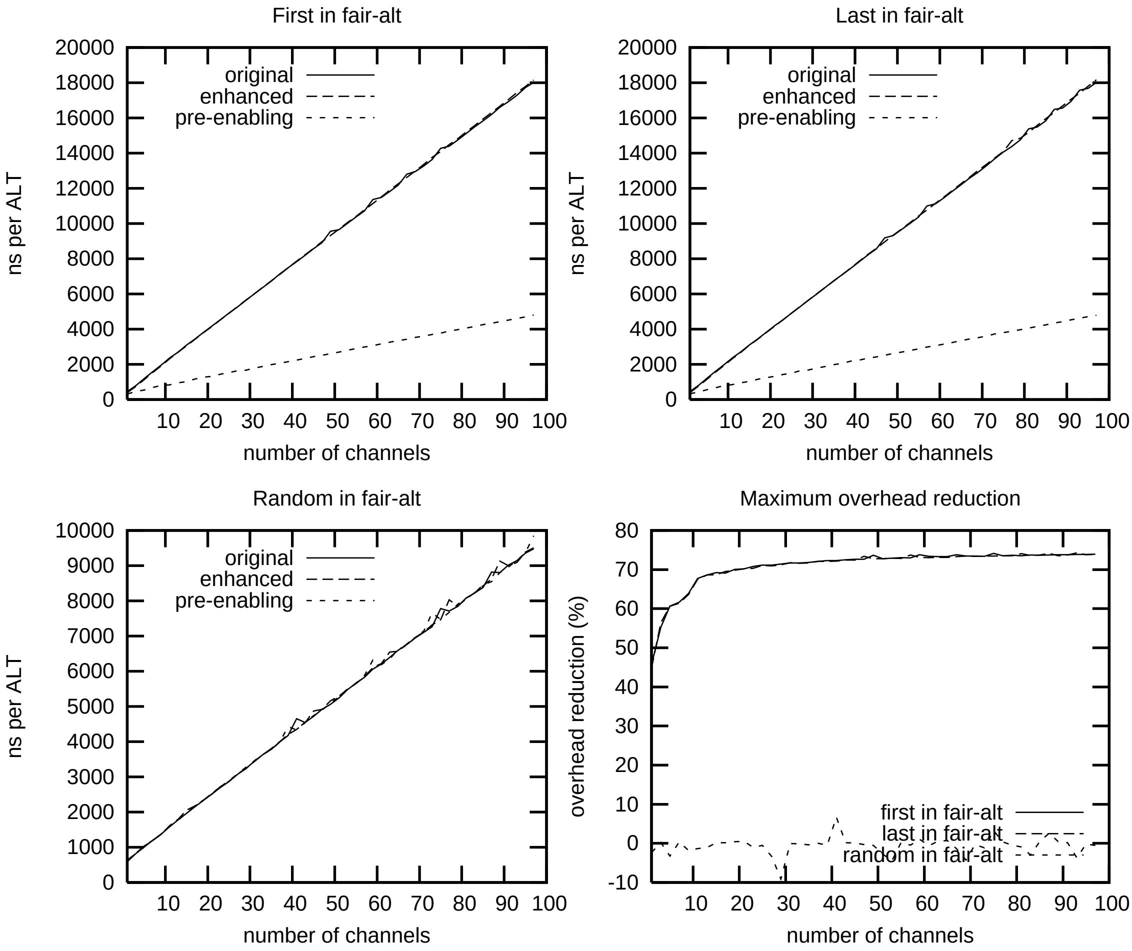


Figure 3.9: Benchmark results for a replicated *fair* `ALT`. Results are shown for first-in-alt, last-in-alt and random-in-alt communication, as well as the best-case reduction in overheads

Behaviour of Fair-`ALT`s

Fair-`ALT`s are typically used when input channels must be serviced *fairly.* When a random channel is ready each time, the optimised `ALT` sequences have little overall effect, as expected. Where either the first or last channel is always ready, pre-enabling gives a moderate saving — of about 70% when more than 30 channels are involved (although this is slightly artificial, as all but one of those channels are inactive).

3.9.5 Auto-Extended `CASE` Input

As mentioned previously, in section 3.7.2, an outputting process *will* be incorrectly rescheduled if the corresponding inputting process does not handle the tag sent — and where that tag has no following sequential protocol. Note that this only applies in *stop* error-mode — where processes are simply removed from the run-queue when they generate a run-time error. The default error-mode is *halt* error-mode, where the scheduling of all processes is stopped when a run-time error is generated.

'Stop' error-mode was initially not fully supported by KRoC/Linux. This has now been remedied, following from the support for run-time debugging (described in section 7.3).

As an example, consider the following (fairly arbitrary) occam protocol declaration and process inputting from a channel of that type:

```
PROTOCOL CONTROL
  CASE
    reset                   -- reset device
    write.byte; BYTE        -- write a byte to the device
:

PROC hw.device (CHAN CONTROL in?, ...)
  WHILE TRUE
    in ? CASE
      BYTE ch:
      write.byte; ch
        ...  write 'ch' to device
:
```

Clearly, if this process is sent a '`reset`' message, it will `STOP`. In the default halt error-mode, what happens to the outputting process is largely irrelevant — it will be terminated along with the rest of the system. However, this only applies in cases where both processes are running within the same system — which might not be the case in reality[17].

For those cases where the inputting process does not handle an empty tag, and the corresponding outputting process does not terminate as a result, that outputting process *must not* be allowed to continue — semantically, the communication never succeeded (the inputting process could not handle it). The solution, to this particular problem, is to have the inputting process perform an extended input on the tag. If the tag is not handled, the inputting process `STOP`s before completing the extended input, so the outputting process is never resumed.

This functionality is enabled by specifying the '`-xtag`' option to the compiler, or the '`-X6`' (experimental) option to the '`kroc`' wrapper script. If it is not already, the tag input is extended only when there are unhandled empty tags — e.g. '`reset`' in the above '`CONTROL`' protocol. Internally, the above '`hw.control`' process is transformed to:

```
PROC hw.device (CHAN CONTROL in?, ...)
  WHILE TRUE
    in ?? CASE
      BYTE ch:
      write.byte; ch
        SKIP                -- extended process
        ...  write "ch" to device
      ELSE
        STOP                -- without completing extended input
:
```

Thus, if an outputting process communicates an unhandled tag, such as '`reset`', it will never successfully complete that input.

[17]Various extensions presented within this thesis enable communication between separate occam systems. In particular, user-defined channels (section 6.4), the CSP-driver (section 6.7), and the higher-level interaction support (section 7.4).

3.9.6 Strict Checking

One small addition to the occam compiler is that of 'strict' checking. Enabled with the '`-strict`' compiler flag, this effectively turns a large proportion of compiler warnings into errors. Most of the warnings affected are those relating to the new extensions, throughout the whole of this thesis. For example, in strict mode, any undefined or unknown variable use is an error, rather than a warning (within the undefinedness checker — section 4.6).

Strict mode additionally makes the use of channel direction specifiers compulsory, for parameters and abbreviations. Given that channel direction specifiers significantly enhance the comprehensibility of code, forcing their use in strict mode appears to be quite reasonable — largely in the interests of promoting good programming practices.

3.10 A C-Like Syntax for occam

In a fairly random poll of second-year occam (principally computer-science) students, given a choice between C, Java and occam for arbitrary (non-IO intensive) program implementation, the majority of students choose Java. occam is generally rejected solely on the basis of its syntax — primarily the indentation. This view is not just held by students. Generally, it seems the programming community prefer languages that have a C-style syntax, possibly because of their obvious familiarity. As an indirect result of this, many editors lack support for efficient editing of occam programs. Editors available under UNIX (specifically '`origami`' and '`vim`') do significantly better in this department, currently where most KRoC ports are targeted[18].

To remedy this rather unfortunate situation, a C front-end for the occam compiler is currently under construction. Previous discussions on this topic have often envisaged pre-processors that turn a C-style syntax into occam. This has its own problems, however, such as reporting errors in the original sources. As such, this support is being implemented directly within the occam compiler itself — the semantics of the language being compiled is still that of occam (and CSP), only the syntax has changed.

Although not directly intentional, the resulting syntax has many similarities with the Handel-C language [APR$^+$96]. This is to be expected in some ways, since Handel-C was originally derived from occam, but specifically for *hardware* compilation. Handel-C supports many hardware-specific features that this C-style occam syntax does not. The most obvious missing feature is support for arbitrary-sized integer types[19].

The following sections present the various syntactical structures that the C-style syntax uses. One key point to remember is that the language is still occam— not C. As such, the various rules that apply to occam expressions (no operator precedence), also apply to the C-style version of the syntax. The most substantial change is the replacement of indentation with curly braces, '{' and '}'. To avoid any ambiguity, braces are necessary wherever matching indentation occurs in the occam version. As far as names are concerned, either the traditional occam dot '`.`' or an underscore '`_`' may be used as a separator in names (but not as the first character). Both are equivalent as far as the compiler is concerned (internally dot, externally underscore). Thus, new C-style code can refer to either '`out_string`' or '`out.string`' — the same function (from '`course.lib`') is used.

[18]Currently, no KRoC port for Microsoft Windows tm exists, which has a vast user-base. Attempts to create a Windows port have been partially successful, but several critical issues remain. Usefully, the central components of KRoC (compiler, translator and run-time system) ported with very little effort; the remaining problems are largely tool-chain related — including a failure of the linker to produce executables from occam sources.

[19]Such types could be provided for in software emulations, perhaps (speculatively) with less than one order of magnitude overhead — some operations however, particularly divide and modulo, will be complicated.

3.10.1 Definitions and Declarations

The keywords of the C-style syntax are no different from that of the existing occam syntax, except they are presented in lowercase. Furthermore, the '`in`', '`of`' and '`at`' keywords have been removed completely. Since keywords cannot be used as names, this would prevent the use of '`in`' as a formal-parameter name.

Declarations (of variables and constants) and definitions (of procedures and functions), follow the existing occam style and use of keywords. There is one slight issue with the placement of these, however. In languages such as C and Java, declarations are made *inside* the scope that defines their lifetime. For occam, variables are always declared *before* they enter scope, such that they only exist in the immediately following process (that may be a '`SEQ`' or other process constructor). This implementation retains the occam way of handling declarations, and is unlikely to be a burden for the programmer (any compiler errors as a result of mis-placed declarations would only require 'hoisting' of those declarations).

As an initial example, consider the following occam process (technically, procedure) definition (that lacks any substantial body):

```
PROC my.nos (CHAN INT out!)
  INT x:
  ...  body of "my.nos"
:
```

The C-style equivalent of this code is simply:

```
proc my.nos (chan int out!)
{
    int x;

    ...  body of "my.nos"
}
```

Abbreviations, both value and reference (including results), are formed in the occam style, using the '`is`' operator. Initial declarations may use either '`is`' or '`=`'. For example:

```
int y is x;                // abbreviate 'x' to 'y'
val byte ch is str[4];     // value abbreviation
initial int z = (y * 2);   // declare and initialise 'z'
```

Functions continue to be defined in the occam style, with a slight modification to the layout of 'value' processes. Short function definitions may either use '`is`' or '`=`' as a separator. For example:

```
int function f (val int v) = (v + 2);

int, real64 function g (val real32 rv)
{
    int res_i;
    real64 res_r;

    valof {
        ...  body of value process
    }
    result res_i, res_r;
}
```

A feature that is supported, but perhaps whose use is not entirely recommended, is that of replacing '`proc`' with '`void`', and removing '`function`' entirely. This makes definitions syntactically more like C and Java. For example:

```
void some_proc (val int i, val byte[] in, byte[] out)
{
    ...  body of "some_proc"
}

int other_func (val int x, val byte[] data)
{
    ...  (valof/result) body of "other_func"
}
```

Array Specifications

As shown in the last example, array specifications are not necessarily formed in the traditional occam way. The vast body of Java programmers, to whom this is partially intended to appeal, are used to the array-style shown. To allow some flexibility, both the occam-style and new-style versions of array specifications are supported. Mixing them in general programming is not recommended, but for example:

```
void array_test (val []int in, []int out)
{
    int[5] temporary;
    initial mobile byte[] message = "hello world!\n";
    ...  body of process
}
```

3.10.2 Sequential and Parallel Composition

One feature which is *not* supported is that of "automatic sequential code" — as is the norm in languages like C and Java. Code in Handel-C is also automatically sequential, unless '`par`' is explicitly specified. Sequential and parallel blocks are formed in the expected way, using semicolons to separate items. For example:

```
int x, y;

seq {
    par {
        x = 42;
        y = 99;
    }
    out.int (x, 0, screen!);
    screen ! '\n';
    out.int (y, 0, screen!);
    screen ! '\n';
}
```

```
INT x, y:
SEQ
  PAR
    x := 42
    y := 99
  out.int (x, 0, screen!)
  screen ! '*n'
  out.int (y, 0, screen!)
  screen ! '*n'
```

The C-style version of the code clearly follows the occam, and is *exactly* equivalent. However, the sensible use of whitespace in the C-like syntax is programmer-dependent. The above (C-like) code could equally be written as:

```
int x, y; seq { par { x = 42; y = 99; };
out.int (x, 0, screen!); screen ! '\n';
out.int (y, 0, screen!); screen ! '\n'; }
```

This version is actually very slightly different — it includes a semi-colon after the 'par' closing-brace. In general, semi-colons are optional after any closing brace — as is generally the norm in C, C++ and Java (with the exception of C-style structured definitions).

Replicators, as they exist in occam, are not entirely well represented in C. Rather than attempting to provide syntax support for C-style "replicators", the occam style is retained. For example, with both the C-style and occam syntax:

```
seq (i = 0 for 10) {
    seq {
        out.int (i, 0, screen!);
        screen ! '\n';
    }
}
```

```
SEQ i = 0 FOR 10
  SEQ
    out.int (i, 0, screen!)
    screen ! '*n'
```

'par' replicators follow suite: i.e. "`par (i = 0 for n) {...}`". Replicator strides are specified in the same (occam) way, using the '`step`' keyword.

3.10.3 Conditionals

The occam '`IF`' behaves in a similar way to C-type '`if`'/'`else if`' constructs. It makes good sense to keep the C-style equivalent for `IF`, thus:

```
if (x == 42) {
    v = 2;
} else if (x < 20) {
    seq {
        v = 19;
        out ! x;
    }
} else {
    par {
        in ? v;
        out ! x;
    }
}
```

```
IF
  x = 42
    v := 2
  x < 20
    SEQ
      v := 19
      out ! x
  TRUE
    PAR
      in ? v
      out ! x
```

In an occam `IF`, the lack of any `TRUE` guard (after evaluation) results in a `STOP`. This behaviour is unchanged for the C-style syntax — any '`skip`' must still be explicit:

```
if (x < 0) {
    x = 0;
} else {
    skip;
}
```

Omitting the '`skip`' and attempting to specify a pair of empty braces will result in a parse error.

Replicated 'IF's in occam must either contain a condition (and intended process), or another 'IF' structure. Specifying a replicator together with a condition in a C-style syntax is messy, so a nested 'if' must always be used. For example:

```
if (i = 0 for (size X)) {
    if (X[i] == v) {
        x = i;
    }
}                 // else stop.
```

The occam 'WHILE' is easily expressed in the C-style syntax, for example:

```
while (!found) {                              WHILE NOT found
    ...  body of loop                           ...  body of loop
}
```

The only other conditional statement supported in occam is 'CASE'. This is also easily expressed in the C-style syntax, but using the 'switch' keyword instead. For example:

```
switch (str[i]) {                             CASE str[i]
case 'A':                                       'A','B'
case 'B':                                         a, b := i, i
    a, b = i, i;                                'C'
case 'C':                                         SEQ
    seq {                                           a := 0
        a = 0;                                      b := i
        b = i;                                  ELSE
    }                                             SKIP
default:
    skip;
}
```

Note that the C-style version of the syntax does not use the 'break' keyword. Like plain occam, the default behaviour is to always "break", and never to "fall through".

3.10.4 Alternatives

The 'ALT' and 'PRI ALT' in occam have no real equivalent in C. Although the ALT is similar to IF in some ways, an IF-style syntax for C-like ALTs does not work particularly well. Handle-C uses a syntax similar to C's 'switch' construct, but where there is no condition and the 'case's are guards (both input and output are supported in Handel-C).

A syntax similar to that of Handel-C has been implemented for the 'ALT' and 'PRI ALT' constructs. For example, showing both the new C-style and original occam syntax:

```
alt {
    pri alt {
    c[0] ? x:
        P (x);
    tim ? after t:
        T (t);
    }
    alt (i = 1 for ((size c) - 1)) {
    c[i] ? x:
        Q (i, x);
    }
}
```

```
ALT
  PRI ALT
    c[0] ? x
      P (x)
    tim ? AFTER t
      T (t)
  ALT i = 1 FOR ((SIZE c) - 1)
    c[i] ? x
      Q (i, x)
```

Note, again, the lack of any '`break`' statements. This behaviour is always implicit, and any attempts to use '`break`' will result in an error (it is not a keyword in the C-style syntax).

Pre-conditions may be used, separated from the guard using the '`&&`' operator. Skip guards are specified using the '`skip`' keyword. In `occam`, specifications (typically declarations) may be placed immediately before an `ALT` guard. This is still permitted in the C-style syntax, although it does look a little strange. For example:

```
pri alt {
    alt (i = 0 for 10) {
    int x;
    (i < (size c)) && c[i] ? x:
        ...  process data in 'x'
    }
skip:
    ...  no data ready
}
```

The lack of an explicit "`true &&`" pre-condition on the above '`skip`' guard is a result of its optional use within a '`pri alt`' — as covered in section 3.9.1.

3.10.5 Communication, Assignment and Expressions

Communication is specified in almost exactly the same way as standard `occam`, with the slight exception that '`;`' cannot be used to separate items in sequential protocols. Instead a comma is used. Simple protocol definitions have the expected form. The syntax of variant protocol definitions is slightly peculiar, partly to avoid complication in the parser, and resembles a '`switch()`' style syntax more than anything else. For example:

```
protocol PWindow is int;

protocol PDrawRequest {
    case {
        show:                            // empty tags
        hide:
        draw.line, int, int, int, int:  // x1, y1, x2, y2
        ...   other cases
    }
}
```

Variant input and output follow a similar '`switch()`' style syntax:

```
chan PDrawRequest c;
par {
    while (true) {
        c ? case {
            show:
                ...  make widget visible
            hide:
                ...  make widget invisible

            int x1, y1, x2, y2;
            draw.line, x1, y1, x2, y2:
                ...  draw a line between x1,y1 and x2,y2
        }
    }

    seq {
        c ! show;
        c ! draw.line, 0, 0, 42, 128;
    }
}
```

Extended inputs are done following the occam style — simply one or two processes "indented" under the communication. Usually, two processes following each other, not bound by a '`seq`' or '`par`', is an error — the all too familiar 'incorrect indentation' error in occam. Braces must still be used to represent the indentation for extended-inputs, for example:

```
in ?? x {
    out ! x;                // extended process
    x = 42;                 // following process
}
```

As shown above, assignment uses the C-style operator, but is occam-like in all other aspects. That is, the left-hand items must be assignable and the right-hand items must be expressions. The slight exception being for mobile assignments, where source mobiles are left undefined.

Expressions use C-style operators, but must still be bracketed according to occam rules — i.e. the expression '$a + b + c$' must be expressed as either '$(a + b) + c$' or as '$a + (b + c)$'. Type-casts in occam are explicit, formed by using the desired type as a monadic operator — with either '`TRUNC`' or '`ROUND`' required for some conversions. For the C-style syntax, casts must still be explicit, but they are formed in the C way, e.g. "`(byte)x`" or "`(real64 trunc)y`", for "`BYTE x`" and "`REAL64 TRUNC y`" respectively.

The following shows arbitrary occam fragments that make extensive use of expressions, assignment and communication:

```
chan int c;
int x;
par {
    c ! (int round)(100.0 * SIN ((2.0 * PI) / 8.0))
    c ? x
}
```

```
int seed, v;
seq {
    timer tim;
    tim ? seed;
    seed = (seed >> 2) + 1;

    v, seed = random (512, seed);
}

mobile byte[] str;
seq {
    str = "hello, mobile C-like world!\n";
    out.string (str, 0, scr!);
}
```

3.10.6 Inclusion and Separate Compilation

The implementation of the C-style syntax is restricted to the front-end of the compiler, largely in the lexer — with some support in the parser. The choice of 'language' is based on the source file-name alone. If a file has the extension '`.occ`' or '`.inc`', it is parsed as occam. On the other hand, files with extensions of either '`.co`' or '`.ci`' are treated as C-style input files.

The implication of this is that occam files may include C-style sources and visa-versa. This is a feature particularly unique to the occam compiler, since the usual route for supporting multiple language front-ends is usually to provide an entirely new front-end — e.g. the '`gcc`' (for C) variants '`g++`' (for C++), '`gcj`' for Java, and '`g77`' for Fortran-77, as well as others. For the occam compiler, both supported syntaxes are essentially occam— the changes are mainly in keyword cases and the handling of indentation. For example:

```
#use "course.lib"
#include "semaphore.inc"    // occam user-defined semaphore
#include "semtest.occ"      // for PROC sem.test (VAL INT i,SEMAPHORE sem,
                            //                    CHAN BYTE out!)

proc test (chan byte screen!)
{
    #pragma shared screen
    SEMAPHORE scr_sem;
    #pragma shared scr_sem

    seq {
        initialise_semaphore (scr_sem, 1);
        par (i = 0 for 10) {
            sem_test (i, scr_sem, screen!);
        }
    }
}
```

As shown in this example, compiler directives are formed in much the same way as occam, but in lower-case. The placement of some of these is critical — '`#pragma shared`' must still be placed immediately after the declaration, for example.

Chapter 4

Mobile Data, Channels and Processes

This chapter describes *mobiles* — a new layer added to the occam type-system which provides data, channel, and process mobility. Communication and assignment of mobiles follow a *movement* semantics, i.e. a variable output in communication or on the right-hand side of an assignment is left *undefined* — it has moved. The implementation uses this to an advantage: in a shared-memory UMA (Uniform Memory Access) machine, references may be communicated and assigned, significantly reducing the (local) run-time cost of these operations.

A fuller introduction to mobiles, including the motivation for adding them to the occam language, is given in section 4.1. Section 4.2 describes mobile *data-types* in detail, including a description of *mobilespace* — where *static* mobiles are allocated. Mobile data-types in occam were first presented in [BW01a] and [BW01b].

Sections 4.3 and 4.4 present mobile *channel-types*, followed by a description of mobile *processes* in section 4.5. Mobile channels in occam were first presented in [BW02a] and [BW02b], and subsequently with mobile processes in [BW03].

With a movement semantics, in particular the ability for a variable to become *undefined*, additional usage checks are required to ensure the correctness of code. Such checks are implemented by the *undefinedness* checker, described in section 4.6. The undefinedness checker, in addition to analysing mobile variable usage, implements checks for some of occam extensions from chapter 3.

A further checking mechanism for mobile channel-types is proposed in section 4.7, but has not yet been implemented.

4.1 Introduction

Communication and assignment in traditional occam systems incur an $O(n)$ run-time cost for data-copying. In small localised systems (i.e. those which share a common memory), this run-time cost is often undesirable and often unnecessary.

A common solution to this, in occam, is to implement a memory *pool* which uses `RETYPE`s and various compiler-builtins to generate pointers from variables. These pointers (usually represented by the `INT` type) can then be communicated and assigned in $O(1)$ time. The disadvantage of this method is that the compiler can no longer perform parallel-usage and alias checks on the pointers — it just treats them as plain `INT`s. Thus the potential for introducing parallel race-hazards is high.

However, in traditional occam, this is often the only solution.

Mobiles, with their movement semantics, provide a simple and effective solution to this — with an implementation that uses communication and assignment of references and has aliasing strictly controlled. These semantics are enforced by the compiler both at compile-time (in the undefinedness checker, section 4.6), and also at run-time, by invalidating 'lost' references.

Mobiles are not a new concept — movement semantics exist in other languages, for example NIL [SY85], although these tend to be few in number. The programming language Icarus [MM98b, MM98a, MM01] also supports a movement semantics, for both data and channels. Icarus also introduces the concept of a *borrowing* semantics, that provides an explicit mechanism for temporarily *moving* data/channels to another process and back at defined points. A similar mechanism also exists for mobile types in occam, although it is somewhat implicit — in procedure calls and abbreviations.

For channels, communication and assignment using a movement semantics is the only option — copy semantics have no meaning when applied to channels, and are therefore banned in the standard occam language. Mobile channel-types provide this functionality, and also extend it — by allowing explicitly *shared* channel-types (for the construction of *one-to-any*, *any-to-one* and *any-to-any* channels). Section 4.3 describes mobile channel-types in detail.

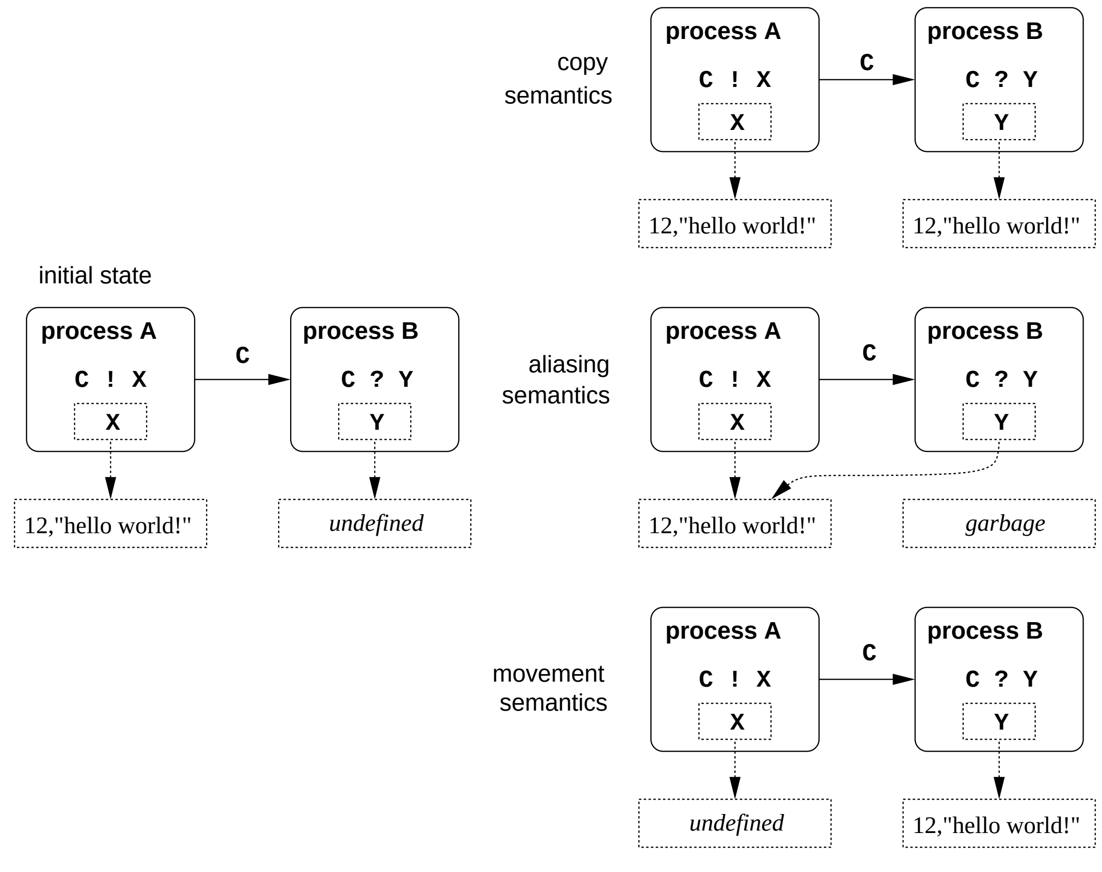


Figure 4.1: Copying, aliasing and movement semantics for communication

Figure 4.1 shows three semantic models for communication: copying, aliasing and moving. Standard occam provides copying semantics only. JCSP [WA99, Wel00] exhibits aliasing semantics, as do languages such as C and C++ (when concurrency is added). For these languages, enforcing strict aliasing is hard — requiring specialist tools (such as that provided with the commercial JCSP.net [WAF02]), or modifications to the language in order to provide aliasing control. The C99 language specification [Ame99] adds a '`restrict`' type qualifier which specifies single-reference semantics for pointers. Checking and enforcing such single-reference pointers is hard however, as noted in [FA01].

4.2 Mobile Data Types

Mobile data-types in occam fall into two categories: static and dynamic. Static mobiles are mobile data-types whose size is known at compile-time. Dynamic mobiles are mobile data-types whose size is *not* known at compile time. In the occam language, the only candidates for dynamic mobiles are unsized array *declarations*. Traditional occam rejects unsized array declarations, due to the lack of a dynamic memory functionality. Dynamic mobile data-types, described in section 4.2.4, are a somewhat special case since they only apply to array-types.

The default memory allocation strategy in occam is static. After analysing each `PROC` or `FUNCTION`, the compiler knows exactly how much memory is required to support it at run-time. The occam run-time model splits memory into two parts, *workspace* and *vectorspace*. The occam workspace behaves much like a C stack, containing local variables and growing downwards. Unlike C, however, the maximum size of this 'stack' is known.

In order to keep workspace offsets small at run-time, the occam compiler allocates large data items in vectorspace — typically fixed-sized arrays and `RECORD` types. The current 'cut-off' point for workspace allocation is at 8 bytes. Anything larger is allocated in vectorspace.

To fit with this static memory-model, mobile (data) variables of a known size are allocated in *mobilespace*. Ordinary workspace or vectorspace allocation for mobiles is not suitable, since a mobile variable may out-live the scope of its declaration — if it is *moved* to another variable or process through assignment or communication. Mobilespace is engineered to handle this safely, without 'losing' references and without introducing aliases through assignment and communication. The layout and management of mobilespace is discussed in section 4.2.2.

4.2.1 Declaring Mobiles

Mobile data types are declared using the '`MOBILE`' type qualifier. Mobility is added to types in occam— not variables. This means, for instance, that channels which carry mobiles must have a `MOBILE` protocol. Mobile types are sub-types of their non-mobile counterparts, such that a variable of type '`MOBILE X`' may be used wherever a variable of type '`X`' is valid.

The reverse is not so — a channel of '`MOBILE X`' cannot be used to communicate a variable of type '`X`', only '`MOBILE X`'s are valid.

Mobile variables are declared in the standard way, but are declared to be of `MOBILE` types. For example:

```
MOBILE INT i, j:
MOBILE REAL64 x, y:
MOBILE [32]REAL32 results:
MOBILE SOME.USER.TYPE var:
```

User-defined types, introduced in occam 2.1 [Inm95], may also be declared `MOBILE`:

```
DATA TYPE DATASET IS MOBILE [128]REAL64:
DATA TYPE PACKET
  MOBILE RECORD
    INT src, dest:
    [512]BYTE data:
:

...  other declarations

DATASET results:
PACKET p:
...  process using ‘results’ and ‘p’
```

The variables ‘results’ and ‘p’ are automatically MOBILE in this example. However, without redefining the types, creating non-mobile DATASET and PACKET variables is not possible. It may therefore be convenient to declare the types non-mobile and the variables as MOBILE versions of those types. For example:

```
DATA TYPE DATASET IS [128]REAL64:
DATA TYPE PACKET
  RECORD
    .. same fields as PACKET above
:

...  other declarations

MOBILE DATASET results:
MOBILE PACKET p:
...  process using ‘results’ and ‘p’
```

Mobile parameters and abbreviations follow along similar lines — i.e. using the MOBILE keyword:

```
PROC foo (MOBILE [32]REAL32 data, PACKET message)
  ...  body of foo
:

MOBILE INT z IS x:
...  process using ‘z’
```

The semantics of MOBILE parameters and abbreviations is one of borrowing — during the duration of a PROC instance, or the lifetime (scope) of an abbreviation, a mobile is *borrowed*. When the procedure returns or the abbreviation descopes, mobiles are moved back into their original variables (e.g. ‘x’ in the above abbreviation).

4.2.2 Mobilespace

As noted previously, mobile data objects cannot be allocated in the workspace (or vectorspace) of a process (approximately equivalent to the C stack and heap in occam). Dynamic allocation is not required: the type and size of all *static* mobiles are known (and dynamic allocation is an overhead). The solution presented here provides allocation for statically sized mobiles within a *mobilespace* — a statically sized memory area (much like workspace or vectorspace), allocated when the run-time system initialises.

The allocation of mobile variables to mobilespace happens during the *mapping* phase in the compiler — where variables are normally allocated to either workspace or vectorspace. Mobilespace is simply a third allocation option in this respect. The allocation strategy for mobilespace is, however, significantly different from the workspace and vectorspace allocation strategies. The lifetime of a variable allocated in mobilespace is controlled through communication and assignment; not through variable scoping alone.

Each mobile variable declared (including anonymous mobile variables inserted by the compiler), is allocated in mobilespace along with room for a pointer — the *shadow-variable*. Additionally, space for a local pointer is allocated in the process's workspace, that is used to hold the address of a mobile variable (locally) at run-time. Figure 4.2 shows this allocation strategy for an example `PROC` with some mobile variables. The use of shadow variables in mobilespace may not seem immediately obvious, but is required to implement mobile variables correctly (with zero-aliasing), discussed further in section 4.2.3.

```
PROC P (..., $msp)
  DATA TYPE FOO
    MOBILE RECORD
      [4]INT dest:
      MOBILE [32]BYTE payload:
  :
  FOO a, b:
  SEQ
    ... process using a and b
:
```

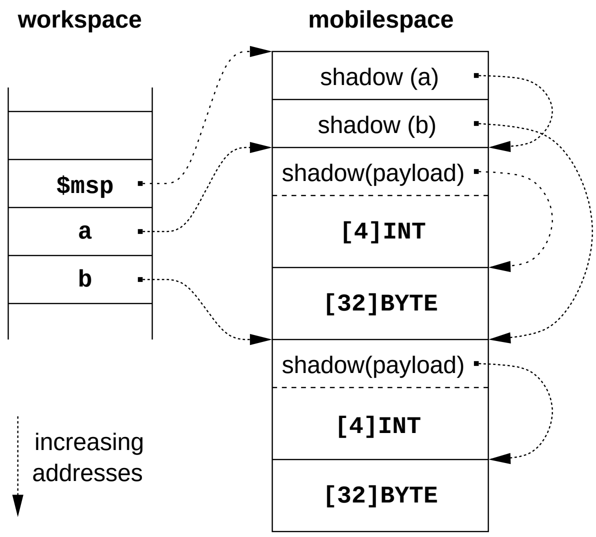


Figure 4.2: Allocation of variables in mobilespace

The mobilespace allocated for any instance of '`P`' is passed as a hidden parameter to the `PROC` — in much the same way as the hidden vectorspace parameter ('`$vsp`'). The hidden mobilespace parameter is called '`$msp`'.

In addition to mobile variables, mobilespace can hold nested-mobilespaces (for `PROC` instances requiring mobilespace), and run-time 'hooks' (for storing dynamically allocated mobilespaces required by dynamic processes — covered in Chapter 5). Allocation of nested mobilespaces is trivial — the mobilespace required by the called procedure is simply allocated as a single block in the caller's mobilespace.

Run-Time Initialisation

The initialisation of mobilespace (to setup the shadow pointers, as shown on the RHS of figure 4.2), is handled at run-time. When the mobilespace for the top-level `PROC` (and thus the mobilespace for all its sub-processes) is allocated, the *whole* space is set to '`MOSTNEG INT`' — traditionally the "invalid pointer" on a Transputer (which had a signed address space). This value is often just referred to as 'MINT', from the Transputer instruction that generates it. A similar technique is used for dynamic mobilespaces too, covered in section 4.2.4. The references from the workspace pointers 'a' and 'b' to their real mobilespace memories is handled when those variables enter and leave scope, described in section 4.2.3.

When the top-level PROC (or any mobilespace requiring PROC) is entered, the first word of mobilespace (accessed through the hidden '$msp' parameter), is checked. If set to 'MINT', then the mobilespace is initialised. Otherwise, the mobilespace has already been initialised (from a previous call to the same PROC).

To generate code for mobilespace initialisation, the compiler uses the 'MOBILEINIT' ETC directive (described in appendix B.2.1), generated immediately after the PROC's entry-point. This directive provides the translator (tranx86) with the constant workspace offset of the '$msp' parameter, the number of initialisations to perform, followed by a list of (shadow-offset, data-offset) pairs. For the process shown in figure 4.2 this mobilespace initialisation sequence will be (in textual ETC):

```
.COMMENT         $msp offset is 3, count is 4
.MOBILEINIT      3  4
.MOBILEINITPAIR  0  2
.MOBILEINITPAIR  1  11
.MOBILEINITPAIR  2  7
.MOBILEINITPAIR  11 16
```

Only mobile variables (and some dynamic mobilespace constructs) require pointer initialisation in mobilespace. The mobilespace required by any nested PROC calls is mapped as a single block — the run-time system guarantees that all global mobilespace is initialised to 'MINT', handling marking of non-intialisation for PROC calls. Instead of using a shadow variable (and local mobile) to hold the mobilespace address for a PROC instance, the (constant) offset within the current mobilespace is linked directly to that particular PROC call. Figure 4.3 shows the mobilespace layout for an example process 'Q', that declares and uses its own (mobile) 'FOO' variable, then runs two instances of 'P' in parallel. The definitions for 'FOO' and 'P' are shown in figure 4.2.

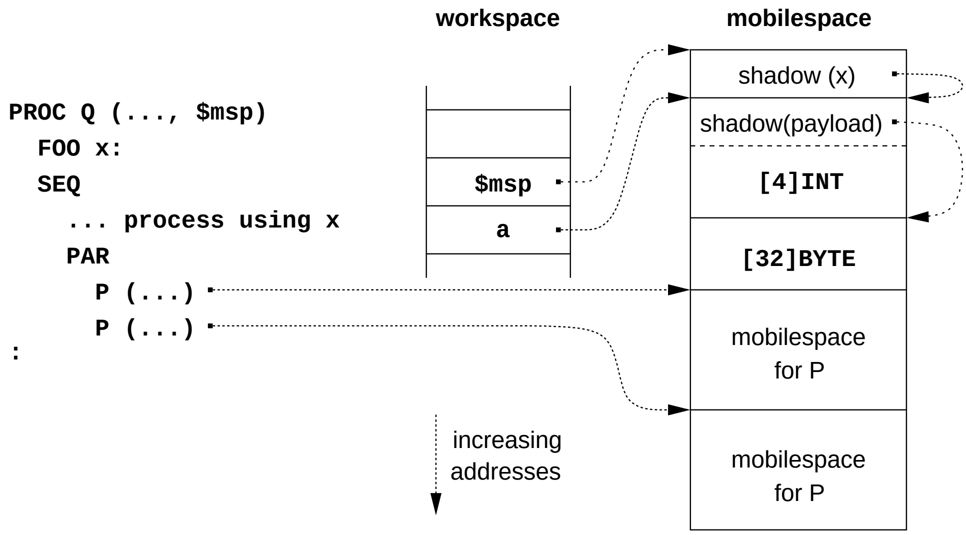


Figure 4.3: Allocation of process instances in mobilespace

The mobilespace initialisation code for the 'Q' process needs only to initialise the 'x' variable. The initialisation of the two nested 'P' calls are handled by 'P' itself, as shown before. The (initialisation) code generated for 'Q' is simply:

```
.COMMENT          $msp offset is 3, count is 2
.MOBILEINIT       3  2
.MOBILEINITPAIR   0  1
.MOBILEINITPAIR   1  6
```

The translator turns such sequences into real code, that performs the check for MINT in the first word, and if so, initialises mobilespace according to the data (mobilespace-map) given. Although the occam compiler could generate mobilespace initialisation code itself, such code after translation will be less optimal — due to the repeated need to load the '$msp' parameter for each initialisation. The actual (annotated) code generated for the above initialisation sequence, on the Intel i386 platform, is:

```
    ;; $msp offset is 3, count is 2
    ;; .MOBILEINIT 3 2
    movl    12(%ebp), %eax          ;; load $msp
    cmpl    0(%eax), $0x80000000
    jne     0f                      ;; skip if already initialised

    ;; .MOBILEINITPAIR 0 1
    leal    4(%eax), %ebx
    movl    %ebx, 0(%eax)

    ;; .MOBILEINITPAIR 5 6
    leal    24(%eax), %ebx
    movl    %ebx, 20(%eax)
0:
```

In theory, there are two places where mobilespace initialisation can be performed. Either on PROC entry — where it currently happens — or on individual mobile variable scope-in and scope-out. The latter approach is used by the occam compiler for initialising channel words. This is a consequence of the allocation strategy for workspace and vectorspace — *overlapping* static activation records (ultimately stemming from Algol-60 [RR64]). Since the location of channels within workspace and vectorspace may change (and may be subsequently overlaid with other declarations), initialisation (of workspace and vectorspace entities) on PROC entry is not an option. For some code however, separate initialisation (and allocation) of channel words would be beneficial — both in reducing the number of un-necessary initialisations, and potentially improving performance on multi-processor machines (particularly in cases of *false sharing* — where a cache line is un-intentionally ping-pong'd between processors).

Minimising Mobilespace

By contrast, the allocation strategy for mobilespace is *non-overlapping* static activation records. Unlike workspace and vectorspace, simultaneous usage of mobilespace is unpredictable — references to an initially local mobilespace will move during communication, and in return, references to remote mobilespaces will be aquired. As such, the free-wheeled recycling of mobilespace would cause significant memory corruption, and does not fit with the initialisation on PROC entry. Some re-use is possible however, by *packing* sequentially used, same sized, closely typed[1] mobile variables. For example:

```
PROC thing (...)
  SEQ

    MOBILE [64]BYTE data:
    MOBILE [3]REAL64 p1, p2:
    PAR
      ...  process using ‘p1’ and ‘data’
      ...  process using ‘p2’

    MOBILE [32]INT16 sdata:
    MOBILE [3]REAL64 x:
    SEQ
      ...  process using ‘x’ and ‘sdata’
:
```

In sequential code such as this, the mobilespace allocated for ‘`p1`’ or ‘`p2`’ can be reused to allocate ‘`x`’. Less obviously, the mobilespace allocated for ‘`data`’ can be reused for ‘`sdata`’, since the two variables require exactly the same amount of memory. A similar strategy is also applied to sequential `PROC` calls.

Declarations of user-defined (‘`RECORD`’) mobile variables can also be packed in a similar way, providing that there are either no nested mobiles, or that any nested mobiles match in a similar manner. Unless the ‘`PACKED`’ attribute is specified on `MOBILE RECORD` types, the compiler will re-order the fields to put nested mobiles first when allocating mobilespace. This can help considerably when packing mobilespace. Consider for example:

```
DATA TYPE S.BLK
  MOBILE RECORD
    INT64 blockno:
    MOBILE [512]BYTE data:
:

DATA TYPE BLK.MAP
  MOBILE RECORD
    MOBILE [128]INT blocks:
    INT n.blks:
    INT32 flags:
:
```

After sorting the fields of these types, sequentially declared variables of either type can be packed in mobilespace. When a record has multiple `MOBILE` fields, the fields are sorted in decreasing order by size. This makes the check for packing compatability largely obvious to the compiler.

There are potentially many cases where two types are not directly compatible for mobilespace packing, but where a minor alteration to one would make them compatible. Alterations would principally be the insertion of extra (dummy) fields, either mobile or non-mobile. Promotion of a

[1] “Closely typed” is somewhat of a vague term — two mobile types are compatible for packing if they have the same fixed size, and the number/placement of nested-mobiles is the same, and that those pairs of nested mobiles are also ‘closely typed’.

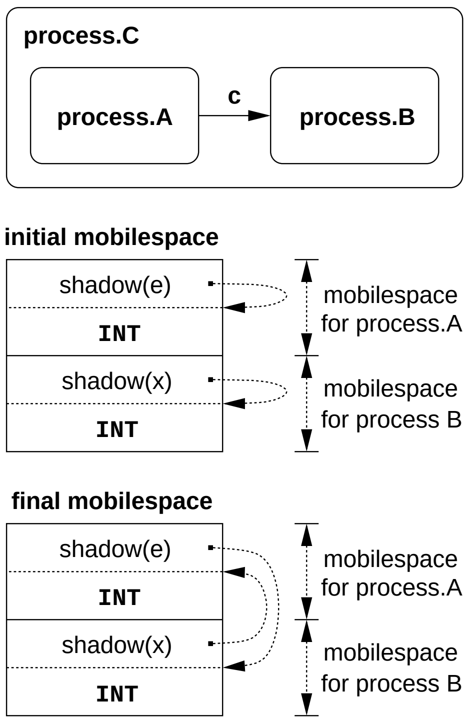
non-mobile to a mobile is also a potential option (when looking for compatible types for mobilespace packing), but requires some careful handling in order to not destroy existing copy semantics. Since nested mobilespaces are currently only partially supported, the implementation of such packing has not been investigated in great details. These issues are covered in greater detail in section 8.3.2.

4.2.3 Scoping, Communication and Assignment

When a mobile variable comes into scope, the pointer in the mobilespace *shadow variable* is copied into the corresponding pointer in the process's workspace. Similarly, when a mobile variables leaves scope, the pointer in workspace is copied back into the mobilespace shadow variable.

The implementation of communication and assignment of mobiles, within a shared-memory system, is handled using *pointer swapping* of the two mobile variables involved. In CSP, communication and assignment have a direct equivalence, enforcing on occam the same semantic effects for both communication and assignment of mobiles. The use of pointer swapping yields two important properties for mobile communication and assignment. Firstly, all *mobile* communications and assignments are of $O(1)$ cost, and secondly, whenever a reference (to mobilespace) is lost, another is gained — the symmetric effect of swapping. Conceptually, communication and assignment are one-way operations, which the occam compiler enforces by *undefining* outputted mobile variables (or mobile variables used on the right-hand-side of assignments). The undefinedness checker, described in detail in section 4.6, implements these checks.

To demonstrate the effect of this implementation, figure 4.4 shows a network comprised of two processes along with mobilespace as a whole for that network.




```
DATA TYPE M.INT IS MOBILE INT:

PROC process.A (CHAN M.INT out!)
  M.INT e:
  SEQ
    ... define 'e'
    out ! e
:

PROC process.B (CHAN M.INT in?)
  M.INT x:
  SEQ
    in ? x
    ... use 'x'
:

PROC process.C ()
  CHAN M.INT c:
  PAR
    process.A (c!)
    process.B (c?)
:
```

Figure 4.4: Mobile-communicating process network showing initial and final mobilespace

When these processes are run, they synchronise and *swap* their local references to '`e`' and '`x`' respectively. Thus, when these variables descope and are copied back into mobilespace shadows, they remain swapped — as shown by the difference between the initial and final mobilespaces in figure 4.4. If these two processes were subsequently run again, the swap would return mobilespace to its original configuration.

During the course of program execution, the pointers within mobilespace can migrate as far as their bi-directional communication and assignment closures extend. Since no pointers are ever gained or lost however, mobilespace remains 'sound', with every internal shadow pointer being valid.

Due to the loading and storing of mobile shadow variables into local pointers, mobilespace may occasionally contain temporary aliases. These are temporary until the affected variable descopes and stores a non-aliased pointer back into mobilespace. To demonstrate this, figure 4.5 shows the state of the mobilespace from figure 4.4 at the point where '`process.A`' has terminated, but where '`process.B`' is still executing inside the '`... use 'x'`' block of code.

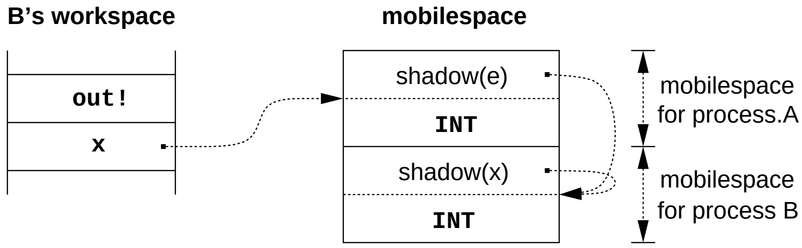


Figure 4.5: Mobilespace with temporary aliasing

Support for mobile communication (using pointer-swapping), is provided through a new set of communication instructions. A list of these new instructions can be found in table B.2 in the appendix, with more information on each in appendix B.1.2.

The principal instructions for mobile communication are '`MIN`' and '`MOUT`', for mobile input and output respectively. These new instructions follow the usage pattern of existing communication instructions '`IN`' and '`OUT`', except that the size (of data communicated) need not be given explicitly.

Pointer swapping for mobile assignment is simpler and does not require any new instructions. For example, the mobile assignment '`x := e`' is handled at run-time with the following code:

```
LD x
LD e
ST x
ST e
```

Copying Mobile Variables

In many cases, it may be desirable (and even necessary) to *copy* (rather than move) a mobile variable — such that the mobile variable output, or on the RHS in assignment, remains defined. The '`CLONE`' operator is provided for this purpose.

'`CLONE`' is a compiler built-in operator (as opposed to a user-defined operator [WM99]) that operates over *all* mobile types. There are several restrictions in the use of this operator however — it is not allowed inside expressions, for example.

The previous *moving* assignment '`x := e`' (where '`x`' and '`e`' are mobile variables) can be changed into a copy by inserting `CLONE` on the RHS:

```
x := CLONE e
```

From CSP, this is equivilant to the following (and illegal) parallel processes:

```
CHAN c:
PAR
  c ? x
  c ! CLONE e
```

The implementation of the CLONE operator in assignment is trivial — the data is simply copied. Communication is slightly trickier — the inputting process will always input via pointer-swapping, using the mobile communication instructions. In order to be compatible, the outputting process must also implement output via pointer-swapping. The above PAR is translated[2] inside the occam compiler into:

```
CHAN c:
PAR
  c ? x

  MOBILE ... $anon:
  SEQ
    $anon := CLONE e
    c ! $anon
```

4.2.4 Dynamic Mobile Arrays

The mobile data-types presented so far have been fixed-size. However, one of the main motivations for adding mobile support to occam was to provide a mechanism for securely managing dynamically allocated memory. In standard occam [Inm84b, Inm95], there is only one candidate for dynamic *data* allocation — run-time sized arrays. Dynamic mobile arrays provide this functionality and are simply declared as un-sized MOBILE arrays. For example:

```
MOBILE []BYTE a:
```

This declares a run-time sized *mobile* array of BYTEs called 'a'. Type-declarations are also permitted, providing (finally) a reasonable implementation for a 'STRING' type in occam:

```
DATA TYPE STRING IS MOBILE []BYTE:

STRING str:
...  process using 'str'
```

Unlike many other languages, there is nothing special about this 'STRING' type — it is simply a run-time sized mobile array of bytes.

Dynamic allocation for these run-time sized arrays is handled using the dynamic memory instructions introduced in section 3.8 (detailed in appendix B.1.1). The *static* memory requirement for a dynamic mobile array is a compile-time constant number of workspace slots — one for a pointer and one for each dimension (to hold the run-time size). Neither mobilespace or vectorspace is required.

The run-time behaviour of dynamic mobile arrays is much different from static mobiles. Since the allocation is dynamic, the possibility for invalid or aliased pointers exists — but only when

[2] Translation here is a phase in the compiler, where operations such as sequential and tagged protocol expansion occur.

a dynamic mobile is *undefined*. For static mobiles, the pointer in workspace is always valid, even though the data it points to may be undefined. To handle run-time undefinedness for dynamic mobiles, the obvious solution would be to use null-pointers — inserting run-time checks where necessary. However, this does not fit well with the compiler's existing run-time handling for arrays — primarily array-bounds checking where a dimension size is unknown (possible for parameters and abbreviations). Instead, when a dynamic mobile array enters scope, its first dimension count is initialised to zero. Any attempt to access the array in its undefined state will result in a run-time error (specifically an array-bounds violation). When a dynamic mobile array leaves scope, the first dimension count is checked for non-zero, and if so, the pointer is released.

Run-time allocation for dynamic mobile arrays is either explicit (allocation of a specific size) or automatic (from a non-mobile compatible assignment). For example:

```
MOBILE []BYTE s1, s2:
SEQ
  s1 := MOBILE [42]BYTE        -- explicit allocation
  s2 := "Hello, new world!"    -- automatic allocation
  ...  process using ‘s1’ and ‘s2’
```

Such allocation is always performed through assignment. The communicative forms of these allocations are also permitted, that the compiler re-writes to produce allocation by assignment (using a temporary). For example:

```
MOBILE []BYTE s1, s2:
SEQ
  CHAN MOBILE []BYTE c:
  PAR
    c ! MOBILE [42]BYTE
    c ? s1
  CHAN MOBILE []BYTE c:
  PAR
    c ! "Hello, new world!"
    c ? s2
  ...  process using ‘s1’ and ‘s2’
```

```
MOBILE []BYTE s1, s2:
SEQ
  CHAN MOBILE []BYTE c:
  PAR
    MOBILE []BYTE $anon0:
    SEQ
      $anon0 := MOBILE [42]BYTE
      c ! $anon0
    c ? s1
  CHAN MOBILE []BYTE c:
  PAR
    MOBILE []BYTE $anon1:
    SEQ
      $anon1 := "Hello, new world!"
      c ! $anon1
    c ? s2
  ...  process using ‘s1’ and ‘s2’
```

The above shows the original (user) code on the left, with the re-written version on the right. When compiled, the first parallel process in the above will produce an undefined variable warning — at the point where ‘`$anon0`’ is output in the re-written version. Although the pointer for ‘`$anon0`’ may be valid, the contents of the array are undefined.

Implementation Aspects

The semantics of dynamic mobile array assignment and communication are the same as for static mobiles. Their implementations differ however. Instead of pointer-swapping to implement assignment and communication, a simple copy of the pointer and dimension sizes is used. Before input, a dynamic mobile array is checked for *definedness* and the pointer freed if necessary. Correspondingly, after output, the first (local) dimension size of a dynamic mobile is set to zero. To implement communication, a separate group of instructions are used: '`MIN64`', '`MOUT64`', '`MINN`' and '`MOUTN`', described in appendix B.1.2. Assignment follows suite, with the LHS being freed if necessary before the assignment, and the RHS's first dimension size being set to zero after the assignment.

The undefinedness checker (section 4.6) additionally inserts special markers to indicate undefinedness, potentially avoiding the conditional free before assignment or input.

Figure 4.6 shows the memory allocation for a typical dynamic mobile array, after having been allocated. This positioning of the dimension sizes follows from a similar mechanism used to pass unsized array parameters. The pointer appears first, followed by the unknown dimensions in successive workspace slots.

```
MOBILE [][]REAL64 m:
SEQ
  m := MOBILE [16][16]REAL64
  ...  process using 'm'
```

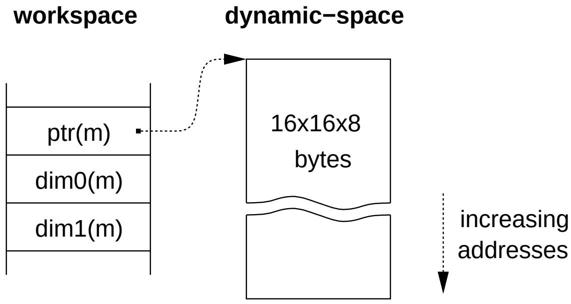


Figure 4.6: Dynamic mobile array memory allocation

Supporting the `CLONE` operator for dynamic mobile arrays is slightly more complicated than for static mobiles. In a similar way to static mobiles, `CLONE` can only be used in a direct assignment — communication using `CLONE` is simplified by the compiler into a temporary and an assignment. For example:

```
MOBILE []BYTE s1, s2:
SEQ
  s1 := "hello, dynamic world"
  s2 := CLONE s1

  ...  process using 's1' and 's2' (both defined)
```

The implementation of the `CLONE` for dynamic mobile arrays conditionally frees the target variable ('`s2`' in the above code), calculates the size of the `CLONE` operand ('`s1`'), allocates that much for the target, then performs a simply copy between the two pointers (using the '`MOVE`' instruction). The (annotated) compiler output for the above `CLONE`d assignment is:

```
  --{{{  conditional free
  LD    SIZE s2          -- loads the first dimension count
  CJ    :Ln              -- jump if zero, otherwise
  LD    s2               -- load address
  MRELEASE               -- release dynamic memory
:Ln
  --}}}
```

```
--{{{  calculate size and store in Wptr[Temp] (offset 0)
LD    SIZE s1          -- loads the first dimension count
DUP
ST    SIZE s2          -- store dimension count in target
LDC   BYTESIN (BYTE)   -- load size of underlying type (BYTE)
MUL                    -- multiply to get whole size
STL 0                  -- store in Wptr[Temp]
--}}}
--{{{  allocate new block and copy
LDL   0                -- previously stored size
MALLOC                 -- allocate dynamic memory
DUP
ST    s2               -- store pointer
LD    s1               -- load source address
REV                    -- swap order (src in Breg, dest in Areg)
LDL   0                -- load size
MOVE                   -- copy data
--}}}
```

This is a relatively optimised form of the `CLONE` — more expansive implementations are possible. The code generated by the compiler is actually a little more optimal than this — since the underlying type is `BYTE`, the compiler will optimise away the sequence "`LDC BYTESIN (BYTE); MUL`". Occasionally the compiler will generate such code by other means — particularly when the 'load constant 1' and 'multiply' are separated in the code (as can be generated by an `INLINE FUNCTION`). To cater for these, the translator (tranx86) will optimise away such code.

4.3 Mobile Channel Types

Mobile channel types provide mobilisation of occam channels: that is, the ability to communicate channels over channels, or rather to communicate *channel ends* over channels. This has long been a desirable feature in occam, since it enables the controlled reconfiguration of process networks at run-time. The Icarus language provides similar support [MM98b], where channel ends are termed *ports*.

Rather than limiting the *end* to being a single channel-end, mobile channel types declare a *channel bundle*, grouping together a number of related channels. Channels may be used in either direction, but that direction must be explicitly specified. For example:

```
CHAN TYPE ENCODE
  MOBILE RECORD
    CHAN MOBILE []BYTE raw?:
    CHAN MOBILE []BYTE cooked!:
:
```

This defines an '`ENCODE`' mobile channel type, consisting of two channels, one in either direction. The variables of this channel type are its *ends*. Given the mixed direction of channels, it does not make obvious sense to apply the terms '*input*' and '*output*' to the ends. Instead, the terms '*server*' and '*client*' are used. This is indicated when declaring variables and parameters of a particular channel type, for example:

```
ENCODE? svr:               -- server end
ENCODE! cli:               -- client end
```

The use of such terms for the ends implies a relationship between the processes using them. This is largely non-strict, the importance to the compiler being to distinguish between the two ends, rather than the usage of those ends. The client-server paradigm fits particularly well in the majority of perceived cases, however. Additionally, if used correctly, it provides a guarantee of deadlock freedom [WJW93].

Such connectivity, from the clients perspective, is almost that of a mobile server *process*, but rather than the server process being communicated between clients, the connection to the server is communicated. Figure 4.7 shows an example of such connectivity, where the client processes communicate the channel-bundle end between themselves to control server access.

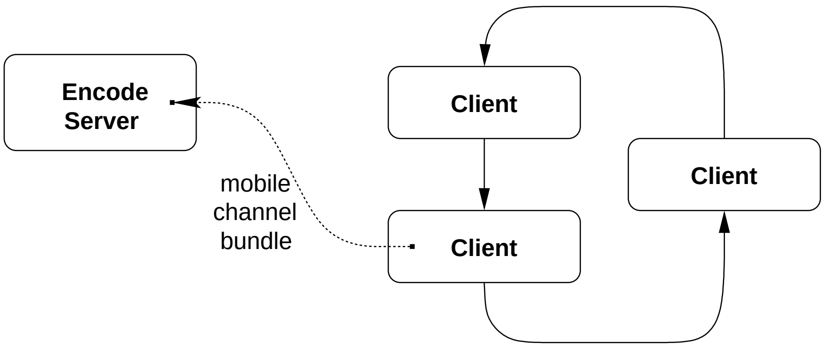


Figure 4.7: Example 'encode' server and clients connected using mobile channel-ends

A more general paradigm often desired is that of a *shared* server process — which the above does not provide directly. Traditionally, such connectivity was managed in one of two ways: either provide explicit connections between the server and all clients, on which the server `ALT`s; or use a number of explicitly *shared* channels (shared with a compiler directive that disable usage checking), access to which can be controlled through the user-defined '`SEMAPHORE`' type [WM99].

There are restrictions with both of these approaches, however. A server that `ALT`s will incur an $O(n)$ cost for servicing clients (where n is the number of clients). Furthermore, fairness in such servers must be explicitly programmed (using a *fair-ALT* for example). Explicitly shared channels, that have usage-checking disabled, suffer from a lack of compile-time checks. It is then up to the programmer to use the semaphore, or other access mechanism, correctly — a burden on the programmer, who would not normally want to be concerned with such things.

To solve these problems, a mechanism for *sharing* a channel-bundle end is provided. This allows controlled access to a shared channel-bundle by multiple clients, that queue *fairly* to gain mutually exclusive access to the channel-ends within (and hence, access to the 'server'). A single server process at the other end simply has an *unshared* bundle of channel-ends, unaware of the potentially many competing client processes. More generally, a mechanism for sharing either the client or server end of a channel-bundle is provided, allowing all permutations of a shared/non-shared clients and servers. This is similar to the `occam3` support for shared channel entities. However, no specific mechanism to support `ALT`ing on the '*claim*' is provided in this implementation — any shared channels must be claimed before use in either input, output or `ALT`s.

A more detailed examination of the declaration and initialisation of mobile channels is given in section 4.3.1. Section 4.3.2 discusses the usage of mobile channels, in particular, those whose ends are shared. The implementation of the semaphore mechanism that handles sharing of channel-bundle ends is covered in section 4.3.3.

4.3.1 Declaration and Initialisation of Mobile Channels

As already shown, channel-bundle ends are declared using a combination of the channel-type name (‘ENCODE’ for example) and an indicator of whether the end is the server-end (‘?’) or the client-end (‘!’) of the bundle. To declare shared ends, the keyword ‘SHARED’ is placed before the type name. For example:

```
SHARED ENCODE? s.svr:        -- shared server end
SHARED ENCODE! s.cli:        -- shared client end
```

To use channel-bundle ends, a set of channels must be allocated, initialised and assigned to the variables representing the ends. This is done by means of a somewhat special MOBILE allocation construct (similar to that used for dynamic mobile array allocation, section 4.2.4). For example:

```
ENCODE? encode.svr:          -- unshared server end
SHARED ENCODE! encode.cli:   -- shared client end

SEQ
  encode.svr, encode.cli := MOBILE ENCODE
  PAR
    ...  process using ‘encode.svr’
    ...  processes using ‘encode.cli’
```

This form of dynamic channel allocation is the same, regardless of whether the ends being connected are shared or unshared. Besides the obvious type compatibility checks, the compiler ensures at there is one client and one server end on the left-hand side of the assignment. Internally, this assignment is transformed into a call to a compiler built-in:

```
ALLOC.CHAN.TYPE (ENCODE, encode.svr, encode.cli)
```

The programmer can call this built-in directly, if desired. Using the assignment form is preferable however, since it clearly defines the semantics of the operation — the creation of a mobile channel bundle whose ends are assigned *simultaneously* to the two end-variables.

Figure 4.8 shows the structure of the memory allocated for this mobile ‘ENCODE’ channel bundle — supporting a shared client end and a non-shared server end. The extra memory that would be required for supporting a shared server end is also shown.

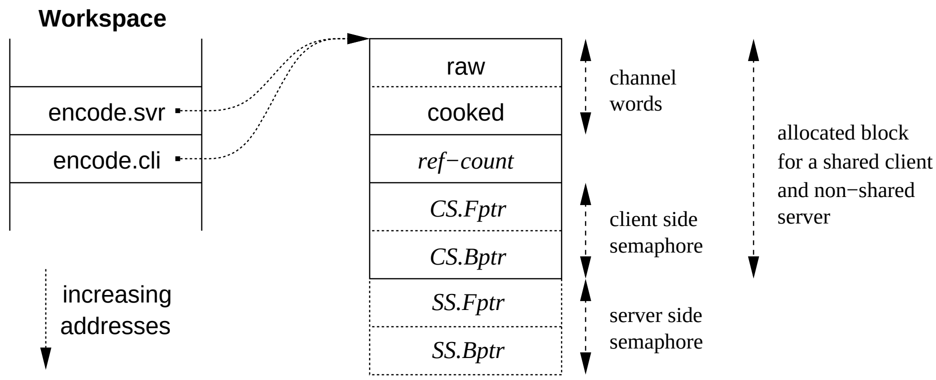


Figure 4.8: Layout of the ‘encode’ channel type in memory

The initialisation of this structure is trivial. For all cases, unshared, part-shared or fully-shared, the channel words are initialised to 'NotProcess.p' (zero in this version of the occam compiler), and the '*ref-count*' (reference count) field is initialised to 2 (the two variables in the process workspace). The remainder of memory allocated consists of semaphores to handle sharing — the client semaphore then the server semaphore. In cases where only the client end is shared, space for the server semaphore is not allocated. For the two remaining cases, both ends shared and a non-shared client end connected to a shared server end, the full space is allocated — a shared server will not know whether the client end is shared or not, so the space required for a potentially shared client end must be reserved. Any allocated semaphores have their '*Bptr*' field set to 'NotProcess.p', and their '*Fptr*' field set to 1. This implementation of semaphores uses the '*Fptr*' field to double as the semaphore-count (hence why it is initially 1). This is a perfectly reasonable optimisation, since when '*Fptr*' is used to hold a process, the count will be zero. The main benefit of this approach is that the semaphore only occupies two words of memory, instead of three, that align better with themselves and other data (if allocation was done with cache-line boundaries in mind). The implementation of this semaphore mechanism is covered in section 4.3.3.

As for previous dynamic mechanisms, the memory is allocated and freed using the Brinch-Hansen style block allocator, discussed in section 3.8. The '*ref-count*' field is used to keep track of the number of references, most useful when one or both ends are shared. When a channel-type end variable descopes, and contains a valid pointer (to such a structure), the '*ref-count*' field is decremented and the memory freed if it reaches zero (a simple reference-counted garbage collection mechanism [JL97]).

4.3.2 Communicating Mobile Channels

The channels within a mobile channel bundle are accessed in the expected manner, with a record-style subscription, e.g. 'encode.svr[raw]'. The directions specified on channels in the type declaration represent the server-end usage for those channels. Client ends are accessed in the reverse manner. For the original 'ENCODE' declaration:

```
CHAN TYPE ENCODE
  MOBILE RECORD
    CHAN MOBILE []BYTE raw?:
    CHAN MOBILE []BYTE cooked!:
:
```

This ensures that the server process *inputs* from 'raw' and *outputs* to 'cooked'; whereas clients output to 'raw' and input from 'cooked'. The outline of a server process, that takes an unshared server-end of an 'ENCODE' channel-bundle might be:

```
PROC encode.server (ENCODE? link)
  WHILE TRUE
    MOBILE []BYTE s:
    SEQ
      link[raw] ? s
      ...  do processing on data in 's'
      link[cooked] ! s
:
```

For shared channel ends, the bundle as a whole must be *claimed* before accessing any of the channels within — i.e. before any subscriptions can take place. The syntax engineered into the compiler for this follows a syntax similar to that in the proposed occam3 [Bar92] language[3]. For example, the corresponding client process for the above 'encode.server' might be:

```
PROC encode.client (SHARED ENCODE! link)
  MOBILE []BYTE v:
  SEQ
    ...  allocate and initialise 'v'
    CLAIM link
      SEQ
        link[raw] ! v
        link[cooked] ? v
    ...  use modified 'v'
:
```

Even though the programmer is required to CLAIM a shared channel-bundle end before using the channels within, it is still his/her responsibility to ensure compatibility with the server (by communicating on 'raw' then 'cooked' sequentially), and additionally to ensure that competing clients do not interleave in destructive ways. The following is a broken example, in which multiple clients could interleave, resulting in deadlock:

```
PROC broken.encode.client (SHARED ENCODE! link)
  MOBILE []BYTE v:
  SEQ
    ...  allocate and initialise 'v'
    CLAIM link
      link[raw] ! v
    CLAIM link
      link[cooked] ? v
    ...  use modified 'v'
:
```

A partial solution to this, that enforces certain behaviours on processes using channel-bundle ends, is examined in section 4.7.

Thus far, the mobile nature of channel bundle ends has not been directly exploited[4]. Figure 4.7 on page 79 shows a server and clients that use a non-shared 'ENCODE' channel-bundle. Access to the server is managed by the clients themselves, by communication of the channel-bundle end linked to the server process. Such a client might be implemented with:

```
PROC uni.encode.client (CHAN ENCODE! in?, out!)
  ENCODE! link:
  WHILE TRUE
    MOBILE []BYTE v:
    SEQ
      ...  allocate and initialise 'v'
      in ? link
      link[raw] ! v
      link[cooked] ? v
      out ! link
      ...  use modified 'v'
:
```

[3]The occam3 language used a claim/grant mechanism to handle client/server sharing of *call-channels*.

[4]Mobile parameter passing, however, is a semi-mobile operation — their references are copied into the PROC call, then back again afterwards (for PROC calls that assign or communicate their mobile parameters).

This example is slightly artificial. The more usual way to achieve this functionality (clients using the services offered through 'ENCODE') would be to use a shared client channel-end, as described above.

For a more complicated system, it might be the case that a client process requests a client-end from a central server process, that manages a number of different 'ENCODE' servers. An example network of this type is shown in figure 4.9, where a 'client' process might be implemented as:

```
PROC managed.encode.client (CHAN INT request!, CHAN SHARED ENCODE! in?)
  SHARED ENCODE! link:
  MOBILE []BYTE v:
  SEQ
    request ! 0
    in ? link

    ...  allocate and initialise 'v'
    CLAIM link
      SEQ
        link[raw] ! v
        link[cooked] ? v
    ...  use modified 'v'
:
```

The 'server manager' process simply ALTs for requests from the clients, responding with the desired client channel-end. The main loop of this server-manager might be:

```
PROC server.manager (SHARED ENCODE! server.0, server.1,
                     []CHAN INT request?, []CHAN SHARED ENCODE! response!)
  ...
  WHILE TRUE
    ALT i = 0 FOR SIZE request?
      INT n:
      request[i] ? n
        CASE n
          0
            response[i] ! server.0
          1
            response[i] ! server.1
:
```

Cloning Shared Channel-Ends

For SHARED channel-ends, an *alias* is created through the use of the CLONE operator. Attempts to CLONE a non-shared channel-end are rejected by the compiler. The use of the 'CLONE' keyword is partly recycling, but fully reasonable. When applied to mobile data items, CLONE causes copy-semantics, rather than a movement semantics. The meaning of "a copy of a channel-end" might seem ambiguous, but in some respects it is not, "a copy of a channel-end that is connected to the same process(es) as the existing channel-end". The 'SHARED' nature of the channel-end allows it to be duplicated, since access is controlled through 'CLAIM' blocks. Attempts to CLONE a non-shared channel-end would clearly create aliases, to which access cannot be controlled — and is therefore disallowed by the compiler.

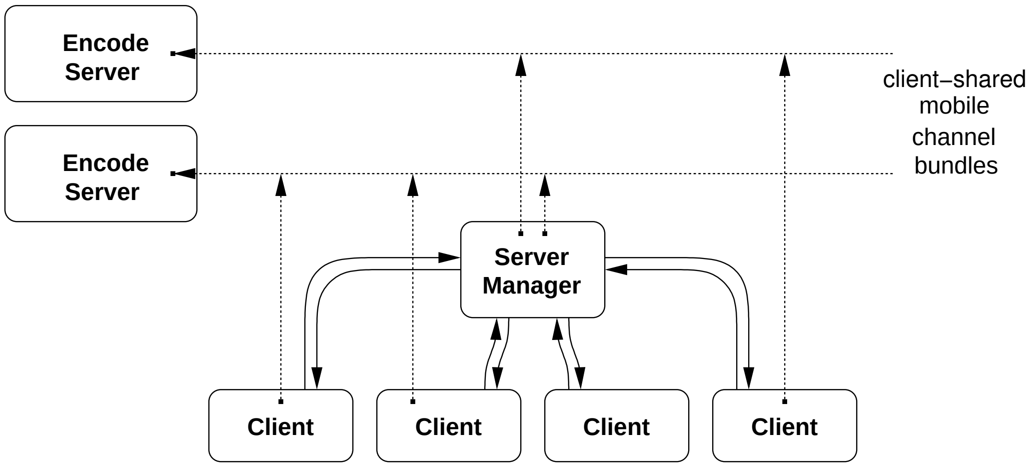


Figure 4.9: Example 'encode' servers, clients and a manager process, using shared channel-ends

In general usage, typically for setting up shared client/server relationships, the high-usage of the '`CLONE`' operator can obstruct the simplicity of code. As such, '`CLONE`' is always inserted by the compiler when `SHARED` channel-ends are communicated or assigned — as opposed to losing the local reference (from the right-hand-side of assignment, or output variable in communication). The above ('`server.manager`') code demonstrates this by omitting the `CLONE` when returning a channel-end to a client. The impact of this is very minor — a slight increase in overhead for cases where '`CLONE`' would not have been used originally.

The semantic issues of leaving a *shared* mobile channel-end variable defined after output or assignment turn out to be largely uninteresting. The only things that can happen (directly) to a mobile channel-end variable are: it is output; it is assigned from; it is over-written on input; or it is over-written on assignment (including allocation). For the last two of these, the variable being inputted or assigned to is conditionally "freed" before the action (input or assignment). That involves checking for definedness (whether it is `NULL` or not), and if valid, decrementing the reference-count, freeing if it reaches zero. This happens in the same way as when a mobile channel-end variable is descoped (section 4.3.1). Thus, before input or assignment, the definedness state of the variable is irrelevant. For the other cases of output and assignment from, attempts to use an *undefined* variable will result in compiler warnings and (trapped) run-time errors. The automatic `CLONE`-ing may hide some of these (program logic) errors as far as compiler and run-time environment are concerned. However, such errors *should* be apparent when testing (depending on the extent of the testing and the particular nature of the error).

One issue that does arise from the automatic `CLONE` for `SHARED` mobile channel-ends, on output and assignment from, is how to forcefully undefine one — i.e. really lose the reference. A somewhat special mechanism, integrated into the undefinedness checker (section 4.6), is provided for this. Assuming a mobile channel-end variable '`link`' (either shared or unshared) is in scope, a program may make the statement:

```
#PRAGMA UNDEFINED link
```

The normal effect of this construct is to inform (force upon) the undefinedness checker, information about the state of a particular variable. For dynamic `MOBILE`s (so far, dynamic mobile arrays and mobile channel-types), this statement causes the compiler to ensure that the mobile is undefined, both in its internal representation of the variable's state, and at run-time by ensuring an invalid (NULL) pointer is left in the relevant workspace slot (performing a conditional free if necessary).

4.3.3 Semaphore Support for Shared Channels

The semaphore support required to handle the CLAIMing of shared channel-ends consists of three operations: initialise, claim, and release. In the initial implementation, these operations were handled entirely by the compiler, resulting in relatively large amounts of ETC output. Recent modifications to the implementation has changed this — semaphore support is now provided through three ETC specials: 'SEMINIT', 'SEMCLAIM' and 'SEMRELEASE'. This is generally a better approach, since it allows these operations to be implemented in an architecture-specific way (that might involve SMP considerations).

tranx86 handles all of these internally, primarily for efficiency in the resulting code. For other architectures, support for this might be better placed in the run-time kernel, with the appropriate calls generated by the translator.

Initialisation, as already mentioned, is trivial. The '*Fptr*' and '*Bptr*' fields are set to 1 and 0 (NotProcess.p) respectively. Ensuring that '*Bptr*' is null when the queue is empty allows for some extra optimisation. The implementations of 'SEMCLAIM' and 'SEMRELEASE' are optimised for the best-cases, which are that the semaphore is free ('*Fptr*' is 1) for 'SEMCLAIM', and that no processes are waiting ('*Fptr*' is NotProcess.p) for 'SEMRELEASE'. A simple benchmark program was constructed to measure the specific overhead of this, consisting largely of the following loop:

```
SEQ i = 0 FOR 1000000
  CLAIM err!
    SKIP
```

The overhead is calculated by measuring the time required for this loop, and subtracting the time for a similar, but SKIP-only loop. In the best case, where no run-time check is needed to validate 'err!', the fast-paths for claim and release total around 9 ns. With a run-time check for the validity of 'err!', the cost is increased to around 12 ns. In this example, 'err!' is a *anonymous* channel-type, as covered in the following section. These have the advantage that they are always defined, and as such need no run-time checks.

A slightly more realistic benchmark is one where multiple processes compete for access to a semaphore. This (in its worst possible case), is implemented using the following:

```
PAR
  SEQ i = 0 FOR 1000000
    SEQ
      CLAIM err!
        RESCHEDULE ()
      RESCHEDULE ()
  SEQ i = 0 FOR 1000000
    SEQ
      CLAIM err!
        RESCHEDULE ()
      RESCHEDULE ()
```

When this code starts up, one of the parallel processes will claim the semaphore and reschedule. The other process then starts up, attempts to claim the semaphore and fails. This process then blocks on the semaphore and reschedules the first process. The first process the exits its CLAIM block, leaving the semaphore claimed and putting the blocked second process on the run-queue. The first process then enters its second 'RESCHEDULE', putting itself on the run queue and scheduling the second process.

The second process then wakes up, immediately inside the 'CLAIM', and reschedules. The sequence then continues as before, but in reverse, with the first process blocking on the CLAIM and the second process rescheduling it as it exits its own CLAIM. In total, each interleaved iteration of the two processes consists of: 2 semaphore claims that block on the semaphore and reschedule; 2 semaphore releases that move a (single) blocked process from the semaphore queue to the run-queue; and 4 explicit 'RESCHEDULE's.

Minus the overhead for the loops, the cost of of these operations is approximately 252 ns. With (scheduler) inlining enabled, this is reduced to about 174 ns. With the overhead for the four explicit reschedules removed, the remaining semaphore operations (and associated scheduling) cost 205 ns, reduced to 136 ns with inlining. These times were measured on an 800 MHz P3.

Rescheduling

Much of the low-overhead here can be attributed to the (new) implementation of 'RESCHEDULE'. The code (generated by the translator from the '.RESCHEDULE' ETC special, described in Appendix B.2.4), only enters the run-time kernel if a synchronisation-flag has been set, or if there is timer activity and timer inlining is not enabled. Otherwise, if the run-queue is non-empty, the current processes is added to the back and a new one scheduled.

The best-case overhead (where the current is the only runnable process) is around 5 ns (a few comparisons and conditional jumps). Two processes that schedule each other each incur an average overhead of 9 ns each for a reschedule. Where three processes are involved, each incurs an average overhead of 7 ns.

4.4 Anonymous and Recursive Channel Types

Parts of the network in figure 4.9, particularly the request channels from the 'client' processes to the 'server-manager' process, are over-complicated. Ideally, a *shared* channel should be used — with the client processes sending their identity as a request, which the manager process uses in order to respond to the correct client.

An alternative (and perhaps almost obvious) version of the network is one in which the clients and server are connected by a single (client-shared) channel-bundle, containing the necessary request and response channels. For example:

```
CHAN TYPE SVR.MGR.IF
  MOBILE RECORD
    CHAN INT request?:
    CHAN SHARED ENCODE! response!:
:

SVR.MGR.IF? smgr.svr:
SHARED SVR.MGR.IF! smgr.cli:
SEQ
  smgr.svr, smgr.cli := MOBILE SVR.MGR.IF
  ...  parallel processes using "smgr.svr" and "smgr.cli"
```

However, this approach tends to assume that the 'server-manager' process will respond immediately when a request is accepted. At the very least, programmers may be tempted to write:

```
PROC managed.encode.client2 (SHARED SVR.MGR.IF! to.manager)
  SHARED ENCODE! link:
  MOBILE []BYTE v:
  SEQ
    -- get "link" (to encode server 1)
    CLAIM to.manager
      SEQ
        to.manager[request] ! 1
        to.manager[response] ? link

    ...  code using "link" and "v"
:
```

It may well be the case that the 'server-manager' process does not respond immediately — i.e. it may be implemented to control the number of clients simultaneously accessing a server, and therefore may need to "hold-up" a client making a request. This is not an unreasonable scenario; a 'high-security' `ENCODE` server may exist, that demands to be consulted before allowing connections (to verify a client, for example). It is the responsibility of 'server-manager' to ensure this, that can be explicitly programmed to prevent access by multiple clients.

There are two particular problems that must be solved in order to program this functionality usefully. The first is allowing clients to be blocked mid-transaction, without affecting the servicing of other clients. The second is how a client reports, to the manager process, that is has finished using a particular server — and perhaps to enforce this behaviour.

In order to correctly support separate processing of multiple clients, the network must be wired with individual channel from the 'server-manager' to each 'client'. It is not sufficient to simply modify the above `PROC` body to:

```
SEQ
  -- get "link" (to encode server 1)
  CLAIM to.manager
    to.manager[request] ! 1
  CLAIM to.manager
    to.manager[response] ? link
  ...
```

This will have highly unpredictable and definitely undesirably results when clients interleave — since there is no way for ordering of clients on the second '`CLAIM`' to be controlled[5]. Specific programming could be inserted between the two `CLAIM` blocks (to force deterministic arrival at the second `CLAIM`), but this is not generally practical.

The way to construct a correct version of such a network would be as already mentioned — a shared channel for requests, and individual (per-client) channels for results. Setting up this shared channel would require constructing a channel-type with just one field, then allocating that channel before distributing to the processes concerned. This is slightly cumbersome, since all that is desired

[5]Technically there is a way to *almost* guarantee ordering on `CLAIM`s, using process priority. If each client executes at a separate priority level, the ordering of the arrival at a `CLAIM` can be determined from its immediately preceding interaction (with the server process). However, such behaviour is not guaranteed, and therefore should never be used. The implementation of priority presented in this thesis (section 7.2) will, however, allow this to be programmed and have the desired result.

is a single, simple, shared channel. For this specific purpose, the *anonymous* channel-type has been introduced, described in section 4.4.1.

The other problem to be solved is how a client process communicates (to a server) that it has finished with a channel-end. By default, when a channel-end goes out of scope, or before it is over-written by assignment or input, the reference count is decremented and the memory freed if it reaches zero (although this will not happen in the client if the server always retains a reference to the channel-bundle — its local 'end'). The server process to which clients are connected has no direct way of knowing when a client 'lets go' a channel-end in this way. The most obvious engineering would be to provide a separate channel, or element of a channel-type, to transport used client-ends from the 'clients' back to the 'server-manager'. The use of an anonymous channel-type (shared channel) might be a good candidate in many cases, but requires explicit wiring into the network.

Rather than adding extra channels to a process network, a specific mechanism has been introduced that allows a channel-type to communicate an end of *itself*, known as a *recursive* channel-type. Section 4.4.3 describes the details of this mechanism. A recursive channel-type is ideal when client processes wish to inform a server (to which they are connected) that it has finished using that server resource. For non-shared channel-ends, this will guarantee that the client will not use that channel-end again — it cannot, simply for the reason that it has communicated its own reference to a server, back to that server.

4.4.1 Anonymous Channel Types

Figure 4.10 shows the wiring for a more useful 'encode' network — one that can support complex management of its clients, as described above. The channels connecting the server-manager to the various clients are still required, since the server-manager must be able to communicate with each client individually. For a *static process* layout, but where connectivity may be dynamic, such as shown in figure 4.10, fixed wiring like this is unlikely to cause a problem — an array of (output) channels can be given as a parameter to the server-manager process, with the opposite (input) ends distributed to the clients.

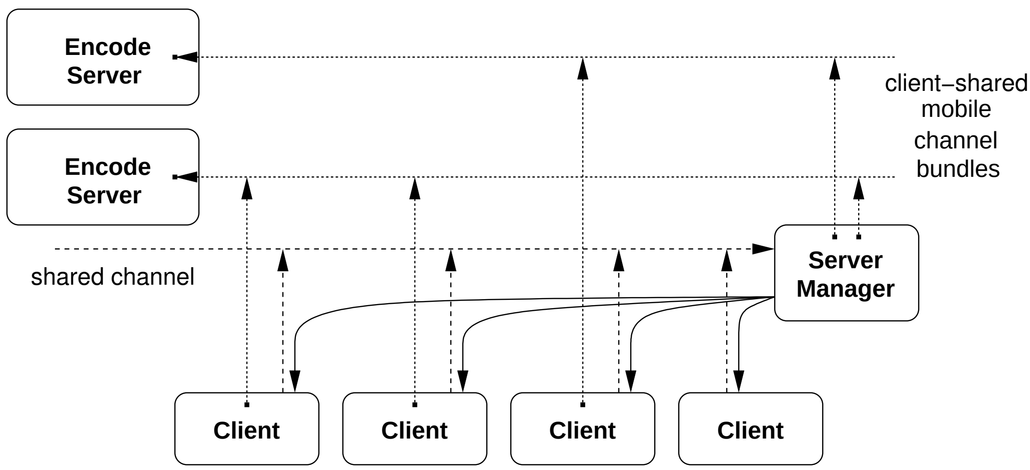


Figure 4.10: Modified 'encode' network, incorporating a shared channel

The single shared channel between the 'client's and 'server-manager' is of an anonymous channel-type. Semantically, these provide a shared channel of any protocol type — simple, sequential, or variant (CASE). The implementation for these uses *mobile* channel-types, but hides that aspect from the programmer — hence its anonymity.

A shared channel, of INTs for example, can be declared simply with:

```
SHARED CHAN INT c:
```

This simple declaration is re-written by the compiler to produce declarations and initialisation code for the channel-ends that are used to provide this shared channel. The above declaration creates an *any-to-any* channel, i.e. both ends are shared. The other two shared combinations, *any-to-one* and *one-to-any*, are declared by adding a direction-specifier to the 'SHARED' keyword. For example:

```
SHARED! CHAN INT::[]BYTE messages:       -- any-to-one (multiple outputters)
SHARED? CHAN MOBILE []BYTE to.servers:   -- one-to-any (multiple inputters)
```

In all three examples, one name is used to declare and initialise two channel-ends (of opposing direction). Consequently, whenever the name of a shared-channel is referred to, the end in question must be explicitly indicated[6](using a direction specifier). The handling of this, and re-writing of declarations, are covered in section 4.4.2. There is no way to refer to both ends simultaneously — and no reason to either.

The usage of a shared channel *end* follows that of its corresponding shared channel-type end — it must be CLAIMed before use, for either input or output. For example:

```
SHARED! CHAN INT c:
PAR
  PAR i = 0 FOR 10
    CLAIM c!
      c ! i + 4
  SEQ j = 0 FOR 10
    INT v:
    c ? v
```

The input ('?') end is treated as an ordinary channel — it is not shared, and has a type of 'CHAN INT'. The direction-specifier on the declaration (that creates an *any-to-one* channel here), applies to declarations only. It does not need to be provided anywhere else, since subsequent usage of the channel refers to each end individually.

Shared-channel parameters and abbreviations follow in a similar way, but the direction must be specified on the new name. For example:

```
PROC s.out.string (VAL []BYTE str, SHARED CHAN BYTE out!)
  CLAIM out!
    SEQ i = 0 FOR SIZE str
      out ! str[i]
:
```

and:

```
SHARED CHAN INT in? IS other?:
```

[6]The compiler could work the end in question from the usage, but this would complicate the compiler somewhat, since it performs the scoping and semantic checking before analysing the usage of channels.

Mobile Shared Channels

Currently, there is no provision for the mobility of shared channels. That is, they may not be communicated or assigned, and channels of their type may not be declared. For example:

```
CHAN SHARED! CHAN INT c:
SHARED CHAN INT x, y:
PAR
  c ! x!
  c ? y!
```

is currently illegal. However, because of the way in which shared channels are implemented (using anonymous channel-types, described in the following section), such code *could* be supported. There are good reasons for not doing this however. Most importantly because it can destroy the logic of a shared channel — by leaving '`y!`' and '`y?`' unconnected ('`y!`' has been over-written). In addition to this, once a shared-channel end has been lost, there is no simple way to re-create the channel (except by reassignment of both ends from a fresh declaration).

The one exception to non-mobility of shared channels is when they are passed as parameters to `FORK`ed (dynamically created) processes, covered in section 5.3. However, the checks for this particular usage of shared channels ensure that the channel will remain in scope for the lifetime of any dynamically spawned processes. This allows a different implementation of shared-channels to be used at a later point, if desired, without having to impose any further restrictions.

4.4.2 Implementing Anonymous Channel Types

Shared channels are implemented using (dynamic) anonymous channel-types. The tree transformations, that turns an original declaration or parameter/abbreviation into the corresponding mobile channel(s), are performed in the type-checking phase of the compiler — in the compiler front-end. For example, the declaration (and process):

```
SHARED CHAN INT x:
...  process using "x!" and "x?"
```

is re-written to produce (the illegal):

```
CHAN TYPE $anon.INT
  MOBILE RECORD
    CHAN INT c?:
:

SHARED $anon.INT! x$cli:
SHARED $anon.INT? x$svr:
SEQ
  x$cli, x$svr := MOBILE $anon.INT
  ...  process using "x!" and "x?"
```

As the name scoping and type-checking progresses, references to '`x!`' and '`x?`' are linked to either the appropriate mobile channel-end ('`x$cli`' or '`x$svr`'), or the subscripted mobile channel-end — to access the actual channel ('`x$cli[c]`' or '`x$svr[c]`').

For anonymous channel-types that are shared at one end only, the channel '`c`' within the non-shared end is renamed. For example, to either '`x$cli`' or '`x$svr`' for '`SHARED? CHAN INT x`' and '`SHARED! CHAN INT x`' declarations respectively. For example:

```
SHARED! CHAN BYTE c:
PAR
  s.out.string (c!, 0, "hello")

  CLAIM c!
    out.string (c!, 0, "world")

  WHILE TRUE
    BYTE b:
    SEQ
      c ? b
      ...  process "b"
```

is transformed into (the illegal):

```
CHAN TYPE $anon.BYTE
  MOBILE RECORD
    CHAN BYTE c?:
:

SHARED $anon.BYTE! c$cli:
$anon.BYTE? c$svr:
SEQ
  c$cli, c$svr := MOBILE $anon.BYTE
  CHAN BYTE c$svr IS c$svr[c]:        -- rename c$svr
  PAR
    s.out.string (c$cli, 0, "hello")

    CLAIM c$cli
      out.string (c$cli[c], 0, "world")

    WHILE TRUE
      BYTE b:
      SEQ
        c$svr ? b
        ...  process "b"
```

Such transformation is non-trivial in the compiler, since the exact transformation depends on what context the channel is referred to in — i.e. whether it is used as a `CHAN` parameter, a `SHARED CHAN` parameter, or for input or output. This is partly why a direction specifier must *always* be used, even when the end required can be deduced (but often not until later on in compilation, by the usage checker).

Anonymous Type Naming

The anonymous types created by the compiler to support shared channels are managed separately, such that the channel-type declarations themselves are never placed in the parse tree directly. This is needed to ensure the global scoping of these types, since two '`SHARED CHAN P.TYPE`'s must always be compatible, providing that '`P.TYPE`' is the same.

The names generated are largely uninteresting — being formed from the protocol alone, prefixed with '`$anon.`'. The range of available protocols is limited to a certain extent, declarations of the form 'CHAN INT; INT x:' are illegal, although it is obvious what is meant. Where PROTOCOLs or data-types are used, the corresponding type name is used.

More complex types, such as mobiles and arrays (both fixed-size and counted), have extra parts inserted into the generated type name. For example, a 'SHARED CHAN MOBILE INT' generates a mobile channel-type named '$anon.MOBILE.INT'. For fixed-size and counted-arrays, the string prefixes 'AA.' and 'CC.' are used instead of 'MOBILE.'. The mapping of protocols onto strings is only ever one-way. The compiler will never attempt to re-build a type from an '$anon.' name, and any attempt to do so would be largely unsuccessful, since some information is lost in the name. More to the point however, the compiler never needs to do this anyway — the actual protocol used can be found by simply examining the (single) channel declaration inside the generated type.

4.4.3 Recursive Channel Types

Recursive channel-types provide the ability to define channel-types that contain a channel carrying ends of that same channel-type. Because of the way in which occam declarations are processed (the *name* of a declaration comes into scope at the colon finishing that declaration), special provision must be made for early scoping of the name, such that it can be used within its own definition. This is done using the 'RECURSIVE' keyword (that may be shortened to 'REC').

For example, a version of the earlier 'ENCODE' channel-type, modified to allow clients to return an *unshared* channel-end, could be:

```
RECURSIVE CHAN TYPE ENCODE.R
  MOBILE RECORD
    CHAN MOBILE []BYTE raw?:
    CHAN MOBILE []BYTE cooked!:
    CHAN ENCODE.R! return?:
:
```

This adds a 'return' channel to the bundle, whose protocol is of 'ENCODE.R' client-ends (indicated by '!'), which are input by a server. The intended use for this it to allow a client to 'return' a channel-end to the connected server (on the other side of that channel-end). For example:

```
ENCODE.R! secure:
SEQ
  request ! 3
  response ? secure
  ...  use channels within "secure"
  secure[return] ! secure
  --  "secure" now undefined/invalid
```

After the corresponding server process performs the input on the 'return' channel, it will hold both ends of the channel-bundle, with at least the guarantee that it holds the *only* (unshared) client-end. Figure 4.11 shows this, omitting the 'server-manager' process and associated wiring for 'request' and 'response'.

Processes are not restricted to sending their own ends through recursive channel-types (as the above occam fragment does), although this was the originally intended use. Another application might be where a 'client' process has the ends of two servers, and communicates one of those ends to the other server, which could then be a client to another server, through the communicated end. There is the potential for deadlock here, through the creation of cyclic client-server networks (that may involve just one process). Avoiding deadlock is the responsibility of the programmer, and can

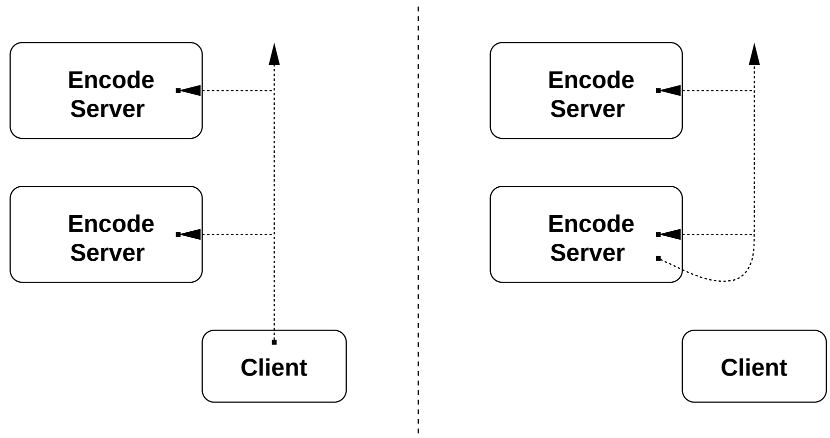


Figure 4.11: Recursive channel-type communication

be achieved through careful design. Locke, in [Loc01], also notes the dangers that arise from allowing the arbitrary re-connection of process networks.

4.5 Mobile Process Types

Mobile processes are the next logical step, having already 'mobilised' data and synchronisation-primitives (currently channel-bundle ends only). Mobile processes provide an implementation for 'agents' in occam. Broadly speaking, 'agents' are independent mobile processes, in the sense that they can be moved between distinct computing devices. For example, a process that moves between a user's hand-held PDA and a desktop PC, that might be used to retrive or store calender information, or even a process that is itself the calender — and simply *moves* to whatever device the user is using.

occam, and CSP technologies in general, provide an ideal framework for building such agents. Critically, occam processes cannot intefere with each other, in any way. If an agent chooses not to interact with its surrounding environment, it is entirely free to do so. The key support required for implementing agents, that occam has previously lacked, is that of process mobility.

To be useful, most agents require state. The state of an occam process (or more importantly, process network) is comprised of two things: any data local to processes; and the run-time state of those processes (for example, whether a process is on the timer-queue, or blocked in a channel). Therefore, in order to move a process, this state must be extracted from an existing network, communicated, then used to construct a new instance of the process or process-network.

Locating and extracting any *data* used by a process is trivial, as is the saving and restoring of that data. However, saving and restoring the run-time state of a process, is hard. To avoid the complication of accessing the active process state, it is easiest to disregard it altogether, and to impose a restriction on process mobility: a process may only be moved when it is not active.

Traditional occam cannot support this exactly — if a process is utterly inactive, its only state can be global data, and this is *not* permitted. Two possibilies for mobile process implementations exist. Firstly, to handle the complex state save and restore operations, allowing fairly arbitrary occam processes to be suspended, possibly migrated, and resumed. Or secondly, to provide a syntax that combines some data (state) with code, but that keeps the two distinct. This work in this chapter is concerned with the latter approach — the introduction of specific language features to provide this support. The first approach is examined in greater detail in section 6.5, but is rather more complicated in the implementation.

Section 4.5.1 covers the concept of 'mobile agents' in greater detail, including other support for these. The general mechanism added to occam, that supports the implementation of agents, is the *process-type*. Section 4.5.2 describes the declaration of process-types and the definition of processes that *implement* those process-types. Section 4.5.3 discusses the usage of mobile process-types, in particular, to construct mobile processes. The implementation of mobile process types is currently incomplete, but is underway. Section 8.3.3 discusses parts of the implementation that are in place, and what needs to be done to complete this support.

4.5.1 Mobile Agents

The concept of mobile agents, in particular the more general concept of mobile process, is not a new idea. The π-calculus [MPW92], for instance, provides a calculus for mobile processes, where process mobility is a fundamental operation.

The most obvious language for the implementation of mobile agents is Java. Usefully, Java provides mechanisms for the *serialisation* of objects — i.e. the transformation of a Java object (or arbitrary object graph) into some form of byte-stream, suitable for communication between Java virtual-machines (over a network, for example).

Despite the provision of a good mechanism for transferring objects, Java lacks some key features that are particularly desirable when constructing mobile agents. The foremost of these is the total lack of self-control that Java objects exhibit. Unlike an occam process, a Java object is unable to control what happens to it. Other objects may contain references to the "agent object", and are un-restricted in their invocation of public methods. Contrast with an occam process, that must explicitly engage in communication on order to interact with its environment. Furthermore, the Java '`Thread`' object introduces unprecedented possibility for aliasing and race-hazard errors — multiple threads might interact (call methods) on each other, with failures ranging from subtle to catastrophic. Some control is possible by enforcing serialised access to methods, but in many cases this may not be sufficient, or may lead to unexpected deadlocks. These issues have been explored in detail by Locke in [Loc01], that provides an interesting insight into the problems inherent in some object-oriented environments.

The 'agent' paradigm is successfully captured by other programming environments however. Icarus [MM98b], for example, provides an '`on`' statement, that implements process migration.

Instead of having agents as an explicit entities within a system, an alternative approach is to abstract the agent into a simple global state, where the 'agent' becomes the access rights to that state. This modified idea of an 'agent', shown in figure 4.12, is easily implemented, particularly when given specific support for distributed global data, as provided by Vinter's PastSet [Vin99, PV02] for example. Traditional parallel processing infrastructures also provide this type of support, for example LAM-MPI [SLG+00], PVM [Sun90] and *Tuple Spaces* (for the 'Linda' language) [Gel85], to name a few.

However, extensive application support is often required to use these, and that may be unsuitable for smaller computing devices. This is not entirely unexpected, since these technologies are primarily designed to support large-scale collaboration between nodes in distributed computing systems (typically clusters), rather than for implementing mobile agents.

Given that traditional occam lacks any reasonable support for process mobility, its addition here has the opportunity to greatly enhance the usefulness of occam, for building mobile agent systems

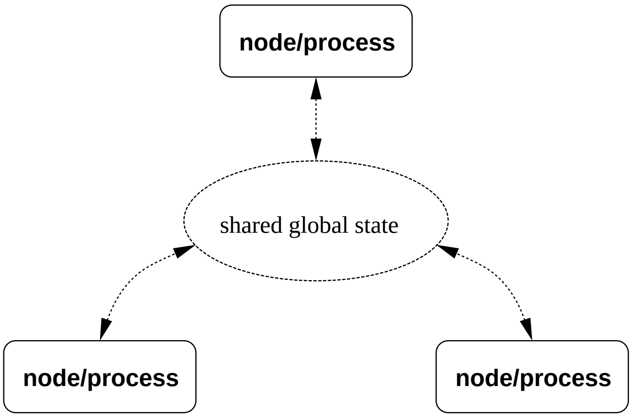


Figure 4.12: Global-state based mobile agent

at least. Additionally, occam exhibits exactly the sort of features that mobile agents require: self-contained, fault-tolerant processes that *communicate* with the (local) environment through well-defined interfaces. Figure 4.13 shows the intended run-time structure for an occam mobile process, that could be a mobile agent.

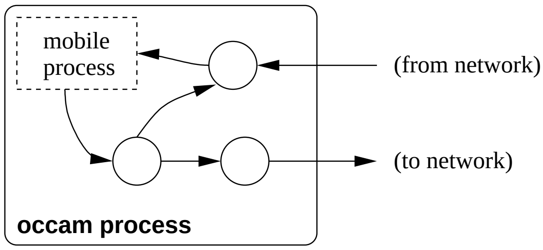


Figure 4.13: An example mobile process and local connections

However, in adding this support to occam, care must be taken to ensure that no data (or process) aliasing or race-hazard errors are introduced (largely through missed compiler checks).

4.5.2 Process Types for occam

The declaration of a *process type* in occam associates a name (the new type) with a particular `PROC` interface. This simply declares an abstract interface type, not any implementation. For example:

```
PROC TYPE CIPHER IS (MOBILE []BYTE key.in?, in?, out!):
```

This defines a `PROC` interface called '`CIPHER`', that might be suitable for an 'encryption' agent. Process types such as this are used in two places. Firstly, in the definition of *mobile processes* (below), and secondly in the declaration of *process variables* — that act as the 'handle' for a mobile process, covered in section 4.5.3.

Mobile Processes

Mobile processes are active entities that combine some data — their state — and code to initialise and manipulate that state at run-time. The state of a mobile process is defined using standard occam (data) declarations.

Initialisation is handled using PROC-style constructors, with arbitrary signatures. The code for mobile process *activation* is also implemented in a PROC-style manner, using earlier named process types to identify interfaces.

The following code shows a partial implementation of a *mobile process*, that offers an interface of the above 'CIPHER' type. A simple constructor-process is included, although the state here needs no explicit initialisation:

```
MOBILE PROC encrypt
  MOBILE []BYTE key:        -- private data

  CONSTRUCT ()
    SKIP
  :

  IMPLEMENTS CIPHER (CHAN MOBILE []BYTE key.in?, in?, out!)
    INITIAL BOOL running IS TRUE:
    WHILE running
      PRI ALT
        key.in ? key
          ...  new encryption key

        MOBILE []BYTE raw, enc:
        DEFINED key & in ? raw
          IF
            (SIZE raw) = 0
              running := FALSE
            TRUE
              SEQ
                ...  encrypt 'raw' using 'key' and place in 'enc'
                out ! enc
  :
:
```

This declares a mobile process called 'encrypt', whose state consists of the single variable 'key' (that happens to be mobile too). A single state-initialisation process is provided, using the 'CONSTRUCT' keyword. For this example, no initialisation is required — the 'key' variable, like all dynamic mobile arrays, is initialised when it comes into scope (or in this case, when the mobile process is created).

There is no limit to the number of 'CONSTRUCT' definitions that a mobile process may have — the above code could reasonably contain no CONSTRUCT processes. However, it might be convenient to provide a constructor that initialises the 'encrypt' process with a key. For example:

```
CONSTRUCT (MOBILE []BYTE key.in)
  SEQ
    CONSTRUCT ()            -- call the first constructor
    key := key.in
:
```

This also demonstrates that constructor processes may call other, previously defined, constructors. CONSTRUCT processes are identified by their PROC-style signatures, rather than by name. Multiple same-signature (or ambiguous) CONSTRUCTs are permitted, subject to occam's normal scoping rules. In cases of ambiguity, the compiler generates a warning (or error in strict mode), since this would normally be undesirable. For example:

```
CONSTRUCT (VAL INT16 i)
  ...
:

CONSTRUCT (VAL INT i)
  ...
:
```

The warning is only generated when the ambiguity occurs, since it is sometimes avoidable. If, after these definitions, a constructor is called with an INT expression as the argument, the only possibility is the second ('VAL INT') constructor. However, if a simple literal were used, either constructor could be used. In these (ambiguous) cases, the most recently defined constructor is used, as would be expected from occam's normal scoping rules.

Interface Implementations

The above 'encrypt' example contains a single 'main body' process that provides an implementation for the 'CIPHER' process-type. The PROC-style signature given with 'IMPLEMENTS' must exactly match that of the 'PROC TYPE', with the exception of parameter naming.

Following in a similar manner from constructors, mobile processes may have implementations for multiple process-types — possibly zero, although the usefulness of such a process would be severly limited. Implementations may invoke other, previously defined, implementations (within that mobile process). As before, strict occam scoping rules apply, preventing the creation of recursive call structures.

Unlike constructors, implementations are identified by the process-type which they implement, permitting multiple same PROC-style interfaces. As an example, consider the following two process type declarations:

```
PROC TYPE MATH.COMPONENT IS (CHAN REAL64 in?, out!):
PROC TYPE AVG.MATH.COMPONENT IS (CHAN REAL64 in?, out!):
```

These interfaces might be implemented by an 'integrator' component, in such a way that the 'AVG.MATH.COMPONENT' code uses the 'MATH.COMPONENT' implementation, in parallel with a suitable averaging buffer process. For example:

```
MOBILE PROC integrator
  REAL64 sum:

  CONSTRUCT ()
    sum := 0.0
  :

  IMPLEMENTS MATH.COMPONENT (CHAN REAL64 in?, out!)
    ...  integrator process (serial)
  :
```

```
    IMPLEMENTS AVG.MATH.COMPONENT (CHAN REAL64 in?, out!)
      CHAN REAL64 c:
      PAR
        MATH.COMPONENT (in?, c!)
        avg.buffer (c?, out!)
    :
  :
```

Although the '`integrator`' process defines implementations for two process-types, no relationship between those types is created — this is not object orientation. Figure 4.14 shows an approximate representation of this '`integrator`' component, clearing showing the two separate implementations. The similarities and differences between process-types and object-orientation are examined in section 8.1.2.

The two implementations provided by '`integrator`' are distinct, and can be reasoned about in the same way as any other occam process. The only difference is that they have additional state available to them — to which normal aliasing and parallel-usage rules apply.

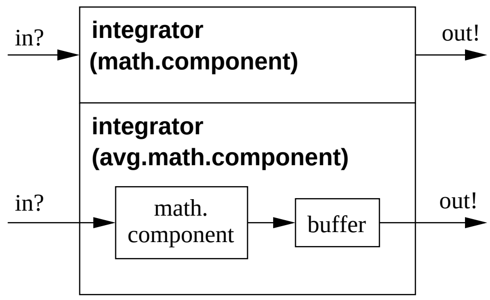


Figure 4.14: Example '`integrator`' mobile process implementations

State and Parameter Restrictions

To be able to reason about a mobile process as an independent agent (which is desirable), the mobile process *must not* contain any 'hidden' links to its environment. Currently, the only way this could be set up is by having a mobile channel-type variable as part of the process' persistent state. Therefore these are banned by the compiler. Other mobile types (data and processes) are permitted within the state of a mobile process, as are ordinary channels and anonymous channel types (shared or unshared). Channels within a mobile process' persistent state are only convenient — to save declaration inside multiple '`CONSTRUCT`' or '`IMPLEMENTS`' definitions. When a mobile process is *inactive* (the state where it can be communicated and assigned), any internal channels would be empty, and for anonymous channel-types, unclaimed.

The '`CONSTRUCT`' definitions within a mobile process are ultimately called from the construction of that particular mobile process, provided as a special form of assignment. Semantically, the mobile process constructor is an expression that generates a mobile process. Following the normal rules of occam, specifically that expressions must not be side-effecting, leads to parameter-type restrictions for '`CONSTRUCT`' definitions. Specifically, non-`VAL` parameters may not be used. `MOBILE` parameters are permitted however, with the exception of mobile channel-ends (that are banned).

Unlike FUNCTIONs ('VALOF' processes), CONSTRUCT definitions may use timers and general non-determinism (ALTs) freely, since these cannot have a direct (side) effect on the process invoking the constructor. The two mechanisms that could cause side-effects, priority and timeouts, are banned — and it is unclear why either of these would be required within a mobile process constructor anyway (given the parameter restrictions).

The formal parameters of an 'IMPLEMENTS' block can take any valid type, including its own. This must be specified in the original process-type declaration however, using the 'RECURSIVE' keyword. For example:

```
RECURSIVE PROC TYPE FOO IS (MOBILE FOO v, CHAN MOBILE FOO out!):
```

When any implementation of the above is activated, that activation will be distinct from the mobile process passed in 'v'. The compiler will ban any attempt to pass a (mobile process) variable as a parameter to its own activation. The use of 'CLONE' for such parameters is permitted however, since the clone will be created before the activation takes place. The behaviour of 'CLONE' when applied to mobile processes is discussed in the following section.

4.5.3 Using Mobile Processes

Mobile process variables (of a process type) provide the basic mechanism for handling mobile processes. After creation and assignment to a mobile process *variable*, a mobile process may either be *activated*, assigned, communicated or destroyed. The (nearly standard) parallel-usage rules applied to mobile process variables prevent concurrent operations, such as attempting to communicate an active process. With the exception of creation, all operations performed on a mobile process variable relate to its type only — not the particular mobile process ('MOBILE PROC') providing the implementation.

The creation of mobile processes follows a style similar to that of other dynamic mobile constructors — as a specialised assignment. For example, a mobile 'CIPHER' process can be created using the 'encrypt' implementation, with:

```
MOBILE CIPHER x:            -- mobile process variable
SEQ
  x := MOBILE encrypt ()    -- allocate and initialise mobile process
  ...  process using 'x'
```

As for other mobile variables, initially 'x' is undefined, and any attempt to activate, assign from, or communicate it will result in a compiler error (from the undefinedness checker, described in section 4.6). The creation of a mobile process is performed using a single assignment, as shown above.

Mobile process creation is the only point at which 'encrypt' is referred to explicitly. Once 'x' has been defined, it is simply a 'MOBILE CIPHER' — and could have any implementation.

Once initialised (defined, possibly using a constructor — as above), a mobile process, through its single variable reference, may either be communicated, assigned, or activated (using the interface defined by its type). The syntax for activating a process variable is the same as that used for instancing standard occam PROCs, except that the name being referred to is a mobile process variable, not a PROC definition.

For example, the "process using 'x'" above, could activate the 'CIPHER' implementation, provide an initial key, then shutdown the process for later use:

```
SEQ
  CHAN MOBILE []BYTE key, e.in, e.out:
  PAR
    x (key?, e.in?, e.out!)       -- mobile process activation
    SEQ
      key ! "SecretMessage"       -- load process with key
      e.in ! ""                   -- terminate mobile process
  ...  process using key-enabled ‘x’
```

Note that this code makes no reference to ‘`encrypt`’ (the mobile process that ‘`x`’ references) — that information is largely unimportant for this process, whose usage of ‘`x`’ is based on ‘`x`’s type alone (a ‘`MOBILE CIPHER`’ in this example).

For the purposes of (parallel) usage checking, activations of a mobile process are treated as *writes* to that mobile process variable. This prevents parallel use of a mobile process by *any* other process, regardless of whether such processes attempt to activate, read or assign the mobile process. Parallel activations are illegal since they can easily lead to race-hazards *inside* the mobile process.

Communicating Mobile Processes

Mobile processes are communicated in a similar way to other mobile variables — i.e. with a movement semantics. Once a mobile process (variable) has been output, it is lost. Subsequent attempts to access the mobile process will result in either compile-time or run-time errors. Similarly, when inputted, any previous reference to a mobile process is lost (replaced with the communicated process).

A mobile process may only be communicated when not active. This is largely enforced by the standard parallel and aliasing checks performed by the compiler, that would reject code such as (the illegal):

```
PAR
  p.out ! x
  x (...)
```

Subsituting ‘`SEQ`’ for ‘`PAR`’ here would also yield an illegal process, since the activation would be attempted on an *undefined* mobile process.

Figure 4.15 shows three processes that communicate mobile processes of a ‘`T.CIPHER`’ type, the ‘`T.`’ indicating that this interface handles termination somewhat more explicitly.

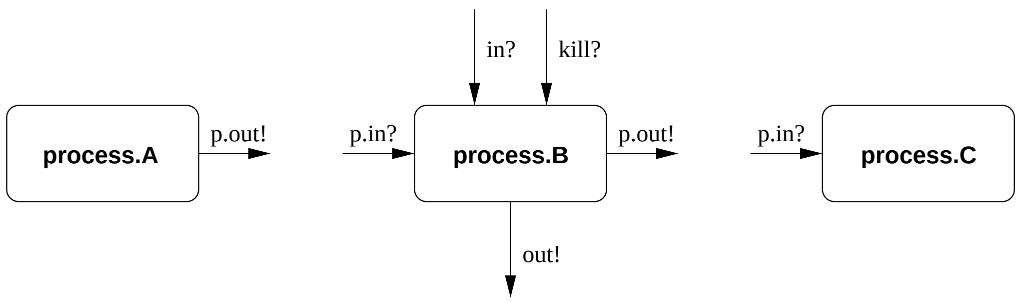


Figure 4.15: Mobile-process communicating processes

The process-type declaration for this modified type is simply:

```
PROC TYPE T.CIPHER IS (CHAN BOOL term?, CHAN MOBILE []BYTE key.in?, in?, out!):
```

A corresponding implementation within the '`encrypt`' process (from page 96) might be:

```
IMPLEMENTS T.CIPHER (CHAN BOOL term?, CHAN MOBILE []BYTE key.in?, in?, out!)
  CHAN MOBILE []BYTE c:
  PAR
    CIPHER (key.in?, c?, out!)
    SEQ
      INITIAL BOOL running IS TRUE:
      WHILE running
        PRI ALT
          term ? running
            running := FALSE
          MOBILE []BYTE v:
          in ?? v
            c ! v
      c ! ""                -- terminate CIPHER
:
```

This implementation re-uses the existing '`CIPHER`' implementation — simply by instantiating it. Parallel usage checking of such code is not as hard as it might at first appear — any accesses made on the mobile-process state by an implementation are fully determinable at compile-time. The code being instantiated is that of the literally preceeding '`CIPHER`' implementation.

For the three processes shown in figure 4.15, '`process.A`' creates a new mobile '`T.CIPHER`' (from the modified '`encrypt`'), initialises it and then outputs it. Assuming the existance of a suitable constructor, this might be implemented simply with:

```
PROC process.A (CHAN MOBILE T.CIPHER p.out!)
  MOBILE T.CIPHER x:
  SEQ
    x := MOBILE encrypt ("SomeSecretKey")
    p.out ! x
:
```

'`process.B`' inputs a previously initialised mobile `T.CIPHER`, then activates it, connected to its own local (parameter) channels. When the mobile process activation terminates, that mobile process is outputted. For example:

```
PROC process.B (CHAN MOBILE T.CIPHER p.in?, p.out!,
                CHAN BOOL kill?,
                CHAN MOBILE []BYTE in?, out!)
  MOBILE T.CIPHER x:
  SEQ
    p.in ? x
    CHAN MOBILE []BYTE temp:
    x (kill?, temp?, in?, out!)
    p.out ! x
:
```

The final process '`process.C`' simply acts as a sink for mobile `T.CIPHER`s. Since there is no explicit freeing required for dynamic mobiles, the process can be inputted and discarded (either by a descoping variable or by being over-written in input or assignment):

```
PROC process.C (CHAN MOBILE T.CIPHER p.in?)
  MOBILE T.CIPHER x:
  p.in ? x
:
```

Figure 4.16 shows the parallel composition of these three processes, forming a simple process network that communicates mobile '`T.CIPHER`'s. The code for this network is as expected (where '`kill?`', '`in?`' and '`out!`' are free):

```
CHAN MOBILE T.CIPHER c, d:
PAR
  process.A (c!)
  process.B (c?, d!, kill?, in?, out!)
  process.C (d?)
```

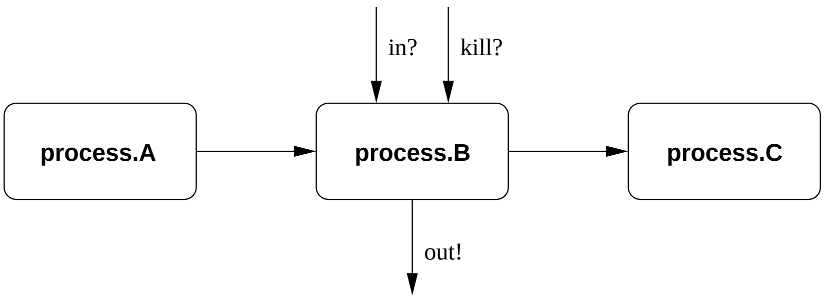


Figure 4.16: Mobile-process communicating process network

Although this example is relatively trivial, it demonstrates a powerful mechanism — that of mobile agents. Unlike some other languages that provide similar capabilities, occam 'agent' implementations are guaranteed to be free from alias and race-hazard errors; a necessary requirement for any software system.

Assigning Mobile Processes

The rules for assignment of mobile processes follow those for communication, based on the equivalence of communication and assignment. Assignment is somewhat simpler however — the mobile process *moves* from the source to the destination. For example:

```
MOBILE CIPHER x, y:
SEQ
  x := MOBILE encrypt ("SomeSecret")
  y := x
  y (key.in?, in?, out!)
```

After the assignment '`y := x`', the variable '`x`' is undefined. Any attempt to use it (other than as the target of assignment or input) will result in a compiler error (from the undefinedness checker, covered in section 4.6).

In a similar way to mobile data-types, the 'CLONE' operator may be used to produce a copy of a mobile process, replicating the state of that process exactly. This can be used in both assignment and communication, and leaves its operand defined. For example:

```
MOBILE CIPHER x, y:
SEQ
  x := MOBILE encrypt ("AnotherSecret")
  y := CLONE x

  CHAN MOBILE []BYTE c:
  CHAN MOBILE []BYTE tmp1?, tmp2?:
  PAR
    x (tmp1?, in?, c!)
    y (tmp2?, c?, out!)
```

This code implements a 'double encrypt' functionality. Similar code, using the hypothetical 'decrypt', would undo the encryption operation (as would be expected).

The implementation of CLONE for mobile processes is relatively simple, since it only involves copying of memory referenced through the original variable ('x' in the above example). The destination must first be allocated, of course, if not done so already. Since the operation of 'CLONE' is specific to the mobile process, and not its process-type reference, the functionality for CLONE is provided as part of the process itself.

For mobile processes that contain other mobiles as part of their state, the CLONE code must perform a *deep* copy, as it would for other nested mobiles.

Type Conversion of Mobile Processes

So far, a mobile process may support multiple implementations (of different 'PROC TYPE's), but once created that interface is fixed — for instance, a 'T.CIPHER' cannot be assigned to a 'CIPHER' — the two types are clearly incompatible. However, given a 'T.CIPHER' implementation of 'encrypt', it might be desirable to *convert* (change) this to the 'CIPHER' implementation (retaining the internal state). Logically, there is no reason why this cannot be done — the operation is effectively the creation of a new mobile process, whose state is initialised from the existing ('T.CIPHER') state. Of course, this requires the explicit usage of 'encrypt' in the conversion.

The proposed syntax for conversion of a mobile process is simply the passing of an existing mobile process into an 'implicit' CONSTRUCT — that behaves very much like CLONE:

```
MOBILE T.CIPHER x:
MOBILE CIPHER y:
SEQ
  x := MOBILE encrypt ()        -- create a T.CIPHER implementation of encrypt
  ...  process using T.CIPHER "x"
  y := MOBILE encrypt (x)       -- create a CIPHER implementation of encrypt from "x"
  ...  process using CIPHER "y" ("x" now invalid)
```

The implicit constructor(s) can easily be generated automatically by the compiler, but could also be programmed explicitly:

```
CONSTRUCT (MOBILE T.CIPHER other)
  ...  extract from "other" into local state
:
```

Supporting the above is certainly possible. `MOBILE`s are valid parameters to constructors, and those constructors may move mobiles into the internal state, but may not communicate them externally — channel parameters are not allowed for `CONSTRUCT` blocks.

However, the parts of '`encrypt`' presented so far do not allow the extraction of the internal "key". This would need to be added to support the above (explicitly programmed) conversion. The difference is that the explicitly programmed version above only has a '`T.CIPHER`' — it cannot know that it was '`encrypt`' that generated it. However, the compiler can, ensuring that the same mobile process ('`encrypt`') is used. Furthermore, the compiler-generated version will *undefine* the original process variable — it has moved, having been converted in the process. A user-programmed conversion would not undefine the original, unless it was *moved* into the new state. To prevent loss of the original process in compiler-generated conversion, '`CLONE`' can be used. For example:

```
MOBILE T.CIPHER x:
MOBILE CIPHER y:
SEQ
  y := MOBILE encrypt ()          -- create a CIPHER
  x := MOBILE encrypt (CLONE y)   -- create a T.CIPHER
  ...  process using CIPHER "y" and T.CIPHER "x"
```

It is somewhat debatable whether providing this support for "automatic" conversion between interfaces is a good idea. Programming it explicitly could prove tricky, however, with a reasonable potential for programmer error. As such, it is proposed that a compiler flag be explicitly used to enable this support. There are certainly cases where having this support is useful — so it is better to provide it than not.

Figure 4.17 shows an example 'conversion' component, that simply transforms '`CIPHER`'s on its input channel into '`T.CIPHER`'s on its output channel (possibly extended). Such components cannot be 'generic' however — any incoming '`CIPHER`' must be implemented by a process that supports the '`T.CIPHER`' interface too. The already implemented '`PROTOCOL.HASH`' compiler built-in can be used for this, although it is not entirely graceful[7]. For mobile process *variables* (including parameters and abbreviations), '`PROTOCOL.HASH`' will generate a hash-code that is unique to the underlying implementation. This can be used to select between known implementations, for example:

```
PROC CIPHER.TO.T.CIPHER (CHAN MOBILE CIPHER in?, CHAN MOBILE T.CIPHER out!)
  WHILE TRUE
    MOBILE CIPHER x:
    in ?? x
      MOBILE T.CIPHER y:
      SEQ
        CASE PROTOCOL.HASH (x)
          PROTOCOL.HASH(encrypt)
            y := MOBILE encrypt (x)         -- convert to T.CIPHER
        out ! y
:
```

[7]The '`PROTOCOL.HASH`' builtin is used mainly for user-defined channels (section 6.4) and the higher-level communication mechanisms (section 7.4). It produces a hash-code for channel protocols, but can be applied to any variable or type.

Figure 4.17: Mobile process interface conversion component

4.5.4 Extending Mobile Processes

One facilitity that might initially appear to be missing is that of code re-use. In object-oriented languages, such as Java and C++, code re-use is (primarily) handled through mechanisms of *inheritance*. Mobile processes provide no such mechanism for inheritance (and sub-typing of process types), but the same paradigm (that of extending a generic component) is easily supported.

For example, an application might wish to extend the (limited) '`encrypt`' mobile process to produce an enhanced component. For a language like Java, this would be done by *extending* the component, creating a relationship between the types in the process. The occam approach, using mobile processes, would take the form:

```
#USE "encrypt"              -- load names from encrypt.occ

-- encrypt2 pre-processes the data before encrypting
MOBILE PROC encrypt2
  MOBILE CIPHER e:          -- any implemented interface will do

  CONSTRUCT ()
    e := MOBILE encrypt ()
  :

  IMPLEMENTS CIPHER (CHAN MOBILE []BYTE key.in?, in?, out!)
    CHAN MOBILE []BYTE c:
    PAR
      e (key.in?, c?, out!)
      ...  local process, reading from "in?" and writing to "c!"
  :

  IMPLEMENTS T.CIPHER (CHAN BOOL term?, CHAN MOBILE []BYTE key.in?, in?, out!)
    MOBILE T.CIPHER x:
    SEQ
      x := MOBILE encrypt (e)

      CHAN MOBILE []BYTE c:
      CHAN BOOL lt:
      PAR
        f (lt?, key.in?, c?, out!)
        ...  local process, reading from "in?","term?" and writing to "c!","lt!"

      e := MOBILE encrypt (x)
  :
:
```

The conversion between the '`CIPHER`' and '`T.CIPHER`' interfaces is not entirely pleasant, but it is possible. For specific cases such as this, where one mobile process completely contains another, providing a simpler mechanism might prove favourable. Care would need to be taken to ensure that the internal process (‘e’ in the above example) was not replaced by another implementation of ‘CIPHER’ — that could happen through standard assignment or communication — although any such code will be largely explicit and contained within that mobile process implementation (‘encrypt2’).

There is, of course, the option of providing a wholly automatic mechanism for type conversion — as a type-cast, similar to C++ and Java. For example:

```
MOBILE CIPHER x:
MOBILE T.CIPHER e:
SEQ
  ...  code to create and initialise "x"
  e := MOBILE T.CIPHER x
  ...  code using T.CIPHER "e".  "x" now undefined
```

However, in order to support this, the run-time system must be able to determine what interfaces are supported by the mobile process in ‘x’. Ultimately, this requires that processes have a table (or other data structure), that can be searched to provide the information necessary for changing between types (within a particular mobile process implementation, ‘encrypt2’ for example).

4.6 Undefined Usage Checking

For all dynamic mobile types (dynamic mobile arrays, channel-types and process-types), there exists the possibility of invalid references. For instance, immediately after declaration, the workspace-slot for a channel-type ‘end’ will be ‘NotProcess.p’ (zero in KRoC/Linux). Attempts to use any such *undefined* variable would result in run-time null dereferences, if it were not for the compile-time and run-time checks described here.

Undefinedness is something the occam compiler has traditionally taken little interest in. In fact, it performs absolutely no analysis in this regard — the body of existing checks cover aliasing and parallel-usage. Undefinedness, in traditional occam, is left up to the programmer — undefined variables may be read freely, and will produce undefined results. Such results are unlikely to be desirable (potentially resulting in run-time errors, such as array-bounds violations and arithmetic overflows).

Undefined usage checking is a necessity for the various mobile types presented in this chapter — if serious run-time error is to be avoided. In theory, there are three approaches to this: compile-time undefined usage checking; run-time undefined usage checking; or a combination of the two. For existing checks (particularly aliasing), the compiler uses the combined approach: analyse as much as possible (with limits on complexity) at compile-time, and insert run-time checks into the generated code for cases that could not be checked at compile-time. It seems reasonable to follow this (combined) approach — compile-time only checks would ultimately restrict programmer freedom (the compiler would be required to reject “unknown” definedness states); and run-time only checks would likely result in excessive code generation (a check before each ‘read’ of a mobile).

The undefined usage checker has been added as a new stage in the (orginally Inmos) occam compiler, performed after the aliasing and parallel-usage checks. In addition to mobile variables, ordinary occam variables (and in some cases channels) are subject to undefinedness checks, adding a significantly useful feature to the KRoC occam system. Perhaps the most obvious benefit has been to students, who are now warned about their (often frequent) use of undefined variables[8].

[8] In this light, the lack of undefined usage checking does not help occam gain favour with students — who are used to compilers that report these errors (in particular Sun’s Java compiler, ‘javac’).

The following sections, 4.6.1 and 4.6.2, cover the ideas underlying the implementation, and the implementation itself. Section 4.6.3 describes specific handling for the nested structures of arrays and records, that may be partially defined.

4.6.1 Undefinedness

When a (non-mobile) variable is declared in occam, its contents remain undefined until it is used as a target for either input or assignment. Once defined, a variable stays defined, until its declaration descopes. This is not the case for mobile variables however — when communicated or assigned, the previously held data (reference) is lost. This is essentially the difference between copying and moving semantics for data (as shown in figure 4.1 on page 66).

Mobile variables, that are not `CLONE`d, are left undefined after mobile output and assignment from. For example:

```
INT x, y:
MOBILE INT a, b:
SEQ
  a := 42
  b := a
  x := 99
  y := a - x
```

The last line of this code will result in a *undefined* value being placed in '`y`', since the value of '`a`' is lost following '`b := a`' (these are both mobile). A more serious error is where an array is indexed using an undefined value, for example:

```
INT i, n:
[4]INT data:
SEQ
  ...  code that initialises 'data' (but not 'i')
  n := data[i]
```

For both examples, the result ultimately depends on what the 'undefined' values of '`a`' and '`i`' happen to be. For the current KRoC/Linux implementation, the entire (static) program workspace, vectorspace and mobilespace are initialised to '`MOSTNEG INT`', before the program starts proper. Thus setting the initial value of any (non-dynamic) variable such as '`i`' above. For both examples above, this initial value will result in run-time errors — an arithmetic overflow in the first example, and an array-bounds error in the second. Other processes might not be so lucky however, getting previously-used memory for an initial workspace.

Undefined Checking Algorithm

The undefined checking algorithm, at its outermost level, operates on individual `PROC` and `FUNCTION` definitions. Each `PROC` or `FUNCTION` is processed in depth-first order (following the source code), maintaining a *stack* of active variables as it goes. For each variable, an array of states is maintained, for tracking multiple states of the same variable where necessary. A global index points at the 'current' state for each variable, and is zero at the outermost level.

At the simplest level, a variable can be in one of three states: *defined*, *undefined*, or *unknown*. For less trivial variables, such as arrays and records, the definedness state may be significantly more complicated.

As the parse tree is walked, processes are examined and their current states updated if necessary — for instance, assignment to a variable will cause the *current* state of that variable to be marked as defined. When a branch in the code is encountered, such as for loops, `IF`s, `ALT`s and `PAR`s, the state *before* the branch is copied into a new state, the global state index updated, and the particular branch of code examined. Once all branches have been processed, and a set of resulting states collected, a 'merge' occurs that combines the new states back into the current state (and restores the global state index). The processing (and result) of the merge depends on the type of branching structure being examined.

Whenever a variable is accessed for reading, its definedness state is checked. If undefined or unknown, an error is generated, although it may not be reported immediately. This helps prevent multiple undefined warnings (or, in some cases, errors[9]) regarding the same variable.

The following section describes the current implementation in detail. Particular care is taken in the processing of arrays and records, in an attempt to analyse their individual element accesses. For mobile variables, the parse tree is modified slightly to identify cases where a mobile is definitely undefined. This helps limit the number of *check-and-free* sequences generated for some dynamic mobile operations (such as input and assignment).

The undefinedness checker, in addition to checking the defined states of variables, performs checks necessary for other language constructs. For example, that `SHARED` channels (both anonymous and of named channel-types) are `CLAIM`ed before use.

4.6.2 Implementation

The undefinedness checker maintains a stack of in-scope variables, defined by the '`udv_t`' (structured) type. Checking starts by scanning the parse tree for outer-level `PROC` definitions. When found, the formal parameters are added to the variable stack (internally called '`udv_vstack`'), before processing the body of the `PROC` (or `FUNCTION`). The processing of `PROC` and `FUNCTION` bodies skips any nested definitions (of `PROC`s and `FUNCTION`s) — these are checked when instanced, since they only have local scope.

Whenever a new *name* comes into scope, an entry is added to the variable stack. New names occur for variable declarations (of any type), channel declarations, abbreviations (all types) and formal parameters (when processing nested calls). The '`udv_t`' type is defined as:

```
typedef struct TAG_udv_t {
    struct TAG_udv_t *next;           // next in in-scope stack
    treenode *nameof;                 // namenode link for variable

    char state[MAX_NESTING];          // state array: UDV_.. constants
    treenode *depend;                 // dependency if known (for UDV_UNKNOWN)
    struct TAG_udv_t **nested;        // nested state (of record fields, arrays, ...)

    int did_warn;                     // non-zero if warnings generated
} udv_t;
```

To illustrate how the checking algorithm works, consider the following (fairly simple, but slightly broken) `PROC` definition:

[9] Undefined uses of mobile variables are always errors. Additionally, strict mode (section 3.9.6) will cause all undefined warnings to be errors.

```
PROC thing (CHAN INT in?, RESULT INT r)
  INT x, y:
  SEQ
    in ? x
    in ? y
    IF
      x < 0
        r := (x + y)
      TRUE
        y := (x - y)
:
```

When the checking of this PROC body starts, the undefined stack will contain two variables — the 'in?' and 'r' parametmers:

	States						
Names	**0**	1	2	3	4	5	...
in?	$\top$						
r	$\bot$						

The states are shown as '$\top$' for *defined*, '$\bot$' for *undefined* and '?' for *unknown*. For channels (such as the 'in?' parameter), information about whether the channel has been used for input or output is recorded, shown as '$c_?$' and '$c_!$' respectively. The current state is indicated in bold — zero in the above initial state.

All parameter types, with the exception of 'RESULT' parameters, are assumed to be defined when the PROC starts. Result parameters are initially assumed to be undefined.

The first items found in the tree-scan of 'thing' are the declarations of 'x' and 'y'. These are added to the variable stack, marked as undefined.

'SEQ' components are simply processed in order, without requiring any changes to the current state. The next items found are the inputs from 'in?', first into 'x' then into 'y'. The processing of (simple) input is that the channel is marked as '*inputted*' ('$c_?$') and any variable targets on the right-hand-side are marked as defined. Prior to processing the 'IF' construct, the variable stack will be:

	States						
Names	**0**	1	2	3	4	5	...
in?	$c_?$						
r	$\bot$						
x	$\top$						
y	$\top$						

The handling of 'IF' requires that each branch be examined individually, and resulting states be merged back together. This is done by first *pushing* the state stack, copying the current state into the new state, then processing the guard and body of IF branch individually. Both branches of the 'IF' in the 'thing' example are simple assignments, both using an expression containing 'x' and 'y' (both defined). The second branch is in error slighty — assuming the assignment was intended to be to 'r', and not 'y'.

After processing the first branch of the IF, the variable stack is:

	States						
Names	0	**1**	2	3	4	5	...
`in?`	$c_?$	$c_?$					
`r`	$\perp$	$\top$					
`x`	$\top$	$\top$					
`y`	$\top$	$\top$					

Once the first branch has finished, the state stack is pushed again (copying values from the pre-`IF` state – 0), and the second branch examined. The resulting states for this are:

	States						
Names	0	1	**2**	3	4	5	...
`in?`	$c_?$	$c_?$	$c_?$				
`r`	$\perp$	$\top$	$\perp$				
`x`	$\top$	$\top$	$\top$				
`y`	$\top$	$\top$	$\top$				

Once the various states have been collected, they are merged back into the pre-`IF` state (0). The merge, applied to each variable in the stack, can be viewed as a function that takes a set of collected states and the original state, producing a new state.

The technicalities of the various merges are described in detail in the following sections. For the example `PROC`, the state after merging the `IF` branches is:

	States						
Names	**0**	1	2	3	4	5	...
`in?`	$c_?$	$c_?$	$c_?$				
`r`	?	$\top$	$\perp$				
`x`	$\top$	$\top$	$\top$				
`y`	$\top$	$\top$	$\top$				

When scanning reaches the end of the `PROC` body, at which point '`x`' and '`y`' have descoped, the formal parameters (left on the variable stack) are checked for any required definedness. The only types of parameter that require checking at this point are `MOBILE` and `RESULT` parameters. If a result parameter is left in an undefined or unknown state, a warning (or error in strict mode) is generated.

`MOBILE` parameters, that are initially assumed to be defined, can be *moved* by the process, undefining the parameter. If there is any undefinedness in a mobile parameter when the `PROC` finishes, this is recorded in the `PROC` signature and no warning is generated. `RESULT MOBILE`s are permitted and must be defined when the `PROC` finishes.

Merging `IF`s and `ALT`s

Algorithm 4.1 shows the algorithm used to merge undefined states following an `IF` or an `ALT`, applied to each variable in the stack. The logic of this merge is that at least one branch must execute — if one does not, the process `STOP`s.

Replicated `IF`s and `ALT`s are only scanned once, generating a new state that is simply copied back into the current state — any nested `IF`s or `ALT`s are processed as above.

Algorithm 4.1: Undefinedness checker `IF/ALT` merging algorithm

// S_i : *initial var state*
// $B = \mathbf{min}(S_{n..m})$: *reduced branch result states*
// S_f : *final state*
if $(? \in B)$ **or** $(B = \{\top, \bot\})$ **then**
  $S_f \Leftarrow ?$
else if $B = \{\top\}$ **then**
  $S_f \Leftarrow \top$
else
  $S_f \Leftarrow \bot$
end if

Merging `WHILE` Loops and `SEQ` Replicators

'`WHILE`' loops are processed for undefinedness twice. When the construct is entered, the variable stack is pushed and the body of the loop processed. The resulting state is then merged back into the before-`WHILE` state, taking into account the fact that the loop may not execute at all. The body is then processed again, using the previous result state. This represents the resulting definedness states if the loop ran more than once. This is then merged back into the pre-`WHILE` state, and the stack reset.

Both merges are performed using Algorithm 4.2. The second pass is used to capture undefinedness that happens as a result of the first time round the loop. For example:

```
MOBILE []BYTE x:
SEQ
  x := "hello, new world!*n"
  WHILE TRUE
    m.out ! x               -- mobile output
```

This code is clearly in error — '`x`' will be undefined after the first scan of the loop, overall resulting in an unknown state.

Algorithm 4.2: Undefinedness checker `WHILE`/repl.-`SEQ` merging algorithm

// S_i : *initial var state*
// S_n : *result state*
// S_f : *final state*
if $(S_n = \bot)$ **or** $(S_n = \top)$ **then**
  if $S_n \neq S_i$ **then**
    $S_f \Leftarrow ?$
  end if
else
  $S_f \Leftarrow ?$
end if

Replicated '`SEQ`'s (loops) are processed in a similar way, using the same algorithm. However, checking for a replicated `SEQ` is only performed if the count is either *unknown*, or known and more than 1. If the count is 1, the undefinedness check is only performed once. In this (rare) case, the outcome is considered definite — the loop must always execute. For example:

```
MOBILE []BYTE x:
SEQ
  SEQ i = 0 FOR 1
    x := "hello, new world!*n"
  m.out ! x                 -- mobile output
```

This code is safe, even though it is a slightly odd thing to write. For other constant-count `SEQ` replicators, the outcome after the second scan is considered the definite result — and is simply copied back into the pre-repl-`SEQ` state.

Merging Parallel Processes

Merging the undefined results of parallel processes is perhaps the simplest merge. This is mainly due to the parallel usage rules enforced by occam— i.e. CREW (concurrent read, exclusive write) for data, and one inputter, one outputter for ordinary channels.

For ordinary data and channels, parallel usage checks can be avoided by explicitly marking the variable (data or channel) as shared ("`#PRAGMA SHARED thing`"). This is *not* permitted for mobile data — the compiler insists on retaining full control over these.

For any `PAR` construct, only one branch of that `PAR` may change the state of a single variable (or a subscript or range – for arrays and record variables). The previous parallel-usage checking has ensured this much. For non-mobile data, the only possible transitions are: $\perp \rightarrow ?$, $? \rightarrow \top$ and $\perp \rightarrow \top$. Thus in any `PAR`, the *best* result from any branch is taken — but for non-mobiles only. The restrictions imposed on mobiles ensure that only a single branch of a `PAR` may change any mobile's state, that is simply taken as the overall (`PAR`) outcome for that mobile. For example:

```
INT x:
#PRAGMA SHARED x
MOBILE []BYTE msg:
PAR
  msg := "this is the message!*n"
  x := 1
  x := 2
```

After processing the `PAR`, the states of both '`msg`' and '`x`' are *defined*. Algorithm 4.3 shows the algorithm used to merge `PAR` branches, as described above.

The handling of replicated `PAR`s is trivial — the replicated process is scanned and the resulting states simply copied back.

4.6.3 Handling Arrays and Records

Arrays and records are non-trivial types as far as the undefinedness-checker are concerned. Both can exhibit *partial-definedness*, for example:

```
[2]INT data:
INT x:
SEQ
  data[0] := 42
  x := data[1]
```

If arrays (and records) were considered to be 'whole', the above code would compile without warning. There is clearly an error however — '`x`' and '`data[1]`' are undefined!

Algorithm 4.3: Undefinedness checker `PAR` merging algorithm

// S_i : initial var state
// $B = \mathbf{min}(S_{n..m})$: reduced branch result states
// S_f : final state
if $S_i = \bot$ **then**
    if $\top \in B$ **then**
        $S_f \Leftarrow \top$
    else if $? \in B$ **then**
        $S_f \Leftarrow ?$
    end if
else if $S_i = ?$ **then**
    if $\top \in B$ **then**
        $S_f \Leftarrow \top$
    else if $\bot \in B$ **then**
        $S_f \Leftarrow \bot$
    end if
else
    if $? \in B$ **then**
        $S_f \Leftarrow ?$
    else if $\bot \in B$ **then**
        $S_f \Leftarrow \bot$
    end if
end if

Record types (from occam 2.1 [Inm95]) are somewhat simpler than arrays — the number of elements is always fixed, although those elements may represent further structured types. As a simple example, consider the type:

```
DATA TYPE POINT
  RECORD
    INT x, y:
    REAL64 size:
:
```

And the following (incorrect) code that uses it:

```
POINT p:
SEQ
  p[x], p[y] := 10, 20
  out ! p                    -- output, partially defined
```

When output, '`p`' is only partially defined — the '`size`' field has not been initialised. To handle these, the '`nested`' field within the '`udv_t`' type (shown on page 108) is used. This array, of the same ('`udv_t`') structure is used to record the state of each individual field, as shown in figure 4.18.

The analysis of arrays is done using *ranges* — a method not dissimmilar to parts of the occam compiler's existing aliasing checker. For example:

```
[24]INT data:
SEQ
  data[0] := 42
  ...  process using "data"
```

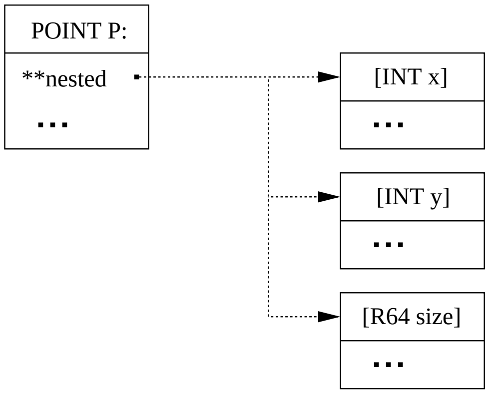


Figure 4.18: Undefined nesting for records

The size of '`data`' is well-known here, but this is not necessarily the case (e.g. formal parameters, abbreviations and dynamic mobile arrays). Figure 4.19 shows a possible layout for '`data`', having had its first element defined, but the rest left undefined.

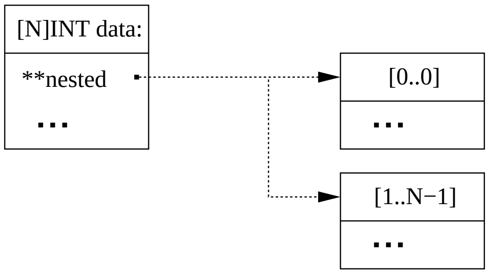


Figure 4.19: Undefined nesting for arrays

As the array element states change, the ranges associated with the array change. Whenever a range is added or modified, the resulting ranges are 'compacted' if possible. For example, the code:

```
SEQ
  [data FOR 12] := [i = 0 FOR 12 | i]
  [data FROM 12] := [i = 0 FOR SIZE [data FROM 12] | i * 2]
```

leaves all of '`data`' defined. Assuming the initial state from figure 4.19, the first assignment will create a new range from [1..11], and modify the second (undefined) range to start at 12. Compacting the ranges (that are ordered) merges the first two (defined) ranges — [0..0] and [1..11], into [0..11]. The second assignment marks the second range as defined, which is then merged into the single range [0..23].

Nested arrays and records are handled by 'widening' the undefined structure. The compiler stops tracking the nested state after 4 levels of nesting. This prevents excessive memory consumption for pathological code, but should be enough for most programs; for example:

```
DATA TYPE COMPLEX.R64
  RECORD
    REAL64 real, imag:
:

DATA TYPE RC.POINTS
  RECORD
    INT n.points:
    [64]COMPLEX.R64 x, y:
:

[2][3]RC.POINTS objects:
SEQ
  objects[0][1][x][5][real] := 24.4 (REAL64)
  ...  code manipulating "objects"
```

Here, the undefinedness checker will not track the innermost '`COMPLEX.REAL64`' fields — that record will left in an '*unknown*' state.

Keeping track of ranges within an array is relatively simple, given occam's methods for handling these. Accesses are either to individual elements, to the entire array, or a specific *slice* of the array. When element and slice indices become too complex for static analysis, the compiler resorts to an 'unknown' state for the access. The incorporation of an algabraic usage checker could help significantly here.

4.7 Usage Checking Channel Types

Currently there is no enforcement of communication order in occam, amongst the channels within a channel-type, even though processes may require it. Even the most basic occam processes may have communication ordering requirements, that are not explicit. Fortunately, however, the usage is often obvious.

This section presents a proposal for a '`TRACES`' mechanism, that specifies restrictions on communication within channel-types. This allows the compiler to ensure conforment client (and server) processes, guaranteeing deadlock freedom as regards process interactions using that channel type. The following example shows the proposed syntax, for a relatively simple channel type:

```
CHAN TYPE FOO
  MOBILE RECORD
    CHAN INT req?:
    CHAN INT reply!:
  TRACES
    SEQ
      req?
      reply!
:
```

In the same way as the fields of a channel type, traces are specified from the server viewpoint. Furthermore, processes using the server-end of such a channel-type *must* implement the given trace exactly. Processes using the client-end of a channel-type are slightly less resticted — any implementation that can communicate successfully with a server process performing the specified trace is permitted. For the above example, a server implementation must communicate sequentially. Client processes could communicate in the sequence shown, or in parallel. However, attempts (at the client-end) to communicate in the reverse sequential order would result in a compiler error.

If the trace in the above channel-type declaration was changed to 'PAR' (with the same two elements), server processes would be restricted to parallel communication, whereas a client could either communicate in parallel or sequentially (in either order).

For some traces, parallel implementations are not possible — particularly when the same channel appears more than once in a given trace. For example:

```
TRACES
  SEQ
    req ?
    reply !
    req ?
```

Multiple traces are supported, but only through the use of explicit selection — either ALT or CASE. For handling CASEs (of variant protocols), all possible variants would need to be specified, but could be 'condendensed' — to the form of a standard occam CASE process.

Chapter 5

Dynamic Process Creation

This chapter describes three extensions that add a dynamic process capability to occam, in the form of dynamic process creation. All of these require support at the Transputer instruction level for dynamic memory allocation, as described in section 3.8. A reference for these instructions is given in Appendix B.1.1.

The first extension provides support for self-recursion in occam and is described in section 5.1. Mutual recursion was considered for implementation, but rejected due to its need for forward-references — section 5.1.4 examines this in more detail.

The second extension, described in section 5.2, provides support for run-time sized `PAR` replicators — enabling the creation of an arbitrary number of parallel processes which (barrier) synchronise before the `PAR` finishes.

The third extension, described in section 5.3, provides a more general mechanism for dynamic parallel process creation — the `FORK` — without explicitly using the `PAR` process constructor. Barrier synchronisation for *these* dynamic processes is optional, and need not be local.

5.1 Recursion in occam

Recursion in occam has traditionally been prevented for two main reasons — one practical and one engineered. Firstly, the lack of dynamic memory allocation would have imposed a restriction on the depth of recursion. Secondly, the scoping of names in occam is such that they only become visible at the end of their declaration (where the '`:`' appears). For `PROC`s, this means that any use of its own name inside the code body is invalid, unless a different `PROC` of the same name is in scope — in which case it will be used.

It is possible to fake recursion however, often quite convincingly, by using the scoping of names to an advantage [Poo92]. In an arbitrary `PROC` called '`foo`', any previously defined `PROC`s also called '`foo`' are in scope and perfectly valid. This technique will fail on some KRoC implementations however — when the same name is defined at the outermost level multiple times, the UNIX linker will mostly likely stop with a "multiply defined symbol" error. The tranx86 translator partially handles this by only allowing one entry-point for any given name per file processed. However, the same top-level `PROC` name in separate source files will also usually result in a similar error on UNIX platforms — which is a problem generally, not just for recursion.

A version of unbounded recursion using a special locally defined `PROC` — with a very similar name — has been implemented in a development version of the KRoC/Sparc by Wood in [Woo00]. The Linux version of KRoC implements recursion using a different mechanism, by adding support directly to the occam compiler. The support for recursion in KRoC/Sparc also uses a Brinch-Hansen style memory allocator, but with with all support for such in the translator (octran) and run-time kernel only.

For this implementation, recursive PROCs are declared using the 'RECURSIVE' or 'REC' keywords — the latter being shorthand for the former. For example:

```
RECURSIVE PROC thing (...)
  ...  body of thing
:
```

This has the effect of bringing the name 'thing' into scope early, thereby permitting its use within the body of 'thing'. A classic example for parallel recursion in occam is the parallel recursive Sieve of Eratosthenes. The original code for this is given in Appendix A of Moores's CCSP paper, [Moo99], and requires very little change — just the addition of the 'RECURSIVE' keyword. The new code for the 'sieve' process, augmented with channel-direction specifiers, is:

```
RECURSIVE PROC sieve (VAL INT count, CHAN INT in?, out!)
  IF
    count = 0
      id (in?, out!)    -- just copy in to out
    TRUE
      INT n:
      SEQ
        in ? n
        out ! n
        CHAN INT c:
        PAR
          filter (n, in?, c!)
          sieve (count-1, c?, out!)
:
```

The 'count' parameter is used to limit the recursion. An initial value of 169 will allow for all the primes between 2 and 1000000 to be generated. When the 'sieve' process receives its *first* input (from 'in?'), it outputs it down the prime output channel 'out!', then starts a network of sub-processes: a filter to filter out all multiples of 'n', in parallel with a recursive call to sieve. When 'count' reaches zero, the recursion stops.

Some time after new filter processes stop being created, the pipeline will start to produce invalid primes. However, this point is detectable: as long as the numbers output remain less than the square of the last filtered prime, they are guaranteed to be prime. The program can be modified slightly to check for this, deadlocking before generating false primes:

```
PROC end.sieve (VAL INT n, CHAN INT in?, out!)
  VAL INT max IS n * n:
  CHAN INT c:
  PAR
    filter (n, in?, c!)
    INITIAL INT n IS n:
    WHILE n < max
      SEQ
        out ! n
        c ? n
:
```

```
RECURSIVE PROC sieve (VAL INT count, CHAN INT in?, out!)
  INT n:
  SEQ
    in ? n
    IF
      count = 1
        end.sieve (n, in?, out!)
      TRUE
        SEQ
          out ! n
          CHAN INT c:
          PAR
            filter (n, in?, c!)
            sieve (count - 1, c?, out!)
:
```

With an initial count of 4, this sieve will generate all the primes between 1 and 121.

To show recursion diagrammatically, a fairly simple approach is taken: show the recursive and final (if applicable) networks once. Figure 5.1 shows this for the modified prime sieve program.

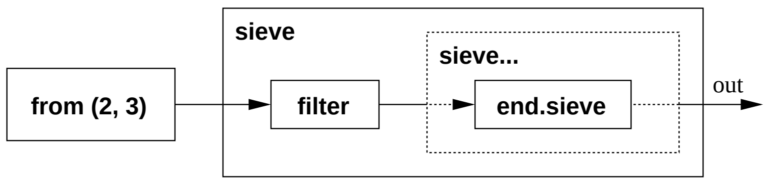


Figure 5.1: Process network for the parallel recursive Sieve of Eratosthenes

5.1.1 Implementing Recursion

The implementation of recursion within the occam compiler is non-trivial. Making a recursive call involves three steps: firstly, allocating memory for the process; secondly, making the recursive call (that involves setting up parameters); and finally, cleaning up when the recursive call returns.

The run-time model for standard occam PROC calls is optimised for efficiency on the Transputer — that performed many operations in hardware. The Transputer 'CALL' instruction, which is used for standard PROC (and FUNCTION) calls, utilises the Transputer stack for passing arguments. Arguments that will not fit into the stack (3 words deep) are pre-stored in workspace allocated for the PROC call. When the 'CALL' is made, the contents of the Transputer stack are placed in the workspace — along with the return address (previous 'Iptr') — and the workspace pointer 'Wptr' is adjusted.

For PROC calls that have 3 or fewer parameters (including hidden parameters), the code generated for a call is trivial. For example, the occam fragment:

```
PROC foo (VAL INT v, INT r)
  ...  code (not requiring anything other than workspace)
:

foo (x, n)
```

will generate the following calling code:

```
LD ADDRESSOF n              -- pointer to n
LD x                        -- value of x
CALL foo
```

Figure 5.2 shows the workspace allocation for the call to '`foo`', and also another for a call to '`foo5`' (a version that takes 5 parameters). In the latter case, the allocation of the two parameters that do not fit into the transputer stack is performed in the workspace of the *calling* process. Conversely, the (theoretical) minimum workspace for a `PROC` is 4 slots[1] (3 for parameters and one for the return address).

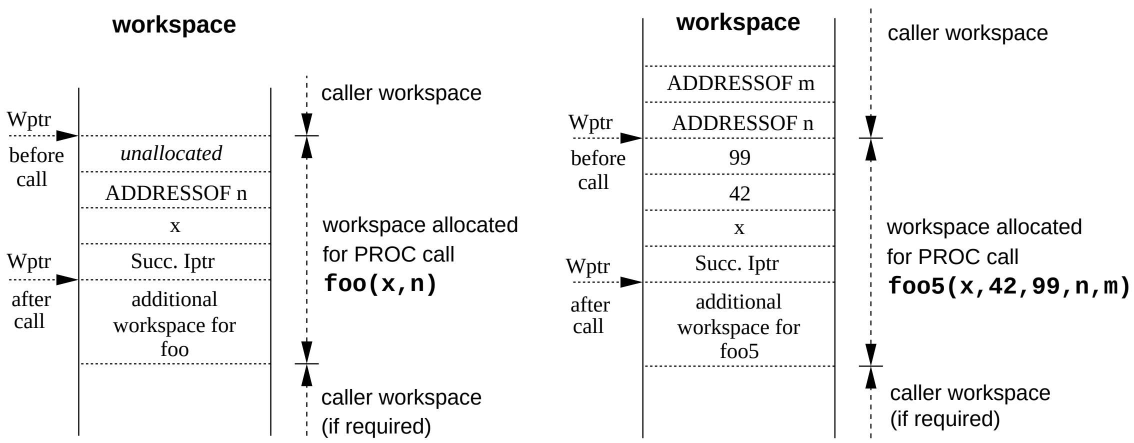


Figure 5.2: Workspace allocation for `occam` procedure calls

The key feature about the parameter allocation is that the called process ('`foo`' and '`foo5`' here) finds all parameters in and above its own workspace. When the compiler maps out workspace, `PROC` calls are allocated their given size *plus* any additional space required for parameter passing. In the case of '`foo`', this additional space is zero. For '`foo5`' it is 2 slots. Because the '`foo`' call does not use all stack entries when `CALL`ed, the third parameter slot is left spare. The compiler knows this when allocating workspace for '`foo`' itself, and as such will use spare parameter slots for local variable allocation.

In order to support recursion, that requires the allocation of dynamic workspaces, parameter allocation must be handled differently — such that parameters that would normally be allocated in the caller's workspace are allocated instead in the called process's workspace. Figure 5.3 shows the workspaces allocated for recursive versions of '`foo`' and '`foo5`'.

A new set of routines have been added to the back-end (mapping and code-generation) of the compiler, that handle the allocation of parameters in a separate workspace. These parameter handling routines are also used for the '`FORK`' extension, described in section 5.3. From within the called process, the parameters and "successor Iptr" (return address) appear the same — meaning that the first (non-recursive) call to a recursive `PROC` is treated normally, and allocated in the caller's workspace.

[1] The actual minimum workspace requirement for a `PROC` is more than 4 slots however, since low workspace is reserved for the run-time state if the `PROC` deschedules.

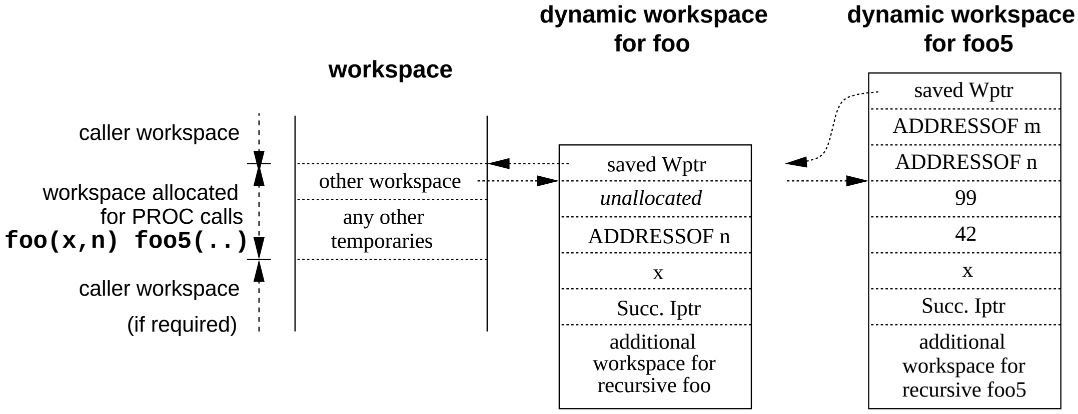


Figure 5.3: Workspace allocation for recursive occam procedure calls

The workspace allocated for a *recursive* call is constant. At a minimum this is a single slot — for storing a pointer to the allocated workspace[2]. Two additional slots may also be allocated, one for the recursive vectorspace, and one for the recursive mobilespace *temporary* (covered in section 5.1.2).

To make a recursive call, the compiler first allocates the required dynamic memory, storing pointers in the current workspace as it goes. The parameters are then evaluated and stored in the dynamic workspace, along with the current 'Wptr' — that is used as a link back when the recursive call finishes. To call the recursive process, and to recover the previous process afterwards, the compiler generates the following code:

```
LDL -1              -- load dynamic workspace pointer
GAJW                -- swap Wptr and Areg (general adjust workspace)
POP
AJW xxx             -- adjust for GCALL
LD ADDRESSOF foo    -- address of procedure to call
GCALL               -- swap Iptr and Areg (general call)

LDL yyy             -- load saved Wptr
GAJW                -- make it current
POP
```

The constant values 'xxx' and 'yyy' depend on the recursive process being called, affected by the number of parameters. After the recursive call has returned, memory dynamically allocated for workspace and vectorspace is freed.

5.1.2 Mobilespace For Recursive Processes

The mechanisms required to support static mobilespace for this and other dynamic process extensions (covered in this chapter), are substantially different from the workspace and vectorspace support for such. The dynamic *allocation* of mobilespace is largely similar to the dynamic allocation for workspace and vectorspace — the required amount of mobilespace memory is a known constant

[2]The workspace pointer stored for a recursive call points into the allocated block, rather than at the base of it. The compiler generates code to adjust the workspace appropriately where required.

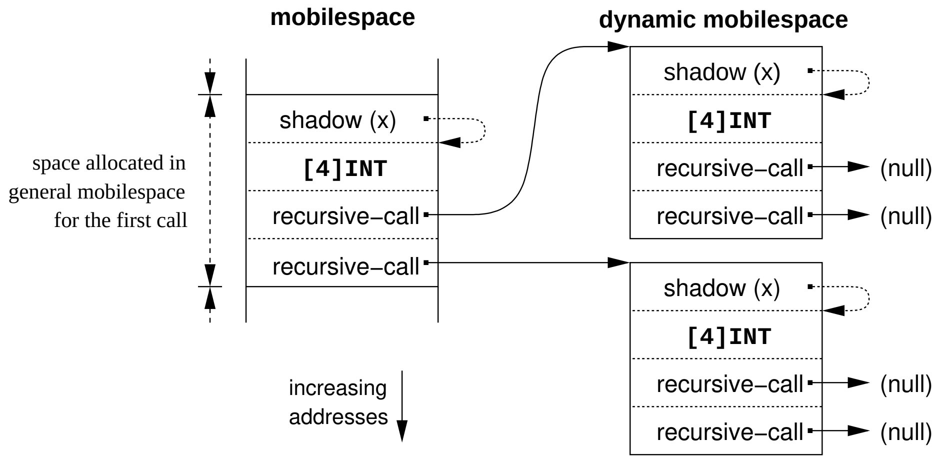


Figure 5.4: Dynamic mobilespace allocation for recursive processes

and is allocated using either 'MALLOC' or 'MNEW'. However, dynamic mobilespace can never be returned to the memory pool. The reason for this is simple: pointers within a dynamic mobilespace may have migrated into other mobilespaces, that may in turn have acquired pointers to either the system-wide static mobilespace or another dynamic process's mobilespace. Freeing a dynamic mobilespace to the common memory pool is potentially disastrous — when it is reallocated for a different purpose, for example.

Instead, after use, dynamically allocated mobilespaces are carefully stored into the enclosing mobilespace, allowing them to be recycled.

Recursion has the simplest dynamic mobilespace allocation: a single slot in mobilespace, initialised to null, is used to store a pointer to a dynamically allocated mobilespace. When this is allocated and initialised at run-time, a new mobilespace 'hook' is created, ultimately resulting in a tree of linked dynamic mobilespaces. The following occam fragment shows an example of a recursive PROC that uses mobilespace:

```
RECURSIVE PROC filter (VAL INT n, CHAN MOBILE [4]INT in?, out!)
  IF
    n = 0
      MOBILE [4]INT x:
      WHILE TRUE
        SEQ
          in ? x
          ...  process 'x'
          out ! x
    TRUE
      CHAN MOBILE [4]INT c:
      PAR
        filter (n-1, in?, c!)
        filter (n-1, c?, out!)
:
```

This PROC sets up a pipeline of 2^n processes, communicating fixed-size mobile arrays. Figure 5.4 shows the resulting mobilespace layout after this PROC is called with 'n' as 1 (to create a pipeline of

2 processes). The amount of dynamic memory ultimately left 'hanging' from the main mobilespace is determined by the maximum depth of recursion at that point.

One slight deficiency of this technique is that there may be many top-level hooks for dynamic mobilespace — at a minimum, one for every parallel call to a recursive PROC. In practice there may be many such hooks, since the effectiveness of mobilespace packing (section 4.2.2) is limited.

5.1.3 Tail Call Optimisation

For recursive processes that use *sequential* tail-call recursion, it is possible to optimise away the recursion and re-use the memory allocated for the current instance (most likely within the workspace of another process). There are some restrictions on when this is possible, however, since no pointer into the current instance's workspace may be passed to the recursive instance. A traditional example might be the factorial function, with the recursion 'wrapped-up' inside:

```
PROC factorial (VAL INT n, RESULT INT v)

  RECURSIVE PROC sub.fac (VAL INT i, n, RESULT INT v)
    IF
      i = 1
        v := n
      TRUE
        sub.fac (i-1, i*n, v)
  :

  sub.fac (n, 1, v)
:
```

This is an ideal candidate for tail-call optimisation, fulfilling the requirements that the recursive call is sequentially last, and that the parameters passed to the recursive call are not local pointers. This remains the case even if the IF is re-arranged:

```
    IF
      i > 1
        sub.fac (i-1, i*n, v)
      TRUE
        v := n
```

The code generated by the compiler for the tail-call optimisation is trivial: the new parameters are evaluated (using temporaries for the expressions), then stored in the current workspace (over-writing the existing parameters). To re-enter the PROC, the workspace-pointer is adjusted if necessary and a jump made to the PROC's entry-point.

5.1.4 Mutual Recursion

The mechanism for recursion described so far is restricted to self-recursion. Mutual recursion, and any other cyclic call patterns, are not allowed (rather than explicitly banned). To permit mutual recursion, a mechanism for forward references would need to be provided — the 'RECURSIVE' keyword applies recursive behaviour to only the PROC which follows it.

Technically, supporting mutual recursion is possible, since it requires no special handling. However, forward references (even if they are explicitly made for recursive purposes), significantly damage the scoping rules of occam.

Assuming the availability of a 'FORWARD' declaration mechanism, the following might be attempted for mutual recursion:

```
FORWARD PROC bar (..):

PROC foo (..)
  SEQ
    ...  do something or STOP
    bar (..)
:

PROC bar (..)
  SEQ
    ...  do something or STOP
    foo (..)
:
```

One thing which is immediately unclear is where the 'RECURSIVE' keyword should be placed. In the above code, this keyword is omitted entirely, leaving the 'FORWARD' keyword to indicate recursive intent — the call of 'foo' from 'bar' is non-recursive.

There is a slightly more serious problem, however, related to the correct scoping of names. Adding to the end of the previous (mutually recursive) code:

```
PROC bar (..)
  SEQ
    ...  another implementation
    foo (..)
:
```

With two definitions of 'bar', there is ambiguity in which definition 'FORWARD' applies to, and hence, which 'bar' should be called from 'foo'. To avoid these problems, the only form of recursion allowed is self-recursion.

The effect of mutual recursion is possible, however, using self-recursion and a 'switch'. For example, the mutual recursion in the above 'foo' and 'bar' processes could be implemented using:

```
RECURSIVE PROC foo.bar (VAL BOOL do.foo, ..)
  PROC foo (..)
    SEQ
      ...  do something or STOP
      foo.bar (FALSE, ..)
  :
  PROC bar (..)
    SEQ
      ...  do something or STOP
      foo.bar (TRUE, ..)
  :

  IF
    do.foo
      foo (..)
    TRUE
      bar (..)
:
```

Adding a second declaration for 'bar' immediately after the first does not introduce any ambiguities either — since it is only referenced *after* its declaration.

CSP does not treat mutual (and arbitrary) recursion well either, which is maybe reason alone for not supporting them, but provides a syntax and semantics for self-recursion. The addition of support for recursion thus provides a direct translation between recursive CSP and occam processes. For example, the (rather dangerous) occam code:

```
RECURSIVE PROC P (CHAN INT in?, out!)
  SEQ
    INT v:
    SEQ
      in ? v
      out ! v + 1
    CHAN INT c:
    PAR
      P (in?, c!)
      P (c?, out!)
:
```

Can be expressed (in a parameterised form of) CSP, as:

$$P(in, out) = \mu\, Q(in, out) \bullet in?v \rightarrow out!(v+1) \rightarrow (Q(in, c) \underset{\{c.i | i \in \mathbb{N}\}}{\|} Q(c, out)) \setminus \{c.i \mid i \in \mathbb{N}\}$$

Technically speaking [Hoa85, Law02] sequential composition involves processes synchronising on the '$\checkmark$' event, used to model CSP's sequential composition operator '$\fatsemi$'.

5.2 n-replicated PARs

The lack of variable PAR replication has always been a frustrating feature in occam. A common reason for wanting it is to allow the construction of n-way parallel 'delta' processes:

```
PROC delta.n (CHAN INT in?, []CHAN INT out!)
  WHILE TRUE
    INT v:
    SEQ
      in ? v
      PAR i = 0 FOR SIZE out!
        out[i] ! v
:
```

Because the size of 'out!' is not known when the PROC is compiled, the compiler rejects the replicated 'PAR'. However, where this PROC is actually called, the compiler probably will know the number of channels being passed — and could still allocate a static amount of space for the PROC call. This functionality is not part of the compiler yet, as it would require significant changes to the way in which workspace sizes are represented inside the compiler (using expressions instead of constants)[3]. Changes would also be required in the parallel usage checker, probably more significant than those required to support workspace sizes as expressions.

[3]Such modification is technically possible though, and may be investigated in the future.

The current solution to writing an n-way delta is to enforce an upper limit on the number of channels used, for example:

```
VAL INT MAX.DELTA.SIZE IS 32:

PROC delta.n.max (CHAN INT in?, []CHAN INT out!)
  WHILE TRUE
    INT v:
    SEQ
      in ? v
      PAR i = 0 FOR MAX.DELTA.SIZE
        IF
          i < (SIZE out!)
            out[i] ! v
          TRUE
            SKIP
:
```

Following from this, instead of full n-replicated `PAR` support, a limited form *could* be provided that allows a variable replication with an upper limit. Given some suitable modifications to the compiler, this might be used to implement the n-way delta as:

```
VAL INT MAX.DELTA.SIZE IS 32:

PROC delta.n.limit (CHAN INT in?, []CHAN INT out!)
  WHILE TRUE
    INT v:
    SEQ
      in ? v
      PAR i = 0 FOR (SIZE out!) LIMIT MAX.DELTA.SIZE
        out[i] ! v
:
```

The modifications required to implement this would, in practise, be relatively simple. For the purposes of usage checking, the compiler could simply assume that '`SIZE out!`' was its specified '`LIMIT`'. For the mapping and code-generation, enough space for '`MAX.DELTA.SIZE`' replications would be reserved, but only '`SIZE out!`' performed. A run-time check before the '`PAR`' could check that '`SIZE out!`' was in range.

Instead, the chosen route has been to implement full run-time support for n-replicated `PAR`s (allowing the implementation of the earlier '`delta.n`' process). The issue of parallel usage-checking is largely avoided — reduced to a compiler warning — discussed in section 5.2.3. The implementation of the n-replicated `PAR` is described in section 5.2.1, first describing the implementation of the existing (but modified slightly) replicated `PAR`. Support for n replications relies heavily on dynamic memory allocation support, covered in section 3.8. Supporting mobilespace for n-replicated `PAR`s is more complex, and covered separately in section 5.2.2.

5.2.1 Implementing n-replicated PARs

The existing code for supporting fixed-count replicated `PAR`s is somewhat extensive. Rather than re-implementing '`REPLPAR`' (the tree-node type that defines a replicated `PAR`), the existing code has been extended (substantially in some places).

Constant Count Replicated PARs

The workspace for a fixed-size replicated PAR is allocated as a single block in the enclosing workspace. The amount of workspace required is calculated by multiplying the (constant) number of replications by the total workspace size for the replicated process, plus a fixed overhead. If the replicated process requires either vectorspace or mobilespace, these are allocated as single blocks in the enclosing vectorspace and mobilespace respectively.

Figure 5.5 shows the workspace layout while setting up a *fixed-count* replicated PAR. The creation of processes is handled by a loop, that sets up each process's initial workspace before calling 'STARTP' for it. The process performing the replicated PAR has a number of special slots reserved within it, used for both setting up the replicated PAR and for shutting it down. The three slots "JoinLab", "ParCount" and "SavedPriority" are used for all PAR constructs, accessed by 'ENDP' when a parallel process terminates. When 'ENDP' is called (passing a pointer to 'JoinLab'), the value in 'ParCount' is decremented. If it reaches zero, 'ENDP' returns directly to the address stored as 'JoinLab', possibly changing priority to 'SavedPriority' on the way (priority is covered fully in section 7.2).

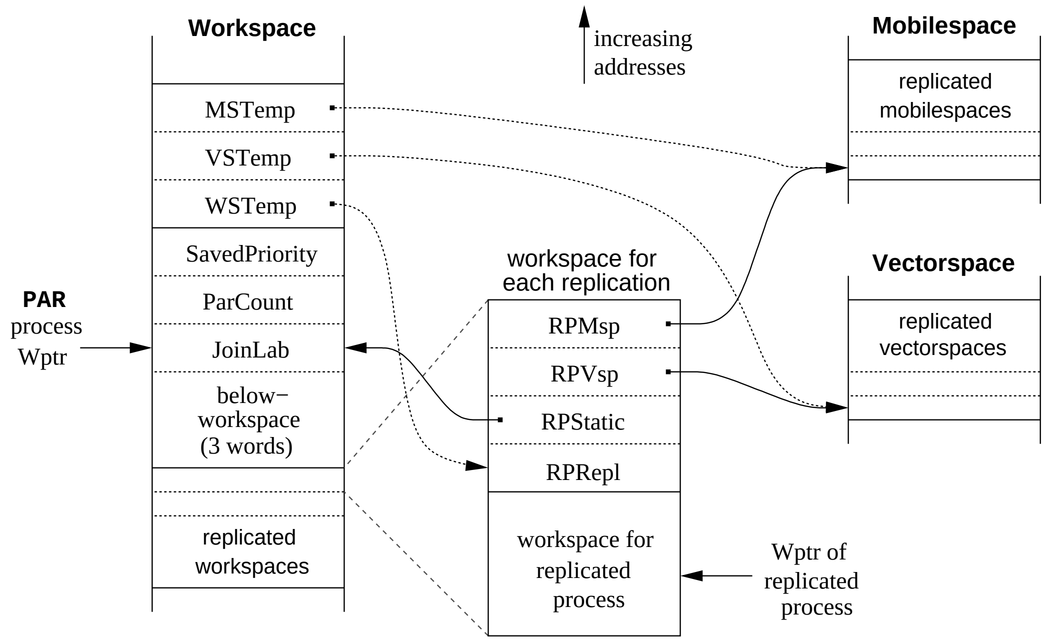


Figure 5.5: Setting up a fixed-size replicated PAR

The "WSTemp" reserved word is always allocated, and is used during the setting up of each replicated process. "VSTemp" and "MSTemp" are similar, but are only allocated and used if the replicated process requires vectorspace and/or mobilespace respectively.

When a replicated PAR begins, the 'ENDP' specials and replicated-specials are initialised, along with the workspace required for the replicator itself (not shown). The replicator count is set to the number of replications *plus* one — once all the replicated processes have been started, the process doing the PAR calls 'ENDP' itself.

"VSTemp" and "MSTemp", where required, are initialised to the starts of the vectorspace and mobilespace reserved for the replicated processes. "WSTemp" is initialised to point at a fixed offset within the first replicated process's workspace (to the "RPRepl" slot).

Each iteration of the process creation loop (run 'ParCount' -1 times — the original PAR replicator count), initialises the workspace of the process held in 'WSTemp', copying the replicator value into 'RPRepl', the current workspace-pointer into 'RPStatic' (used to access variables outside the PAR), and 'VSTemp'/'MSTemp' into 'RPVsp'/'RPMsp' if required. 'STARTP' is then called, passing 'WSTemp' minus a fixed offset for workspace used by the replicated process itself. For trivial replicated processes (such as the output in a fixed-count INT delta), this offset will be zero. After calling 'START', the values of 'WSTemp', 'VSTemp' and 'MSTemp' (as required) are incremented by the size of the replicated process's workspace, vectorspace and mobilespace respectively.

When the loop terminates, it simply calls 'ENDP' with its own workspace pointer as the successor. There is a possibility that by the time the PAR process exits the process creation loop, all the created processes have already run and terminated. Therefore, not using 'ENDP' at this point (and initialising 'ParCount' to n rather than $n + 1$) might introduce a race-hazard, between the last section of loop-code and the code at 'JoinLab' (after the 'PAR'), reached by the last terminating replicated process.

When a replicated process terminates, it loads the value of 'RPStatic' (pointing at the 'successor' workspace) and calls 'ENDP'. If it is the last replication that has terminated, execution continues at 'JoinLab' — the code directly after the replicated PAR.

Variable Count Replicated PARs

The implementation of the *variable-count* replicated PAR is similar to the fixed-size version, both sharing large amounts of code within the compiler. Instead of reserving memory for replicated processes inside the enclosing workspace, vectorspace or mobilespace, memory for the replicated processes must be allocated at run-time — the replication count is unknown.

The handling of workspace and vectorspace is reasonably simple: memory is allocated before the replicated PAR and freed afterwards. Mobilespace handling for replicated processes is complicated by the fact that once allocated, mobilespace cannot be freed — and memory leaks are undesirable. Section 5.2.2 covers the handling for n-replicated mobilespaces.

A scale of allocation strategies for workspace and vectorspace are possible — from allocating the memory for each replicated process individually, through to allocating a single block of memory for all replicated processes. The implementation currently uses the fine-grained approach, allocating workspace and vectorspace individually for each replicated process. The 'granularity' of memory allocation here affects two things: reuse on subsequent instances of the replicated PAR, if the replication-count varies over time; and the space/time efficiency of dynamic memory usage.

More process memories per memory-block causes larger allocations, possibly leading to wastage (as the memory-pool sizes increase exponentially), but overall there will be less calls to the memory allocator. Furthermore, more processes-per-allocation will decrease the likelihood of that same memory being re-used (through the memory allocator) on subsequent calls, with different replicator counts. Fewer processes per memory allocation make maximum possible re-use of memory[4], but suffer more the overheads of the memory allocator.

To keep track of memory allocated for a variable replicated PAR, an array is used to hold pointers to dynamically allocated workspaces. This array is created dynamically at run-time, sized according to the number of replicated processes. A pointer to this array is allocated in the workspace of the process doing the PAR, called "WSArray", along with a temporary used to hold its (byte) size,

[4]Given the quantisation of the memory allocator (into half-powers of 2, covered in section 3.8), it might be the case that allocating more process memories per memory blocks leads to better space efficiency, through less wastage in the memory allocator. Mathematically speaking, this is certainly true for certain process memory sizes, but as yet has not been fully investigated.

"ACount". If vectorspace is used by the replicated process, another workspace slot is reserved, "VSArray". 'ACount' is initialised to zero, incrementing as processes are allocated and decremented as they are freed.

A simple process that makes good use of a variable replicated `PAR` is the n-way delta process, '`delta.n`' (as already mentioned). This has a very small memory requirement — 16 words of workspace for the static part of '`delta.n`' (including parameters), and 6 words of workspace for each replicated process:

```
PROC delta.n (CHAN INT in?, []CHAN INT out!)
  WHILE TRUE
  INT v:
  SEQ
    in ? v
    PAR i = 0 FOR SIZE out!
      out[i] ! v
:
```

Figure 5.6 shows the static and dynamic workspaces for '`delta.n`', after the replicated processes have been started. Initialisation of processes is performed in much the same way as it is for fixed-size `PAR` replicators, except that the workspace is allocated through '`MALLOC`'. Before the loop that creates processes starts, an initial workspace is allocated and stored in 'WSTemp' (with the appropriate pointer adjustment). The code within the process creation loop initialises the special slots for the process at 'WSTemp' (as before), and stores a re-adjusted pointer to it into 'WSArray', indexed by 'ACount', before calling '`STARTP`' to add the new process to the run-queue. Following this, before the loop-end, a new memory block is allocated and stored, with the appropriate adjustment, in 'WSTemp'.

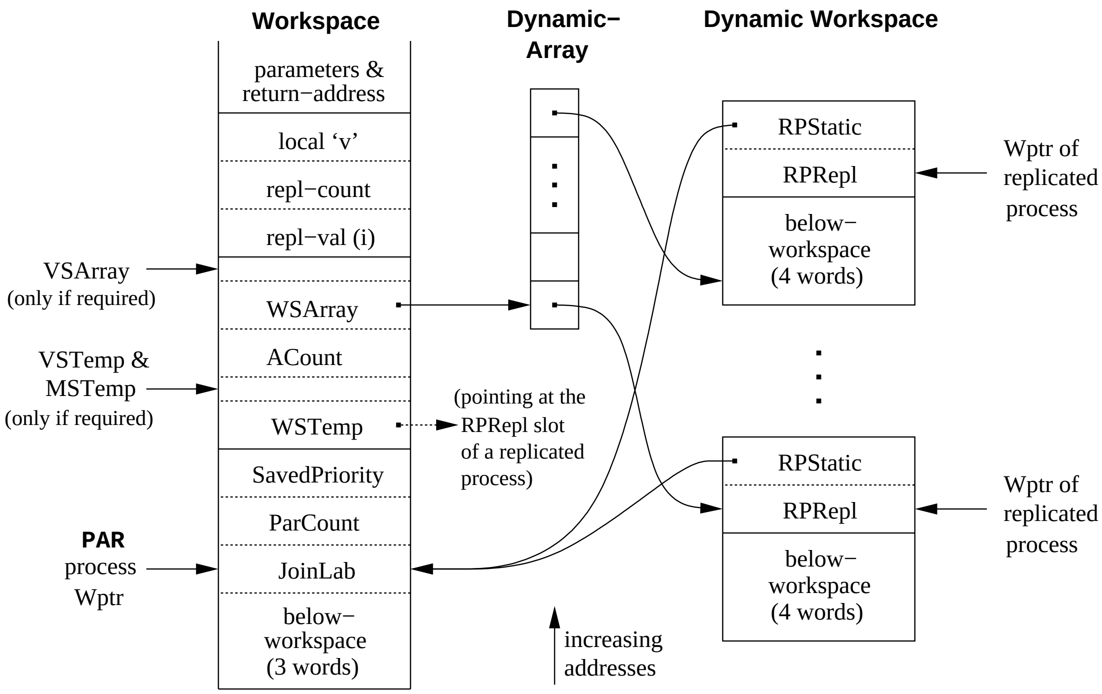


Figure 5.6: Setting up the n-way delta using a n-replicated `PAR`

When the process creation loop finishes, the extra workspace is freed, before calling 'ENDP' in a similar manner to before. The implementation is done this way since it fits better with the existing compiler code for replicated PARs (and replicators in general).

Once all the parallel processes have run to completion, execution resumes at the address in 'JoinLab'. For fixed-size replicated PARs, this is simply the code following the PAR. For the variable replicated PAR, code is generated at 'JoinLab' to iterate over 'WSArray', freeing the dynamic workspaces as it goes, before freeing the dynamic array itself. This is combined with code to free replicated vectorspaces, if required by the replicated process.

5.2.2 Mobilespace Support for n-replicated PARs

Dynamic mobilespace allocation for the n-replicated PAR follows on from similar support for dynamic mobilespace allocation in recursive processes (discussed in section 5.1.2). Instead of using a single slot in mobilespace, two slots are allocated. The first is used to point to an array, that contains pointers to dynamic mobilespaces or 'MINT'. The second slot is used to store the number of elements in the array. When the replicated PAR begins, the number of replications is checked against the second (count) mobilespace slot. If greater, a new pointer array is allocated, and elements from the old pointer array are copied over. The old array is then freed and fresh entries in the new array initialised to 'MINT'.

As each replicated process starts up, it checks the value in the dynamic array used to hold mobilespaces. If 'MINT', a new block of mobilespace is allocated, initialised[5] and stored in the array.

The common way to create a pipeline of processes is to use PAR replication, rather than recursion, and this can now be done dynamically. For example:

```
PROC filter2 (VAL INT n, CHAN MOBILE [4]INT in?, out!)
  PROC in.filter2 (CHAN MOBILE [4]INT in?, out!)
    MOBILE [4]INT x:
    SEQ
      in ? x
      ...  process ‘x’
      out ! x
  :

  MOBILE []CHAN MOBILE [4]INT chans:
  SEQ
    ASSERT (n > 1)
    chans := MOBILE [n-1]CHAN MOBILE [4]INT
    PAR
      in.filter2 (in?, chans[0]!)
      PAR i = 0 FOR n-2
        in.filter2 (chans[i]?, chans[i+1]!)
      in.filter2 (chans[n-2]?, out!)
:
```

Figure 5.7 shows the mobilespace layout for such a process, before any data has been communicated. Although this code is sound, the compiler is unable to usage-check channel access, and generates an

[5]The standard method of initialising mobilespace is on PROC entry, which suffices for recursion and FORKed processes. Code to perform mobilespace initialisation for n-replicated PARs is inserted at the point where the replicated process (which need not be a simple PROC call) starts.

appropriate warning message at compile-time. The provision for a dynamic mobile array of channels is a somewhat odd case — it may only be communicated or assigned when not in use, limiting its usefulness as a mobile entity. As demonstrated in the above example, support for these is driven by the desire to dynamically create arrays of channels, not to have mobile channels. To be useful as a mobile, channels must be described in terms of *ends* (catered for in section 4.3).

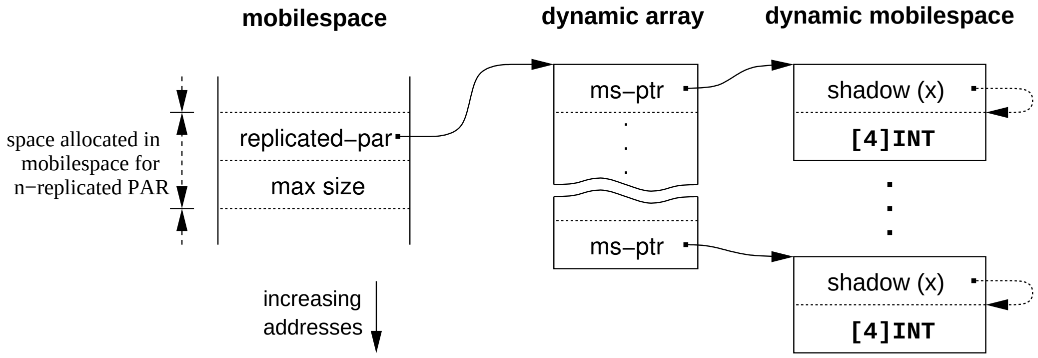


Figure 5.7: Dynamic mobilespace allocation for n-replicated PARs

5.2.3 Usage Checking n-replicated PARs

The one major, perhaps serious, deficiency of the n-replicated PAR is its lack of rigorous parallel usage checking. By default, the occam compiler performs static analysis on replicated PARs, iterating through each replication, checking for parallel usage violations as it goes. Static analysis like this is unsuitable for the n-replicated PAR, since the number of replications is determined only at run-time.

For n-replicated PARs that have constant start and stride expressions, some use can be made of the existing usage checking functionality. In such cases, the body of the replicated PAR is checked *twice*, with the first two replicator values (which can only be determined if the start and stride are constant). If this succeeds, the whole PAR is assumed to be mostly safe (a compiler warning is generated). At run-time, however, this might not be the case — hence the question of safety in this implementation. Where the start is non-constant *and* not-known, virtually no parallel usage checking can be performed — and the compiler generates an error to this effect. However, traditional division of array elements amongst parallel processes is checkable to some extent (assuming the stride is constant), since the replicator starts are at least known, even if not constant. For example:

```
PROC array.sin ([]REAL64 vals)
  PAR i = 0 FOR (SIZE vals) / 4 STEP 4
    PAR j = i FOR 4
      vals[j] := SIN (vals[j])
:
```

This checking, and its deficiencies, extend only as far as the replicator value — i.e. expressions that involve the replicator. Parallel usage checking for processes *inside* an n-replicated PAR is left unhindered.

The majority of these problems could be solved by the use of an algebraic usage checker, in a similar way to the one employed in SPOC [NDHW94]. Much of the parallel and alias checking within SPOC relies on the 'Omega Test' [Pug92, PW95], that is used to check for arbitrary solutions to sets of linear equations — in particular, whether two array indices are the same when accessed in parallel, or whether two abbreviations are in fact the same element. For non-determinable cases, run-time checks are needed — the compiler already generates these types of run-time check for certain code, for example:

```
PROC thing ([]INT data, VAL INT i, j)
  INT v IS data[i]:
  INT w IS data[j]:
  SEQ
    ...  code operating on ‘v’ and ‘w’
:
```

It is hoped to resolve these issues in KRoC at some point, but that will require extensive work within the somewhat delicate parallel-usage and alias checking portions of the compiler. As such, the currently preferred mechanism for dynamic process creation is the ‘FORK’, described in the following section.

5.3 FORKs and FORKING

The FORK is a mechanism for dynamically creating parallel processes at run-time. The processes that are *spawned* run in parallel with the process that created them. Synchronisation of FORKed processes with the environment is controlled through a FORKING block, through which no FORKed processes may run.

Initial ideas for the FORK were to allow an arbitrary occam process to be spawned. This, however, introduces a lot of complication into the compiler. For the current implementation, a restricted form of FORK is available: that of a PROC call. Arbitrary FORKing of any occam process has not been ruled out completely however, and is discussed in section 8.3.1.

The motivation for providing FORK was largely to allow the construction of scalable server networks. The desire for such networks is entirely natural — and almost obvious. A typical example might be a TCP/IP server, for some protocol (HTTP, FTP, ...), as shown in figure 5.8. Initially the program would just wait for incoming connections. As each connection is received, the program could ‘fork’ off a server process to handle that particular client.

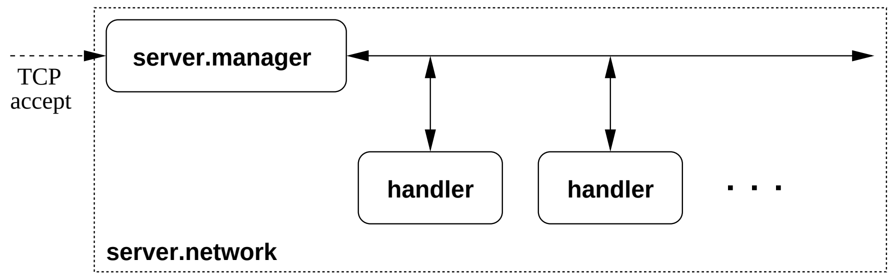


Figure 5.8: Forked scalable server network

In order to construct such networks, some use needs to be made of *shared* channel-types, as described in section 4.3. When the ‘server.manager’ process FORKs a new ‘handler’, it simply passes a CLONE of the relevant shared channel-bundle end.

This mechanism is substantially more flexible than either parallel recursion or n-replication, which can be used to provide comparable functionality. Recursion alone can implement such unbounded server networks, but at the cost of a continually increasing memory footprint, as the code recurses to handle each new connection. The n-replicated PAR can provide a run-time sized server network, but that size is still fixed at run-time. At best, a combination of the two can be used. However, such implementations are often not obvious (to code), and are easily prone to error if mis-programmed.

The FORK provides a clean solution to these problems, by providing a scalable mechanism for run-time process creation. The implementation allocates and frees memory as necessary — memory is allocated when a new process is FORKed, and freed when the FORKed process terminates. The only constant overhead is that of the process controlling the network, containing the FORKING block in which new processes are FORKed.

The syntax of the FORK is simple — simply the keyword 'FORK' followed by a PROC name and actual parameters. The handling of parameters is somewhat special, discussed in section 5.3.1. A simple "hello world" program using FORKs might be:

```
PROC hello (SHARED CHAN BYTE out!)
  VAL []BYTE str IS "Hello, FORKed world!*n":
  CLAIM out!
    SEQ i = 0 FOR SIZE str
      out ! str[i]
:

PROC fork.example (SHARED CHAN BYTE screen!)
  FORKING
    FORK hello (screen!)
:
```

This example is particularly uninteresting, since the only process contained within the 'FORKING' block is the single 'FORK' of 'hello'. The use of a SHARED channel (end) is required, although no parallel usage ever occurs in this code.

5.3.1 FORK Parameter Passing

By definition [GRS93, GRS94], PROC parameters in occam follow a renaming semantics — most obviously when an INLINE PROC has its parameters turned into abbreviations when inlined by the compiler. For FORKed PROCs, a communication semantics is used to handle parameter passing. In a normal PROC call, the invoking process transfers control into the called procedure and continues when that procedure returns. FORKed procedure calls are different however — they appear to return immediately. Because of this, any parameters passed to the FORKed process could potentially be interfered with in code following the FORK. Communication semantics for the parameters fit naturally in the general semantics for the FORK, discussed in section 5.3.2. Equivalently, such parameter passing could also follow an assignment semantics — with INITIAL declarations [Moo00b] for example. The main point being that a new set of variables are allocated to be (VAL) parameters to the FORKed PROC — in ordinary PROC calls, any VAL parameters larger than a single word are passed by reference (similar to non-VAL parameters).

For parameter types that cannot be assigned or communicated normally in occam, most notably reference parameters (including some RESULT parameters), special rules apply: the parameter must be explicitly #PRAMGA SHARED. Although this would appear to restrict the usefulness of a FORK, it does not: the use of reference parameters for a FORK is considered harmful and discouraged (aliases are created). The alternative is to use MOBILE types, covered in Chapter 4. These have a movement semantics in communication and assignment, implemented by moving references, making them ideal for use with the FORK.

For mobile channel-types, *unshared* ends are *moved* into the `FORK`ed process, whilst *shared* ends are auto-`CLONE`d. Special provision is made to support the passing of a (shared) anonymous channel-type end, that provides shared channels (section 4.4.1). In this special case (where such ends cannot be communicated or assigned normally), checks are made to ensure the end passed does not go out of scope before the `FORK`ed process finishes.

5.3.2 Semantics of FORK

As with all the other mechanisms for dynamic process creation, the `FORK` can be modelled in CSP/$\mathcal{CSPP}$, providing for formal reasoning about such code. In particular, several equivalences between `FORKING`/`FORK` processes and ordinary occam processes exist — the ability to prove these correct inspires greater levels of confidence for the usage of such extensions.

The `FORKING` block, that provides a barrier for `FORK`ed processes indented under it to synchronise on, can be modelled as a recursive process. This provides the mechanism for generating and synchronising `FORK`ed processes. The process representing the contents of the `FORKING` block is run in parallel with this recursive process. These two processes synchronise on a termination event, t, and a set of channels, $E = \{e_1, e_2, \cdots, e_n\}$, associated with the `FORK`ed processes $Q = \{Q_1, Q_2, \cdots, Q_n\}$. The `FORKING` mechanism, rather generalised, can be expressed as:

$$FORKING(t, E, Q) = \mu\, F \bullet \big(e_1?x \rightarrow (Q_1(x) \parallel F)\big) \,\Box \cdots \Box\, \big(e_n?y \rightarrow (Q_n(y) \parallel F)\big) \,\Box\, (t? \rightarrow Skip)$$

To `FORK` a process Q_x, the body of the `FORKING`, P', simply synchronises on the event e_x. It is assumed that on such synchronisations, the parameters (if any), present in the occam program, are communicated[6]. After the process P' has terminated, a synchronisation is made on t to terminate the `FORKING` process, simply:

$$P' \mathbin{;} t! \rightarrow Skip$$

Composed in parallel, with t and E hidden, this provides a full semantics for the occam `FORKING` block. The `FORK`ed processes $Q_1 \cdots Q_n$ are literally substituted for the processes they represent (occam `PROC` instances). The process P' is formed from the occam process indented under the `FORKING` block (P), by replacing "`FORK Qi (`x`)`" with "`e[i] !` x". Thus:

$$\big(FORKING(t, E, Q) \underset{\{t\}\cup E}{\parallel} (P' \mathbin{;} t! \rightarrow Skip)\big) \setminus \{t\} \cup E$$

Such a process interacts with its environment through the events of $Q_1 \cdots Q_n$ and P. These processes may also synchronise on events shared between themselves — with the necessary care being taken to avoid internal deadlocks.

Using this definition of the `FORKING` block, it is easy to verify the correctness of certain code equivalences. It is known, for example, that the following two occam code fragments are equivalent:

```
FORKING                                    PAR
  SEQ                                        R (x)
    FORK R (x)                               S (x)
    S (x)
```

The body of the `FORKING` block in this example can be expressed in CSP as: $P' = e_1!x \rightarrow \mathtt{S}(x)$,

[6] Parameters which do not fit the model of parameter passing by communication, remain free — to be captured by the environment later. Clearly, such 'parameters' must also be free in the body of a `FORKING` — P'.

with forked processes $Q = \{Q_1 = \mathtt{R}\}$. The `PAR` on the right is simply $(\mathtt{R}(x) \parallel \mathtt{S}(x))$. Substituting P' and Q into the earlier `FORKING` definition gives the process:

$$\big((\mu F \bullet (e_1?y \rightarrow (\mathtt{R}(y) \parallel F)) \,\square\, (t? \rightarrow \mathit{Skip})) \underset{\{t\}\cup E}{\Big\|} (e_1!x \rightarrow \mathtt{S}(x)) \,;\, t! \rightarrow \mathit{Skip}\big) \setminus \{t\} \cup E$$

Where E is the set of events $\{e_1.z \mid z \in \alpha p(\mathtt{R})\}$ — i.e. possible parameters for '`R`'.

The initial behaviour of this process is deterministic — the internal parallel processes can only engage in the communication on e_1. Furthermore, the events of this communication are hidden from the external environment (and assumed not to be free in either '`R`' or '`S`'). By expanding out the recursion one step and removing the initial e_1 communication, this process becomes:

$$\big(\mathtt{R}(x) \parallel (\mu F \bullet (e_1?y \rightarrow (\mathtt{Q}(y) \parallel F)) \,\square\, (t? \rightarrow \mathit{Skip})) \underset{\{t\}\cup E}{\Big\|} \mathtt{S}(x) \,;\, t! \rightarrow \mathit{Skip}\big) \setminus \{t\} \cup E$$

From this, the communication $e_1?y$ is no longer a choice for the recursive process — no other process engages in the corresponding $e_1!v$, and those events are hidden from the external environment. Simplifying to:

$$\big(\mathtt{R}(x) \parallel (\mu F \bullet (t? \rightarrow \mathit{Skip}) \underset{\{t\}\cup E}{\Big\|} \mathtt{S}(x) \,;\, t! \rightarrow \mathit{Skip}\big) \setminus \{t\} \cup E$$

The recursion is now unnecessary (F is no longer referenced inside the recursive process). This simplifies to:

$$\big(\mathtt{R}(x) \,\Big\|\, (t? \rightarrow \mathit{Skip}) \underset{\{t\}\cup E}{\Big\|} \mathtt{S}(x) \,;\, t! \rightarrow \mathit{Skip}\big) \setminus \{t\} \cup E$$

Eliminating t and E entirely gives:

$$\mathtt{R}(x) \,\Big\|\, \mathit{Skip} \,\Big\|\, \mathtt{S}(x) \,;\, \mathit{Skip}$$

This further simplifies to $(\mathtt{R}(x) \parallel \mathtt{S}(x))$, as desired. Proofs of other equivalences follow in similar ways. For example, between:

```
FORKING
  SEQ
    Q (X)
    FORK R (X)
    S (X)
```

```
SEQ
  Q (X)
  PAR
    R (X)
    S (X)
```

Although it is somewhat crude, this model of '`FORKING`' and '`FORK`' should suffice for both human and machine proofs.

5.3.3 Unsynchronised `FORK`ing

As introduced so far, all '`FORK`'s have been within a local '`FORKING`' block — in the initial implementation this was its limit.

However, it may be desirable to `FORK` a process from within a `FORKING` block at a higher level — in the calling `PROC` for instance. A very simple example of this is:

```
PROC other (VAL REAL64 v)
  FORK long.computation (v)
:

PROC main ()
  FORKING
    other (99.0)
:
```

When the ‘`other`’ `PROC` is called, from ‘`main`’, it forks the ‘`long.computation`’ process. This process is `FORK`ed within the higher-level `FORKING`, and therefore must have terminated before that `FORKING` block (in ‘`main`’) can complete.

The handling of this in the implementation is relatively simple — an extra hidden parameter is passed to `PROC`s that `FORK` processes outside of any local `FORKING` blocks. That hidden parameter represents the barrier on which finished `FORK`ed processes synchronise (covered in section 5.3.4). The restriction, of course, is that only communicable parameters can be given to the `FORK`ed process, since it may be impossible to tell if reference parameters are declared outside of the ultimately enclosing `FORKING`.

Global Servers

One feature which has often been desired is that of automatically terminating (typically server) processes. For example, when an application initialises, it might create a number of server processes designed to remain active for the duration of the whole program. For the application to terminate gracefully, such servers must be shut-down before the application can exit (to avoid deadlocking). However, engineering this into the application can often be hard, leading to its own deadlock if programmed incorrectly [Wel89].

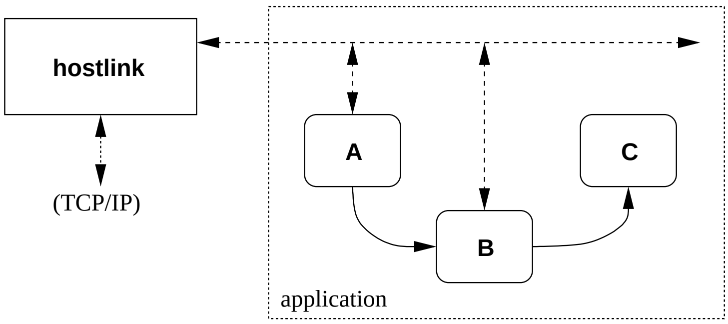


Figure 5.9: Example application network using a global ‘long-lived’ server

Figure 5.9 shows an example network using such a server. In this example, where the server process (‘`hostlink`’) involves itself in external IO (supported mainly through the blocking system-calls extension – section 6.3), the application will never terminate naturally through deadlock. As long as any processes remain on either the run-queue, timer-queue or are blocked on external IO, the run-time system will not deadlock.

Usually the best an application can do is execute ‘`STOP`’ at some suitable point, to force termination. However, this is defeated in ‘stop’ error-mode, where erroring processes are descheduled without stopping the whole system (described in section 7.3).

A solution to this is available using unsynchronised `FORK`s, specifically those that are outside *any* `FORKING` block. By default, and if the top-level process requires it, a system-wide barrier is passed to the application. Before the application terminates, it synchronises on this barrier. Thus, any 'global' servers, such as '`hostlink`' in figure 5.9, must terminate before the application can exit.

An alternative behaviour can be selected through the use of a compile-time flag ('`--nfw`' to the translator), that prevents the application synchronising on any global barrier (passed as a hidden parameter to the top-level process). When the top-level process finishes, assuming no processes are on the run-queue, timer-queue or waiting for external IO, the system simply exits — any `FORK`ed processes are simply destroyed. This does not solve the problem entirely for '`hostlink`', since it may be blocked waiting for external (socket) IO. Thus, a further compile-time option is available, '`--new`', that prevents the run-time system from waiting for pending blocking system-calls (section 6.3). With both options, when the application in figure 5.9 terminates, the '`hostlink`' process will be destroyed and the program exited.

5.3.4 Implementing `FORK`

The implementation of `FORKING` and `FORK` is reasonably simple in practice, sharing some common code with the implementations of the n-replicated `PAR` (section 5.2.1) and of recursion (section 5.1.1).

The `FORKING` construct introduces a new name, '`$fork.barrier`', of the *internal* type '`BARRIER`' — distinct from the user-defined `BARRIER` type [WW97], but with many similarities (i.e. it is a barrier). This barrier is somewhat optimised since only one process ever synchronises with it — the process executing the `FORKING`. This allows some of the barrier state to be omitted, saving a bit on memory — only two words are required, as shown in figure 5.10.

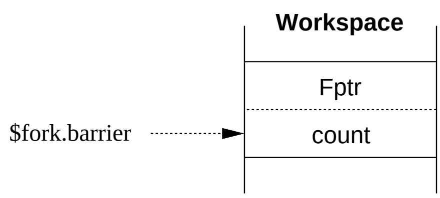


Figure 5.10: Workspace layout for the `FORK` '`BARRIER`'

When the `FORKING` block is entered, the barrier comes into scope and has its '*Fptr*' and '*count*' slots initialised to 'NotProcess' and zero respectively. The '*Fptr*' slot is only used when the `FORKING` block terminates; '*count*' records the number of `FORK`ed processes enrolled on the barrier.

The implementation of the `FORK` itself mostly follows that of a standard `PROC` call, but with additional code placed before and after the call, plus special handling for parameters that require it — described below.

When a `FORK` starts, memories for the `FORK`ed process are allocated from the free-lists. The workspace size allocated for the new process is adjusted to provide the `FORK` with some extra housekeeping information, kept above the new process's parameters. If vectorspace is required, it is allocated and stored in a local temporary. Mobilespace is somewhat more complicated since it cannot be returned to a global free-list, covered in section 5.3.5. Once allocated, the new process workspace is initialised, as shown in figure 5.11.

The initialisation of parameters, described in section 5.3.1, is similar to recursion in the implementation — largely because parameters need to be placed in a different workspace. Forked processes have the additional requirement that `VAL` parameters larger than 1 word (passed using a

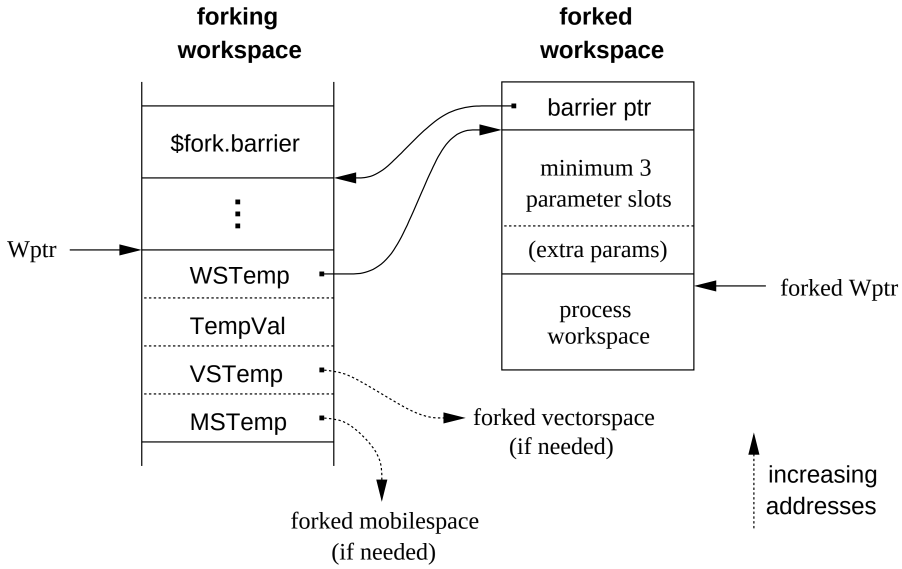


Figure 5.11: Setting up a `FORK`ed process

pointer) must be copied into newly allocated blocks before passing — i.e. a communication/copy semantics. The allocation and copy of these happen as the parameters are initialised. When the `FORK`ed process returns, these are simply freed (through the process's parameter slots).

After initialisation, the '*count*' field in the `FORKING` barrier is incremented — to enroll the new process — before adding that new process to the run-queue. The entry-point for the new process is set to a small bit of stub-code, placed in-line, that simply calls the correct routine (using the '`GCALL`' instruction). The workspace is suitably adjusted before calling '`STARTP`' to enqueue the process, since `GCALL` does not adjust the workspace pointer like `CALL` does.

Placed after the call is code that cleans up the `FORK`ed process. This only involves the deallocation of the various memories; workspace, vectorspace, mobilespace and any allocated parameters. Workspace is freed last, using the '`MRELEASEP`' instruction (described in appendix B.1.1). This is a combination of both '`MRELEASE`' and '`STOPP`', needed to prevent race-hazards when freeing its own workspace.

Prior to releasing its own workspace, the `FORK`ed process resigns from the barrier — whose location is stored as a pointer at the top of the forked process's workspace. To resign, the process simply decrements the '*count*' field within the barrier. If it reaches zero, the '*Fptr*' slot is checked, and if non-null, the process there is added to the run-queue. The '`Fptr`' slot will only ever contain one process, the process performing the `FORKING`, and only if it has finished.

Correspondingly, before the `FORKING` block exits, the '*count*' field is checked. If non-zero, the forking process suspends itself in the '*Fptr*' slot, since `FORK`ed process must still be running.

The code generated by the implementation (the translator in particular) is currently non-SMP safe. However, an SMP implementation is relatively straightforward, given the simplicity of this barrier — i.e. only one process ever synchronises, the others simply enroll and resign as necessary.

Implementing Non-Locally Synchronised Forks

As described in section 5.3.3, `FORK`s need not be enclosed in a local `FORKING` block. When a `PROC` of this nature is found, the compiler adds '`$fork.barrier`' as a hidden (pointer) parameter, in a similar way to the hidden vectorspace and mobilespace pointers. Calls to such `PROC`s may also trigger the addition of the hidden parameter, if they are not made from within a local `FORKING` block.

5.3.5 Mobilespace for `FORK`ed Processes

Mobilespace presents a slight problem because it cannot be released back to the main free-lists — for the same reasons as n-replicated `PAR` mobilespace: pointers may have been exchanged with other mobilespaces. Instead of holding mobilespaces in an array, they are linked together on a free-list (initially empty) inside the `FORKING` process's mobilespace, shown in figure 5.12.

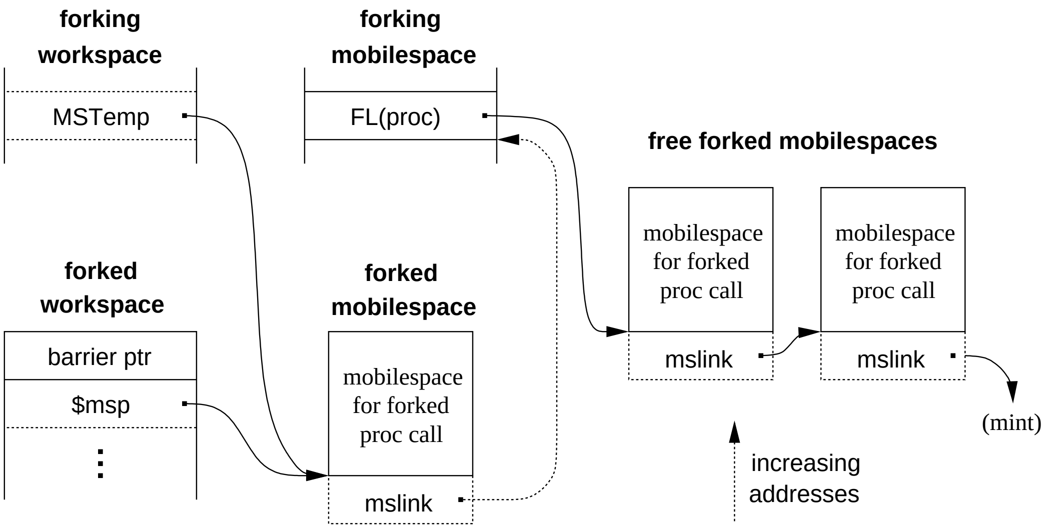


Figure 5.12: Mobilespace for `FORK`ed processes

When the (mobilespace requiring) `FORK`ed call is setup, the free-list in mobilespace is checked for any existing blocks. If present, the first one is removed and placed in the local '*MSTemp*' workspace slot, to be later copied into the mobilespace parameter. The local mobilespace free-list is then 'shuffled along'. If the free-list is empty (set to '`MINT`' – most negative integer), a new block is allocated from the free-list and has its first word initialised to '`MINT`'. The standard `PROC` mobilespace initialisation code uses this '`MINT`' as a trigger to setup the internal structure of that mobilespace, as described in section 4.2.2.

The '*mslink*' field between mobilespaces is either allocated explicitly (for memory allocation using '`MNEW`' and '`MFREE`'), or can re-use the similar '*link*' field in all '`MALLOC`'d memory — used to return a block of memory to the free-lists. The dynamically allocated mobilespaces will never be returned to the system free-lists, so can be safely re-cycled. During the lifetime of the forked process, this mobilespace link field points back to the free-list within the `FORKING` process's mobilespace.

When the forked process completes, the mobilespace is re-attached to the relevant free-list, to be re-used by any future `FORK`s. The value already in the free-list is placed in the current '*mslink*', and will either be '`MINT`' or other free mobilespaces.

5.3.6 Forked Recursion

Some time after adding the `FORK`, a desire for `FORK`ed recursion arose. This happens to be particularly useful, especially in combination with unsynchronised forking, since it allows a process to freely replicate itself inside some `FORKING` environment. One use investigated by Jacobsen et al. [JWB02] was the creation of graph-like structures using forked processes and mobile channel-types.

The expressive power of such networks is potentially huge, especially if recursive channel-types are used, allowing the network to be both dynamically re-configurable and dynamically scalable.

Forked recursion directly is in practise just forking another process — if the `FORKING` is non local, the `PROC` making the forked recursive call need not wait for any of its new instances to terminate. Any recursive forks within a local `FORKING` will be bound by the `FORKING` block, since any recursive call must finish before it can.

The active communicating nature of these networks make them ideal for many purposes. So far, the main use of these has been to investigate tree-like structures, for storage and manipulation of data within the active tree. However, they could also be used to implement neural networks, and in an entirely natural way — multiply connected active communicating processes, with the ability to grow and re-structure in a controlled manner. Networks in the order of a million processes could easily be created, assuming sufficient memory (for both the processes and the channels linking them).

Part II

Wider Interaction and Accessibility

This part of the thesis is largely concerned with the interaction between occam/CSP programs and the external run-time environment. Chapter 6 presents various extensions that greatly improve the run-time facilities available to occam programs. These are largely KRoC/Linux specific, and would require significant work to port to other platforms.

Chapter 7 presents a number of remaining, somewhat independent, extensions. These include, for example, support for process priority and a mechanism for concurrently running C and occam processes.

This thesis concludes in Chapter 8, with a discussion of the current state of this research generally, and many new future possibilities.

Chapter 6

CCSP and Linux

This chapter describes the various extensions to the occam run-time kernel, CCSP, that allow occam programs to take advantage of facilities provided by the underlying OS — Linux in this case. The reverse, improving Linux's support for occam programs, is also investigated.

Some of these extensions involve changes to the translator (tranx86), primarily for interfacing occam with the new features. Section 6.2 gives a general overview of the interface between occam programs and the run-time kernel, as generated by tranx86. This differs substantially from the interface used in the original KRoC/Linux (as generated by `tranpc`).

Sections 6.3 through 6.5 present the various extensions to CCSP (and tranx86), that vastly improve the run-time facilities available to occam programs: blocking system-calls, user-defined channels, and dynamically loadable processes. Section 6.6 discusses the addition of CPU-timer support for KRoC, that improves the resolution and accuracy of the occam `TIMER`.

Section 6.7 presents an extension to Linux that improves support and performance (in some cases quite considerably), for other extensions presented in this chapter.

6.1 Introduction

In the days of the Transputer, access to files and other OS functionality was provided over a Transputer-link connected to the host[1] — or with a suitable network interface for some host to access the Transputer (including booting).

Putting aside the details of the physical interconnect between host and Transputer, communication between the two uses a pair of channels carrying the '`SP`' protocol. To access the host (for example, often including screen output and keyboard input), occam programs used the '`hostio`' and '`hostsp`' libraries [SGS94], that provided the two '`SP`' channels as arguments. The server program on the host (usually '`iserver`') interpreted those requests and performed the appropriate action.

With the arrival of KRoC [WW96], a new method of interfacing with the environment was introduced — whilst still supporting the `hostio` and `hostsp` libraries. The KRoC interface to the environment consists of three `BYTE` channels, corresponding to the UNIX standard input, output and error streams. Wood [Woo98] added the ability to call C functions from occam, used to implement the `hostio` and `hostsp` support, as well as greatly enhancing the usefulness of KRoC— since it allows programmers to interface with arbitrary C code.

[1] Often, Transputers were to be found living on plug-in cards for existing hardware, with the appropriate interfacing to allow the host to access a Transputer-link.

The three BYTE channels (keyboard, screen and error) are largely sufficient for most simple applications, where the only interaction required is with the user at a terminal. The support for calling C functions allows a greater degree of interaction, but suffers from some fairly restrictive limitations. Most importantly, if an external C function blocks in the OS kernel, all occam processes are suspended, since KRoC schedules occam processes within a single OS-level process. This makes writing occam applications which use general OS services such as networking and IPC almost impossible.

The blocking system-calls extension (section 6.3) removes this restriction by providing a mechanism that allows an occam process to block in the Linux kernel, without suspending the scheduling of parallel occam processes. The interface to this is presented in a similar manner to external C calls [Woo98] — i.e. a mechanism for running arbitrary C code (which may block) is provided.

Both external blocking and non-blocking C calls are done through a PROC interface in occam. The user-defined channels extension (section 6.4) allows arbitrary C code to be placed behind channel operations (which may additionally be blocking). This allows for a greater level of interaction with external code — supporting ALTing on external events directly.

In a good number of large applications (and increasingly smaller applications), run-time loadable modules are used as a way of dynamically extending functionality. Generally, this is achieved either (i) through the use of the system's dynamic linker ('ld' in Linux), or (ii) in some application-dependent way. This functionality has been added to KRoC/Linux using the first method, including support for suspending, migrating, and resuming these dynamically loaded processes. This extension is described in section 6.5.

This chapter concludes with a discussion of a Linux device-driver designed to enhance support for occam/CSP programs, within in the KRoC/Linux environment (section 6.7). Particular attention is given to enhancing the blocking system-call operations, that incur a relatively high overhead.

6.2 Interfacing With occam

The original KRoC/Linux, that used tranpc [Poo96] as the translator from extended *virtual-Transputer* byte-code (ETC) [Poo98] to Intel i386 object-code, used the processor 'call' and 'ret' instructions (using the stack to pass parameters and results to and from the run-time kernel). The CCSP [Moo99] run-time kernel provided entry-points in C, using an indirect jump-table and blocks of in-line assembler to handle the occam process-state loading and restoring, as well as parameter and result passing. This interface is documented extensively in [Moo00b].

The occam program state, as far as the scheduler/run-time kernel is concerned, consists of those CPU registers which hold meaningful values in the translated occam program. The original Transputer process state consisted of the (32-bit) words shown in table 6.1. We ignore the 'Iptr' register mostly, since it only really makes sense to map that onto the target machine's corresponding register ('EIP' on the i386).

Name	**Description**
Wptr	workspace pointer (process descriptor)
Iptr	instruction pointer
Fptr	first process in the run-queue
Bptr	last process in the run-queue
Tptr	timeout queue

Table 6.1: Transputer process state

On the majority of KRoC targets, which are RISC[2] machines, these words can be mapped directly onto target processor registers. On a CISC[3] architecture, such as the i386, the number of available processor registers is limited — to 8 general-purpose integer registers on the i386, shown in table 6.2. Given this small number of registers, mapping the whole Transputer state into processor registers is not really possible[4].

Name	Description
EAX	general purpose (accumulator)
EBX	general purpose (base)
ECX	general purpose (count)
EDX	general purpose (data)
ESP	stack-pointer
EBP	base-pointer (frame-pointer)
ESI	source index (for block copy)
EDI	destination index (for block copy)

Table 6.2: Intel i386 general-purpose integer registers

6.2.1 Target Register Mapping

The original KRoC/Linux (using `tranpc` to generate code) mapped 'Wptr' onto the 'EBP' register, with 'Fptr', 'Bptr' and 'Tptr' as variables in CCSP — compiled occam code only ever needs to manipulate 'Wptr' directly. The 'ESI' register was used to hold a pointer to the table of run-time kernel entry-points, leaving 'EDI' free and 'ESP' for the existing C stack-pointer. The four general-purpose registers, 'EAX', 'EBX', 'ECX' and 'EDX' were used for computation (dynamically mapped onto the Transputer integer stack).

This register mapping has changed slightly with the introduction of tranx86, due in part to the removal of the indirect jump-table for entering the run-time kernel. The four general-purpose registers ('EAX', 'EBX', 'ECX' and 'EDX') are still used for computation (mapped onto the Transputer integer stack); 'EBP' is still used for 'Wptr', and 'ESP' remains used for the C stack-pointer. However, 'ESI' and 'EDI' are now used to hold the 'Fptr' and 'Bptr' registers respectively, significantly lowering the overheads for some translator-inlined scheduler operations. This is summarised in table 6.3.

Inmos T800	Intel i386	Description
Iptr	EIP	Instruction-pointer
Fptr	ESI	Run-queue front pointer
Bptr	EDI	Run-queue back pointer
Tptr	*var*	Timer queue

Table 6.3: Transputer register mapping on the Intel i386

[2]Reduced Instruction-Set Computing — typically simple instructions and many registers.
[3]Complex Instruction-Set Computing — typically complex instructions and few registers.
[4]Technically speaking, mapping the whole Transputer state into processor registers on the i386 is possible, since only 4 registers are needed for computation — leaving 4 for the state. However, for efficiency reasons, we choose not to do this.

6.2.2 Run-time Kernel Calling

The transfer of control between `occam` programs and the CCSP run-time kernel, when translated by `tranpc`, involved using the C stack (of CCSP) to pass values, before '`call`'ing an address in a lookup-table. Returned values were also passed on the C stack (although these are less common).

The code generated by `tranpc` to call the run-time kernel involved pushing the contents of the Transputer stack onto the C stack, then calling an entry-point. The entry-point address was obtained from a constant offset in a lookup table, whose address was held in a register. The CCSP side of this interface was significantly less pleasant however, requiring some relatively non-trivial inline assembler. This interface was required largely due to the use of `tranpc` for other targets [Poo96].

The interface assembly code, within CCSP, loaded the workspace-pointer from the '`EBP`' register, then popped the arguments off the C stack into general-purpose registers. '`gcc`' was then given a (static) mapping between the general-purpose registers used and ordinary C variables — a fairly `gcc` specific technique called "register constraints" and described in [Sta98]. Figure 6.1 shows this calling sequence diagrammatically.

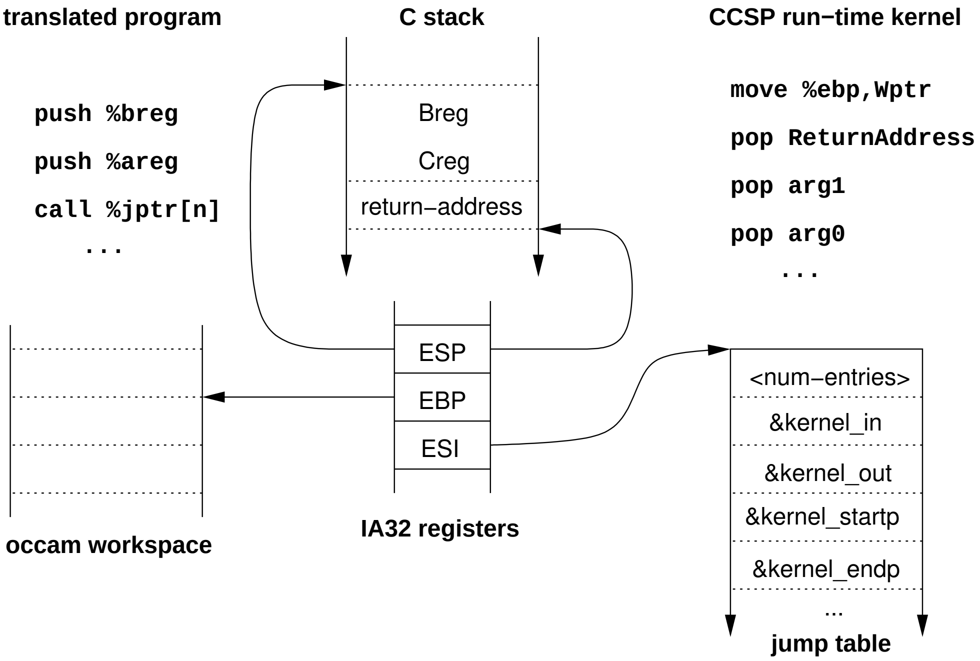


Figure 6.1: Original calling sequence between `tranpc` and CCSP

With the introduction of `tranx86`, the opportunity to re-implement the `occam`/CCSP interface arose. The main motivation for this was the high overhead of a run-time kernel call — partly associated with pushing and popping arguments from the C stack. Other motivations were that a processor register was constantly in use during `occam` execution (the pointer to the entry-point lookup table). Additionally, the '`call`' instruction used needlessly pushed the return address onto the stack for some instructions (in the case of '`ENDP`' for example).

An ideal interface would be for `tranx86` to arrange for the arguments to be in particular processor registers at the point of the kernel call, and for the assembler in the run-time kernel to map these registers directly onto C variables. In many cases, this is possible. In some other cases, however, `gcc` rejects the inline assembler register-variable mappings due to incompatibilities with its own generated code.

In theory, gcc should completely work-around user-specified assembler constraints, but in reality the code-generation sometimes gets too complex and the constraints cannot be satisfied easily. Rather than searching aggressively for solutions, it often just gives up. To make matters even more complicated, different versions of gcc can behave substantially differently where inline assembler (in particular register constraints) are concerned.

The current code in CCSP makes attempts to work in all cases, aided by various configure-time checks in the source distribution, such as to avoid trying to compile code which will result in compiler errors[5].

In order to provide for the differing requirements of different kernel calls, and the need to be compatible with differing gcc versions, a variety of kernel-call interfaces are supported. All of the new calling conventions enter the run-time kernel by means of a name which is resolved at link-time. This has a small penalty in terms of code-size, but frees up the previously used 'ESI' register — that is now used to hold 'Fptr'.

Whenever values are returned from run-time kernel calls, they are passed using the C stack. This is because gcc (in most common versions to date) fails to satisfy the constraints used in the inline assembler required to return values in processor registers. For passing arguments into the run-time kernel, both methods (passed in registers or passed on the C stack) are supported.

Passing by register is the preferred method, but for some kernel calls gcc fails to satisfy the assembler constraints. Efforts have been made to optimise the most common run-time kernel calls (such as communication and process startup/shutdown instructions). The translator will optionally inline certain instructions, completely avoiding any kernel call in some cases — scheduler-inlining will inline the 'RUNP' (run-process) and 'STARTP' (start-process) instructions for example. Details on (some) of the inlining performed by tranx86 can be found in [Bar01].

Name	**Method**
call	regular call (i386 call instruction), return-address is left on the C stack
storeip-jump	return-address is placed at 'Wptr[-1]' followed by a jump to the entry-point
regip-jump	return-address is placed in a register, followed by a jump to the entry-point
jump	the entry-point is jumped to directly (return-address discarded)

Table 6.4: Methods used to transfer control from occam to the CCSP run-time kernel

Four calling conventions are provided for transfer of control from occam to CCSP, shown in table 6.4. The 'call' method is used when CCSP needs to have the return-address in a variable, but cannot constrain it into a register. On the CCSP side, the return-address is popped off the C stack into a C variable. When the return address can be constrained, the 'regip-jump' calling sequence is used.

The 'storeip-jump' and plain 'jump' are used when CCSP does not need to know the return address in occam explicitly, either because it would have only been placed in the 'Iptr' workspace slot (for most descheduling instructions), or because it is not needed at all. Figure 6.2 shows an example of the 'storeip-jump' kernel calling mechanism, for the '_Y_in32' entry-point, which takes two arguments from the virtual-transputer stack.

[5]The first source-release of KRoC/Linux (containing a binary of tranpc) suffered particularly from these problems. Additionally, certain Linux vendors shipped distributions with broken gcc compilers, which failed to compile some inline assembly. Not the fault of the gcc developers in any way however, the broken versions were known to be broken, but those Linux vendors shipped them anyway, much to the disliking of the Linux and gcc communities.

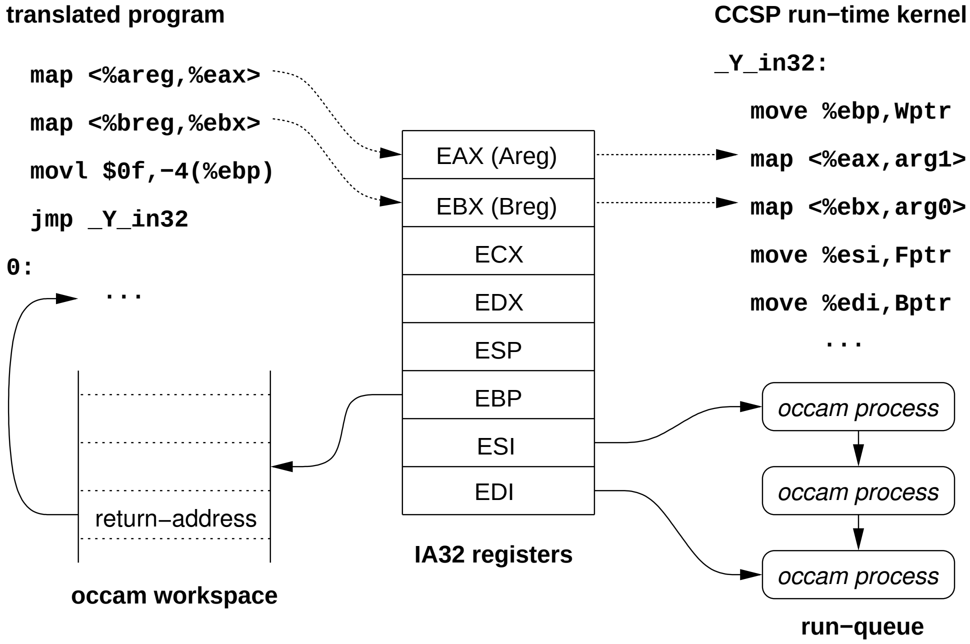


Figure 6.2: 'storeip-jump' calling sequence between `tranx86` and CCSP

The CCSP run-time kernel has been subjected to a large number of changes over the past three years. One significant change was the removal of the jump-table (which was an array in C) and the breaking up of the previously monolithic '`kernel()`' function — which contained all the various entry-points. The modified CCSP has a separate C function for each kernel-call, making the source significantly more readable and lightening the load on `gcc`'s optimiser and register/stack allocator. The '`_Y_in32`' call, for example, has the following structure:

```
void kernel_Y_in32 (void)
{
    K_SETGLABEL_TWO_IN (Y_in32, channel_address,
                        destination_address);

    temp_ptr = *channel_address;
    if (temp_ptr == NotProcess_p) {
        ... channel not ready code
        K_JUMP (X_scheduler);
    } else {
        ... channel ready code
        RESTOREJRET;
    }
    barrier ();
}
```

The macros '`K_SETGLABEL_TWO_IN`', '`K_JUMP`' and '`RESTOREJRET`' are inline assembler blocks, kept slightly separate from the kernel code itself in the interests of portability. The first, '`K_SETGLABEL_TWO_IN`' defines an entry-point for `occam`, which will pass two arguments in general-purpose registers. This macro is implemented (for the i386 architecture) as:

```
#define K_SETGLABEL_TWO_IN(X,A,B)
    __asm__ __volatile__ ("               \n"
        ".globl _"#X"                     \n"
        "       _"#X":                    \n"
        "       "#X":                     \n"
        "       movl    %%ebp, %0         \n"
        "       movl    %%esi, %1         \n"
        "       movl    %%edi, %2         \n"
        : "=m" (Wptr), "=m" (Fptr), "=m" (Bptr),
          "=b" (B), "=a" (A))
```

The first three lines of assembler define the entry-points. The externally visible name (when '`Y_in32`' is substituted for the macro variable '`X`') is '`_Y_in32`'. The internal entry-point (which only used in certain calls) is '`Y_in32`'.

The following three lines transfer the contents of the '`EBP`', '`ESI`' and '`EDI`' registers into the C variables '`Wptr`', '`Fptr`' and '`Bptr`' respectively. Within the assembler, the C variables are referred to by *constraint operands*, which are mapped to the actual variables following the assembler code (for example, '`"=m" (Wptr)`' maps the output memory-reference of '`Wptr`' onto '`%0`'). The mapping of the passed arguments to the '`A`' and '`B`' macro parameters is done in the second line of constraints, and requires no explicit assembler at all.[6]

The second macro used in the example C code above, '`K_JUMP`', is trivial — it simply performs an explicit jump to a named internal entry-point. In the majority of cases, the target is '`X_SCHEDULER`', that schedules an `occam` process for execution, as well as checking the synchronisation flags for activity. The final macro in the example, '`RESTOREJRET`', is used to return to the `occam` program. This particular variant restores '`Wptr`', '`Fptr`' and '`Bptr`' to processor registers, then jumps to the address stored at '`Wptr[Iptr]`' (which would probably have been placed there by a 'storeip-jump' style kernel-entry from `occam` — table 6.4). This macro is implemented (for the i386 architecture) as:

```
#define RESTOREJRET \
    __asm__ __volatile__ (
        "       movl    %0, %%ebp         \n"
        "       movl    %1, %%esi         \n"
        "       movl    %2, %%edi         \n"
        "       jmp     *-4(%%ebp)        \n"
        : /* no outputs */
        : "g" (Wptr), "g" (Fptr), "g" (Bptr)
        : "memory", "cc")
```

In a similar manner to before, the C variables are referred to by operand constraints, but mapped as input operands rather than output operands. The additional third set of constraints specifies what gets 'trashed' by the assembler. In this case it is 'memory' (variables may be affected) and 'cc' (the condition-codes) — this may seem obvious, but `gcc` cannot deduce that this code never returns (it executes an unconditional jump, '`jmp`').

The peculiar '`barrier()`' call at the end of the C function is an assembler macro that does nothing, but trashes 'memory' and 'cc'. This is required to work around a minor bug in some faulty `gcc` versions, that do not wholly respect the '`-fno-defer-pop`' compiler flag. It may be noted

[6] If `gcc` subsequently wishes to use, say `EAX`, for some other purpose, it will insert an instruction to move the contents of the `EAX` register into the variable mapped to it.

that none of the assembler code involved in getting in and out of the run-time kernel references the stack-pointer (‘ESP’) — the code involved relies on the fact that the stack-pointer on exit to occam is the same as it was on entry from occam. A slight bug in some gcc versions causes this assumption to break, when it incorrectly optimises stack-pointer manipulation (the incorrect part being that we explicitly tell the compiler *not* to perform this optimisation on the command-line — the ‘-fno-defer-pop’ flag). This would normally be a minor issue, since it would not affect C-only programs. The effect on occam programs was that each affected kernel call would consume a small bit of C stack when called, leading to large stack growth over time, and eventually a crash.

6.2.3 External Synchronisations

In order to respond to events outside occam, KRoC utilises UNIX (POSIX) signals. There are two such events in most KRoC implementations: a signal from the keyboard process to indicate a new key is ready (described in section 6.3.9), and a timeout signal for implementing occam timers. In both cases, the signal handler sets a (byte) flag in the synchronisation word. Whenever the run-time kernel reschedules, it checks the synchronisation word and, if any flags are set, handles them accordingly.

If the run-time kernel was sleeping (in the ‘safe_pause()’ function), the act of signalling will wake it up. Figure 6.3 shows the layout of the synchronisation word, including some new flags.

kbd_sync (keyboard process)	block_sync (blocking syscalls)
dummy0	priority_sync (process priority)
tim_sync (timeout handling)	
dummy1	interrupt_sync (for RMoX)

Figure 6.3: Run-time kernel synchronisation flags

The ‘kbd_sync’ flag is set by a signal handler when the keyboard process signals the occam kernel (section 6.3.9). At the same location is ‘block_sync’, which is set when the occam kernel is signalled by a finishing blocking-call (section 6.3.6).

The ‘priority_sync’ flag is used when process priority is enabled (the default now), to indicate that a higher priority process is runnable. Priority is covered in section 7.2. The timer synchronisation flag, ‘tim_sync’, is generally set by the signal handler for ‘SIGALRM’ — KRoC timeouts (on UNIX systems) are generally implemented using the ‘alarm()’ or ‘setitimer()’ system calls — POSIX defines the former, BSD the latter.

On the Intel Pentium and above processors, timers are implemented using the CPU cycle-counter (accessed through the ‘rdtsc’ instruction [Int99]) instead of using the underlying OS’s ‘alarm()’ mechanism. The main motivation for this was to reduce the high cost and low-granularity of the timeout mechanism — the standard Linux i386 kernel timeout resolution is 10ms.

On the DEC Alpha architecture (for which a KRoC/Linux does not exist), the Linux kernel has a significantly finer timing resolution of 1ms[7]. When CPU timers are enabled (determined by the processor type at compile-time), the ‘alarm’ mechanism is only used when sleeping in ‘safe_pause()’.

[7] The discussion about changing the magic ‘HZ’ Linux kernel constant for the i386 architecture occasionally appears on the mailing list. The general view, however, is that changing it will break too many applications (those which make assumptions about its value). Patches do exist however, and appear to work, so may well find their way into a future Linux version.

When scheduling occam processes normally, the run-time kernel uses the CPU cycle-counter to check for timeouts (an operation which can be performed sufficiently quickly so as not to add significantly to the scheduling overhead — both when process(es) are waiting on timeouts or when there are no waiting processes).

Additionally, as long as the run-time kernel is busy (there are runnable occam processes), '`alarm()`' signals are avoided, giving a much finer resolution — controlled by the parallel granularity of the occam program — more process scheduling leads to finer timeouts.

However, the timeout resolution reverts to 10ms whenever the kernel runs out of runnable processes and sleeps in '`safe_pause()`'. Achieving a finer timeout resolution is possible, but requires either user-space busy loops (a technique which MESH [BDv99a, BDv99b] uses), or substantial changes in the way sleeping is implemented — investigated in section 6.7.2.

The last synchronisation flag in figure 6.3, 'interrupt_sync' is currently used by the (experimental) RMoX system [BJV03] only. There, this flag is used to indicate that a *hardware-interrupt* occurred, needed to support low-level interrupt handling from occam.

6.3 Blocking System Calls

The blocking system-calls extension, first presented in [Bar00b], provides a mechanism for allowing the execution of a blocking system call (reading from a network socket for example), without blocking the run-time kernel (CCSP).

One of the immediate advantages of this is the ability to write network aware occam applications (typically internet servers — web servers, mail servers, etc.). These types of application often require the ability to handle multiple (remote) clients simultaneously, without effects such as the latency and bandwidth of one client affecting the processing of other connected clients. Section 6.3.1 describes one of the primary motivations for blocking system-calls — building internet server applications.

The use of blocking system-calls is not the only solution to this more general problem — there are other ways in which occam programs can interact with networked applications. This is clearly demonstrated by Vella in [Vel98], who used specific engineering in the run-time kernel to support networking. Blocking system-calls provide additional benefits however — they are not limited to the provision of networking, although this is a primary motivation for them.

The implementation of blocking system-calls occurs largely within CCSP, utilising Linux "clone" processes (primitive threads). Some support is also required in the translator (tranx86), to support a new run-time kernel entry-point for dispatching a blocking call. The occam interface for blocking system-calls follows that of external C calls [Woo98], with a simple name change to indicate that the called C code should execute *alongside* the occam system, not within it (as standard external C calls do). The occam interface for blocking system calls is described in section 6.3.2.

When a blocking system-call is made from an occam program, the run-time kernel arranges for it to be executed in one of the 'clone' (OS) processes, while the run-time kernel continues scheduling occam processes. When the blocking call finishes, the clone notifies the run-time kernel and the occam process which called it is rescheduled. As an additional feature, a form of blocking system call which may be terminated before it finishes is provided. The correct implementation of blocking system-calls is non-trivial, requiring care to avoid potential race-conditions between the clones and the run-time kernel.

Sections 6.3.4 to 6.3.7 describe the implementation of blocking system calls within the translator and run-time kernel. As well as directly supporting occam programs through the use of explicitly defined blocking calls, the generic blocking system-calls mechanism has applications for the *user-defined channels* extension (section 6.4).

The performance of blocking system-calls is examined briefly in section 6.3.8, but even a modest run-time cost is a small price to pay for such functionality — especially when it can be programmed easily in a naturally parallel manner.

On multi-processor machines, without a multiprocessor KRoC, blocking system-calls can be used to take advantage of extra processors, since the clone processes are free to be scheduled by Linux on any available processor — possibly concurrently with other blocking calls or with the run-time kernel.

6.3.1 Building Internet Applications

When internet server applications are written in traditional languages such as C, the handling of multiple clients is mostly done using a suitable IO multiplexing function — the '`select()`' and '`poll()`' functions in UNIX/POSIX. Such applications often end up as a two-stage operation which loops continuously: wait for all possible events then process those which happened — comparable to a fair-`ALT` in occam. However, programming in this manner correctly is often hard — most obviously because the parallel multi-client design of the system does not map cleanly onto a single-thread IO-multiplexed implementation[8].

The use of threads can be of help here, but for languages such as C, threads introduce a whole new set of potential problems — thread synchronisation, aliasing and parallel race-hazards. Whilst the tools for correct thread control exist ('`pthreads`' — POSIX threads [But97], a common thread API), no correct usage enforced.

Another way of tackling the multi-client problem is to avoid it in the first place, by having a program which handles one client only and a mechanism for starting a new client-handling program each time a new client connects. This is the approach taken by many types of server on UNIX, often started from a daemon such as '`inetd`' (the "internet super-server"). Unfortunately any server application that requires or facilitates interaction between simultaneously connected clients (a chat server for example) is reduced to interprocess communication (IPC) mechanisms to transfer data between clients, which leads to even more problems. Furthermore, the cost of starting a new process each time a new client connects may become prohibitive as the frequency of client connections increases.

The desire to be able to program multi-client server applications easily is overwhelming. Parallel languages such as occam provide the ideal abstraction (of processes and communication), which unfortunately is often limited by the implementation. Putting aside implementation issues, those familiar with the IRC (Internet Relay Chat) network might visualise the design of a server similar to that shown in figure 6.4.

Such a network design would require mechanisms for creating and destroying processes and connections, as clients and servers join and leave, and as clients (users) join and leave channels (that controls the creation and removal of channel processes).

Blocking system-calls provide part of the implementation required for building multi-client server applications, but do not provide mechanisms for dynamic network reconfiguration as needed to implement figure 6.4. Such an implementation is made possible, however, by incorporating the use of dynamic process creation (chapter 5) and mobile channel types (section 4.3). Without facilities for dynamic network creation and reconfiguration, servers such as these require a modest, but sometimes non-intuitive, change in design.

---

[8] The difficulty of programming single-thread server applications is demonstrated by the large number of USENET messages which often result in a standard reply of "`select()` is not broken.", followed by an explanation describing how to use `select()` correctly.

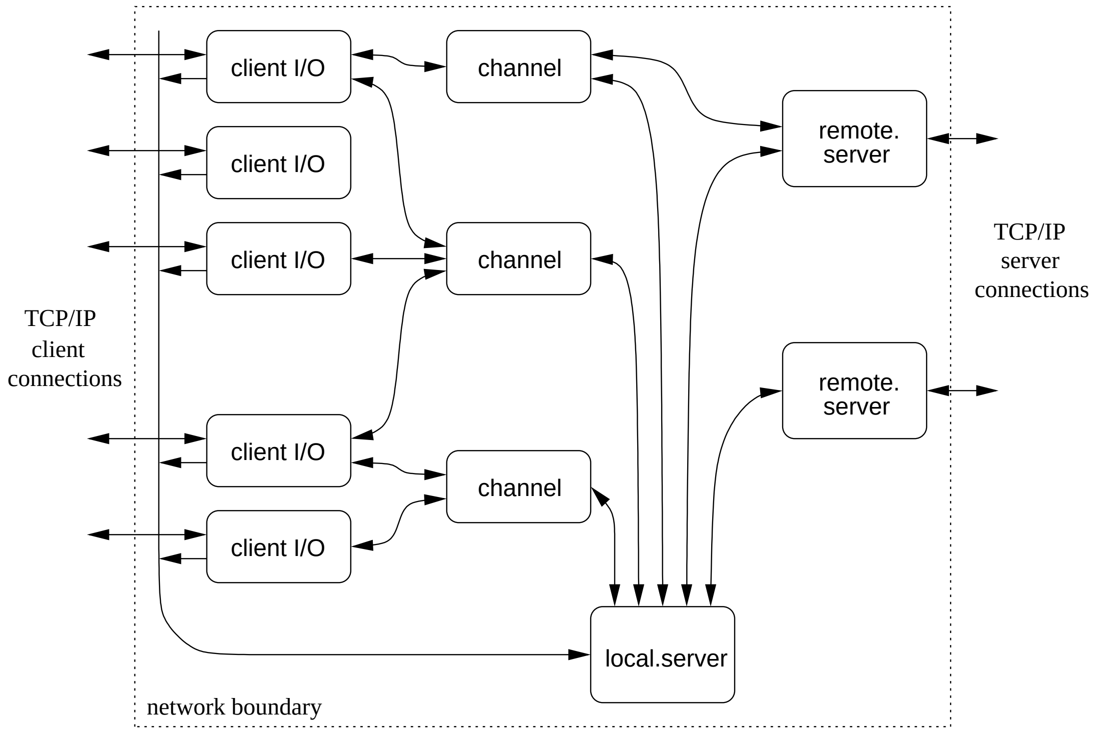


Figure 6.4: Conceptual process network for an IRC server

6.3.2 The occam Interface

The occam interface for programmer written blocking system-calls closely follows the convention used by Wood [Woo98] to implement external C calls. Technicalities associated with parameter passing between occam and C result in slightly hard-to-read C code. These difficulties are relatively minor however. For the purposes of example, an external call which reads a number of bytes from a file-descriptor (which could be a socket) can be implemented with:

```
static __inline__ void real_fd_read (int fd, char *buffer,
                                     int bufsize, int *result)
{
    *result = read (fd, buffer, bufsize);
}

/* PROC ?.fd.read (VAL INT fd, []BYTE buffer, INT result) */
void _fd_read (int *ws)
{
    real_fd_read ((int)(ws[0]), (char *)(ws[1]), (int)(ws[2]), (int *)(ws[3]));
}
```

occam code which wishes to call this as an external C call declares it with an external PROC definition:

```
#PRAGMA EXTERNAL "PROC C.fd.read (VAL INT fd, []BYTE buffer, INT result) = 0"
```

which can then be called in the same way as any other occam PROC.

From the programmer's point-of-view, modifying the code to perform the read as a blocking system-call is trivial, requiring only small changes in the declaration:

```
#PRAGMA EXTERNAL "PROC B.fd.read (VAL INT fd, []BYTE buffer, INT result) = 4"
```

The name change from '`C.fd.read`' to '`B.fd.read`' will require similar changes in any `PROC` that calls it. The change in the 'magic' number at the end of the declaration from 0 to 4 is a necessity. This number is the workspace requirement, in words, of the external `PROC`.

An external C call clearly needs no occam workspace, hence the zero. A blocking system-call requires a small amount to save the occam process state however — since the run-time kernel will continue scheduling occam threads whilst the process making the blocking call is blocked. The particular value 4 is influenced by the presence of process priority (described in section 7.2). If process priority is disabled, only 3 words of workspace are required (one less for the absent process-priority slot).

Since the correctness of this is somewhat critical, the compiler will check for the correct value, emitting a warning if not enough workspace is reserved — code is still generated and the correct amount of workspace is reserved however.

The occam interface for 'killable' blocking system-calls is a modification on this, requiring an extra parameter as well as a name change:

```
#PRAGMA EXTERNAL "PROC BX.fd.read (CHAN INT kill?, VAL INT fd,
                                  []BYTE buffer, INT result) = 4"
```

The additional parameter '`CHAN INT kill?`' is an abstraction only, it should not be used for direct communication. This should not cause a problem, since the scope of this channel ought be very local (given correct design).

A specific C function implemented in the run-time kernel is available to occam programs in order to terminate a blocking system-call. This is accessed using the declaration:

```
#PRAGMA EXTERNAL "PROC C.killcall (CHAN INT kill!, INT result) = 0"
```

Conceptually, this signals the termination of a blocking system-call on the special '`kill`' channel. After calling this `PROC`, the '`result`' parameter contains the termination status, shown in Table 6.5.

Result	**Description**
-1	the blocking call had already finished
0	the blocking call was successfully terminated
1	the blocking call had only just finished
2	the blocking call is currently finishing

Table 6.5: Blocking system-call '`killcall()`' results

The most common result for termination will be 0 — successful termination. Any other result indicates that the call had already finished — with the parameters ('`buffer`' and '`result`') to '`BX.fd.read`' being appropriately defined. The particular result values 1, 2 and -1 relate to the termination algorithm (algorithm 6.5), used to trigger early termination of the blocking call.

6.3.3 Safe Termination

It is worth giving consideration to application design when terminating blocking system-calls. Without careful programming, a system may easily deadlock as events associated with successful termination of the blocking call, and signalling the termination of the blocking system call interleave. For a typical server application, the event which triggers early termination of a blocking call might be a read timeout, or an explicit kill signal if a client is to be disconnected.

The occam fragment below shows a method for terminating the example '`BX.fd.read`' blocking call, triggered by an arbitrary '`termination event`':

```
-- assume 'VAL INT fd' and '[]BYTE buffer' are available
INT result:
BOOL did.kill:
SEQ
  CHAN INT c, signal:
  PAR
    --{{{  blocking call
    INT res:
    SEQ
      BX.fd.read (c, fd, buffer, res)
      signal ! res
    --}}}
    --{{{  collection/termination
    PRI ALT
      signal ? result
        did.kill := FALSE              -- normal finish

      ... termination event
        SEQ
          C.killcall (c, result)
          IF
            result <> 0
              SEQ
                signal ? result
                did.kill := FALSE      -- normal finish
            TRUE
              SEQ
                INT any:
                signal ? any
                did.kill := TRUE       -- terminated
    --}}}
```

After this code fragment has executed, '`did.kill`' indicates whether the call finished successfully or if it was terminated. In the `FALSE` (call finished) case, '`result`' will hold the result from '`BX.fd.read`'. In the `TRUE` case (call terminated), '`result`' will be zero (indicating that the call was terminated successfully — in the other cases, where termination races with completion of the blocking call, the call is considered to have completed successfully — which it did).

One of the main design motivations for killable blocking calls is for use in `ALT`s — as in the above example. To cater specifically for this, the blocking system-call library shipped with KRoC [Bar00a] provides specific `ALT`-able variants of the common blocking socket operations '`read`', '`write`', '`accept`' and '`recvfrom`'. The last of these is for UDP and TCP; the others are for TCP only.

These ALT-able variants provide a simpler interface for the occam programmer, hiding the details of direct termination using 'C.killcall'. This reduces the application's involvement to, for example:

```
CHAN BOOL kill:
CHAN INT response:
PAR
  --{{{  blocking call
  socket.altable.X (kill?, response!, ...)
  --}}}
  --{{{  wait for response or timeout
  PRI ALT
    --{{{  incoming response for normal termination
    INT any:
    response ? any
      kill ! TRUE
    --}}}
    --{{{  timeout (after 1 second)
    tim ? AFTER (t PLUS 1000000)
      INT k.result:
      SEQ
        kill ! TRUE
        response ? k.result
        ... take action on 'k.result' if necessary
    --}}}
  --}}}
```

This fragment demonstrates a killable blocking socket operation ('socket.altable.X') using the two channels 'kill' and 'response'. The design rule is that whatever happens, a single communication must occur on each channel. This ensures that 'socket.altable.X' will not deadlock, as it must too communicate once on each of the two channels. In the case where the blocking call terminates naturally, 'socket.altable.X' performs a parallel input/output, allowing the user to order the communications in whatever way is appropriate for the application.

This design is effectively a localised IO-PAR [WJW93] network, guaranteeing deadlock freedom. The particular application here solves a typical occam programming problem: two processes that must communicate either one way or the other. This would normally require the use of output guards in ALTs — that are not supported.

Cleaning Up After Termination

In some situations, the external C code being executed as the blocking call may need to perform some cleaning-up, after early termination from the occam world. Although this is an operation performed entirely in C, it is relevant to the occam programmer, and thus described here. Within the CCSP system, a function is provided that arranges for a function to be called upon (forced) termination of the blocking call:

```
void *bsyscalls_set_cleanup (void (*)(void *));
```

This function takes, as a parameter, the address (name) of a function that will be called when the blocking call is killed (with 'C.killcall'). The return value from this function is a pointer to memory which can be used to save information required for cleanup. The same pointer is passed to the function called when forced termination occurs, that can then use the previously stored data to perform the cleanup operation.

The pointer returned by 'bsyscalls_set_cleanup()' points at the bottom of the stack used by the active clone process (section 6.3.4). As such, it is the programmer's responsibility to ensure that data placed here will not be overwritten by the blocking call through stack usage — a maximum limit of 4 kilo-bytes is recommended.

One perceived, and investigated, use of this is when a blocking call starts another OS process (through 'fork()' or similar), and waits for it to finish. If this functionality were not used, the OS process started would not be killed when the blocking call is terminated. With this functionality, the blocking call can arrange for an OS process started to be killed on its own termination. The occam OS process library (part of the blocking syscalls library [Bar00a]) takes advantage of this, with a demonstration program which runs the program "ping 127.0.0.1" for 5 seconds before terminating it. This example program can be found in the KRoC/Linux source distribution [WMBW00].

6.3.4 Implementing Blocking System Calls

The implementation of blocking system-calls uses Linux 'clone()' processes. These are effectively 'threads' — sharing memory, file-descriptors and other file-system information (current working directory and 'umask' for example). When the occam program dispatches a blocking system-call, the external C function referenced is run inside one of these clone processes, allowing the occam run-time kernel process to continue scheduling occam processes. For obvious reasons, each clone process has its own stack, whose size is currently fixed[9].

Clones are created through the use of the Linux 'clone()' system-call, which takes arguments for the start-address of the new process, a pointer to its stack and various flags to control sharing. Once a clone has been started, communication with it is handled through shared variables, a (Linux kernel) semaphore, POSIX signals, and two spin-locks.

Placed at the bottom of each clone's stack is a structure holding the state of that particular clone:

```
struct _bsc_thread {
    int pid;                  // clone's process ID
    int thr_num;              // thread number [0..]
    int *ws_ptr;              // occam workspace pointer
    char *raddr;              // occam return address
    int *ws_arg;              // argument to func
    int arg3;                 // 3-argument flag
    int *ws_arg2, *ws_arg3;   // additional arguments (not used by occam)
    int pri;                  // priority of occam process
    void (*func)(int *);      // function to execute
    int adjustment;           // parameter offset
    sigjmp_buf *jbuf;         // jump buffer (allocated above this)
    int terminated;           // terminated ?
    int cancelled;            // cancelled ?
    void *user_ptr;           // pointer to 'spare' space
    void (*cleanup)(void *);  // cleanup function
};
```

[9]The stack for processes started by the kernel (using 'fork()') is allocated automatically, initially a fairly minimal size, but with *grow-down* behaviour (specially flagged inside the Linux kernel's VMA (virtual memory area) structure). When the stack grows below its allocated space, more is allocated automatically. This flag cannot be relied upon when allocating from user-space however, so a reasonably large fixed-size block is allocated (currently 128 kilo-bytes).

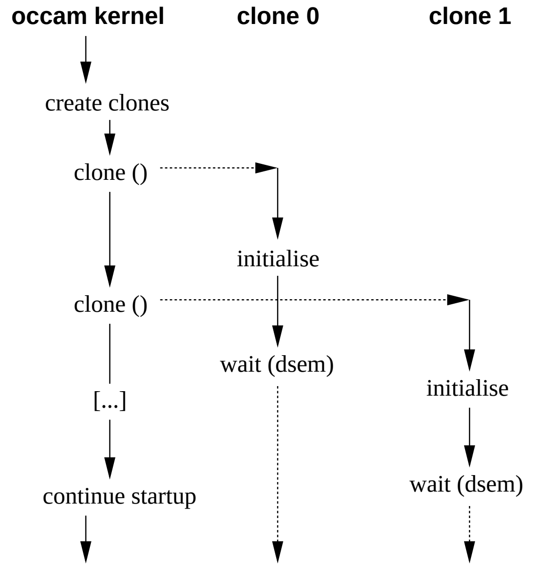


Figure 6.5: Blocking system-call clone startup

The run-time kernel maintains an array of pointers to these structures, used to help a clone locate itself during forced termination.

On startup, the run-time kernel initialises the variables associated with clone management as a whole, then creates a small pool of initial clones — currently three. When a clone starts up, it simply initialises itself then waits on '`dsem`' — the dispatching semaphore (described in section 6.3.5). This startup sequence (for two clone processes) is shown in figure 6.5.

6.3.5 Dispatching Blocking Calls

The occam interface to blocking system-calls is done through the use of external '`B.xxx`' and '`BX.xxx`' declarations, which invoke the corresponding C function '`_xxx`'. The standard interface to external C functions (via '`C.xxx`' external declarations) is handled in the translator (tranx86), that transforms the occam calling sequence into a C calling sequence, additionally saving and restoring the active state of the occam process.

In a similar manner, calls to '`B.xxx`' and '`BX.xxx`' functions are intercepted by the translator, that instead generates a run-time kernel entry into the entry-points '`X_b_dispatch`' and '`X_bx_dispatch`' respectively. The two arguments passed to the run-time kernel call are the address of the C function ('`_xxx`'), and the address of the occam arguments. As described in [Woo98], this occam argument-address is used by the called C function to get at the occam `PROC` arguments. This differs slightly from the interface mechanism first used, and presented in [Bar00b], where the translator used was tranpc.

Both blocking system-call entry-points in the run-time kernel call the generic clone dispatching function '`bsyscalls_dispatch`', that performs the actual dispatching. The implementation of this is shown in algorithm 6.1.

In addition to dispatching the requested call (specified by '*this-one*'), the algorithm also ensures that at least one clone will be available next time. The case where 'num-free' is zero will not happen in a uni-processor KRoC, but would theoretically be possible in an SMP implementation.

Algorithm 6.1: Blocking system-call dispatching algorithm (kernel-side)

```
1: //num-free : number of available clone processes
2: //dlock : dispatching spin-lock
3: //dsem : dispatching semaphore
4: //dinf : dispatching information
5: T_f ⇐ false
6: if num-free = 0 then
7:   create_clone()
8:   T_f ⇐ true
9: else if num-free = 1 then
10:   T_f ⇐ true
11: end if
12: while lock-or-fail (dlock) = false do
13:   sched_yield()
14: end while
15: dinf ⇐ this-one
16: num-free ⇐ num-free −1
17: sem_signal( dsem )
18: if T_f = true then
19:   create_clone()
20: end if
```

The spin-lock 'dlock' is used to protect the shared data structure 'dinf'. The dispatching algorithm only ever claims this spin-lock, calling the OS system-call '`sched_yield()`' (line 13) while the claim fails — in an attempt to schedule in the process holding the lock. When the 'dsem' semaphore is signalled (at line 17), a clone process will wake up, inheriting the access rights to the lock ('dlock') and the data ('dinf').

If another blocking call is immediately dispatched, before a clone has picked up the most recently dispatched call, claiming the lock fails, preventing the dispatching algorithm from racing with itself — in such (somewhat rare) cases, the occam kernel will spin (at lines 12-14) until a clone process wakes up and releases the lock.

The corresponding clone process algorithm is trivial and shown in algorithm 6.2. Clone processes simply wait on the semaphore 'dsem', and when released, copy 'dinf' into their local state before releasing the 'dlock' lock (at line 8).

Algorithm 6.2: Blocking system-call dispatching algorithm (clone-side)

```
1: //dlock : dispatching spin-lock
2: //dsem : dispatching semaphore
3: //dinf : dispatching information
4: ... initialise
5: while forever do
6:   sem_claim( dsem )
7:   local-state ⇐ dinf
8:   unlock ( dlock )
9:   ... perform blocking call (algorithm 6.6)
10:   ... queue suspended occam process (algorithm 6.3)
11: end while
```

Figure 6.6 shows the main actions and interactions of the occam kernel and any dispatching clone. In the data copied through the 'dinf' structure are fields containing the invoking occam process's workspace pointer and return address. Once dispatched, the occam kernel maintains no reference of the invoking occam process — if the C code called never terminates, the occam process will never be rescheduled.

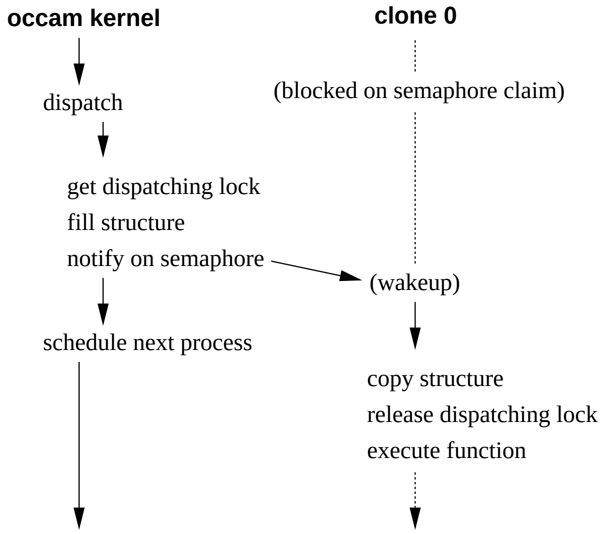


Figure 6.6: Blocking system-call clone dispatch

Even though such a process may be technically deadlocked, the run-time kernel cannot detect this, and will therefore not report deadlock and quit while incomplete blocking calls remain. The one exception to this is the new keyboard handling process (section 6.3.9), which is treated specially — by inspection of the top-level channel connecting the occam program to the keyboard handling process.

6.3.6 Collecting Finished Calls

When a clone process finishes executing a blocking system-call, either because it finished naturally or because it was forcefully terminated, the occam process that dispatched the call must be rescheduled. This is done by means of a shared process queue ('qfptr' and 'qbptr'), protected by a spin-lock ('queuelock'). If needed, a signal is also sent to the occam kernel, which responds immediately by setting the 'block_sync' synchronisation flag (section 6.2.3), and at some later time by moving processes from 'qfptr' to the run-queue. A count of the number of queued finished blocking calls is also maintained, 'numqueued'. Algorithm 6.3 shows the algorithm used by a clone process to reschedule an occam process.

Once the run-time kernel notices the 'block_sync' synchronisation flag, it locks the queue, clears the sync-flag, moves processes to the main run-queue, sets numqueued to zero, and releases the lock. For completeness, this is shown in algorithm 6.4. As noted before, if the run-time kernel was sleeping in '`safe_pause()`', this signal will wake it up.

After performing their side of the collection algorithm, the clone processes simply loop and block on the 'dsem' semaphore waiting for another job. The cost of the operation at line 11 in algorithm 6.4, where processes are moved from 'qfptr', 'qbptr' to the run queue, depends on whether

Algorithm 6.3: Blocking system-call clone re-synchronisation algorithm

```
//qfptr, qbptr : finished blocking calls queue
//numqueued : number of queued processes
//queuelock : spin-lock to protect the above items
lock ( queuelock )
// enqueue occam process stored in 'ws_ptr' (part of the local-state)
if numqueued = 0 then
    qfptr ⇐ ws_ptr
    qbptr ⇐ ws_ptr
    F_s ⇐ true
else
    qbptr[Link] ⇐ ws_ptr
    qbptr ⇐ ws_ptr
    F_s ⇐ true
end if
numqueued ⇐ numqueued +1
unlock ( queuelock )
if F_s = true then
    do_signal( occam-kernel )
end if
```

process priority is supported (section 7.2). In the absence of priority, this operation is $O(1)$, since there is only one run-queue to put the finished processes on. With priority (which is the default), this becomes an $O(n)$ operation. However, n will typically be small — smaller than the cost of using a separate finished-call queue for each priority level.

6.3.7 Terminating Blocking Calls

It is quite conceivable that a blocking call could block forever in the Linux kernel, or that mis-programming could result in a non-terminating loop (in the called C function). For these reasons, a safe mechanism for terminating them is provided.

The implementation of termination, like the other blocking system-call related algorithms, is split into two halves. The clone process performs one half of the algorithm whilst the occam run-time kernel performs the other. The ability to terminate a blocking call allows, for example, a socket write to be included as an `ALT` guard — although careful application design is required, as described in section 6.3.3.

From occam, termination of a blocking call is triggered through the use of the externally defined '`C.killcall()`' procedure, referencing the particular blocking call to be terminated with its magic channel parameter. That channel parameter, passed to the '`BX...`' procedure as the first argument is automatically 'hidden' from the underlying C function — that only sees the subsequent parameters. This allows the same underlying C function to be used for both a plain external '`C...`' call, a blocking external '`B...`' call, or an `ALT`-able (killable) blocking call '`BX...`'.

Algorithm 6.5 shows the algorithm implemented by '`C.killcall`'. While the clone is running, the termination channel 'tchan' contains a pointer to that clone's '`bsc_thread`' structure (described in section 6.3.4). This is used to allow '`C.killcall`' access to the clone's state, in particular the fields used to provide safe termination ('terminated' and 'cancel') and the process ID (the 'pid' field). The atomic swap at line 18 indicates the point where '`C.killcall`' either commits to terminating the clone (via the SIGUSR1 signal), or leaves it alone (because it is already terminating).

Algorithm 6.4: Blocking system-call finished-call collection algorithm

```
//qfptr, qbptr : finished blocking calls queue
//numqueued : number of queued processes
//queuelock : spin-lock to protect the above items
//num-free : number of available clone processes
entry(signal-handler):
block_sync ⇐ 1
return
entry(reschedule):
if combined_sync ≠ 0 then
   if block_sync = 1 then
      block_sync ⇐ 0
      lock ( queuelock )
      // move processes from 'qfptr' and 'qbptr' to the run-queue ('Fptr' and 'Bptr')
      num-free ⇐ num-free − numqueued
      numqueued ⇐ 0
      unlock ( queuelock )
   else
      // check for other sync flags
   end if
end if
// schedule next runnable process or sleep
```

Algorithm 6.5: Blocking system-call termination algorithm

```
// tchan : termination channel (passed to blocking call and killcall)
T ⇐ tchan
if T = null then
   // call has either not started or has already finished
   sched_yield()
   T ⇐ tchan
   if T = null then
      // call has finished
      return −1
   end if
end if
if T[terminated] then
   // call has finished but has not yet cleared channel word
   return 1
end if
F ⇐ 1
// atomically swap T[cancel] and F
swap T[cancel], F
if F = 1 then
   // call is in the process of terminating return 2
end if
// interrupt the clone by sending SIGUSR1
signal T[pid], SIGUSR1
return 0
```

On uni-processor machines in particular, termination may be attempted before Linux has had chance to schedule the clone process. In such cases, '`sched_yield()`' is called (at line 5) in an attempt to schedule the clone.

In some cases, it might be desirable to prevent the clone from dispatching at all (in cases where termination is invoked before the clone process has been scheduled). This would require significant modifications to the algorithms however, and is a hard thing to do given the dispatching method used (an OS-level semaphore). A relatively easy solution is available however, but only when more direct control over process scheduling and signalling is available — i.e. in the Linux kernel itself. This is investigated further in section 6.7.3.

The clone performs the other side of the termination algorithm, as shown in algorithm 6.6.

Algorithm 6.6: Blocking system-call clone termination algorithm

```
// tchan : termination channel (passed to blocking call and killcall)
T ⇐ pointer to bsc_thread structure
T[cancel] ⇐ 0
T[terminated] ⇐ 0
tchan ⇐ T
if sigsetjmp T[jbuf] = 0 then
    // returning directly
    unblock SIGUSR1
    call T[func] (T[ws_arg])
    T[terminated] ⇐ 1
    block SIGUSR1
    S ⇐ 0
else
    // returning from the SIGUSR1 handler
    T[terminated] ⇐ 1
    S ⇐ 1
end if
F ⇐ 1
// atomically swap T[cancel] and F
swap T[cancel], F
if F = 1 and S = 0 then
    // other side committed to terminating, but signal is pending
    wait SIGUSR1
end if
tchan ⇐ null
```

6.3.8 Performance

The performance of the blocking system-calls extension can be gauged, in part, by measuring its overhead. This can easily be done by comparing the overhead of a blocking call with its non-blocking version — simply a matter of changing the name in occam. However, such a test is unable to measure any performance gained from the parallel use of blocking system-calls — for example, reading from multiple sockets. The standard UNIX technique for such code would be a '`select()`' on all the sockets for reading, followed by serialised reading on those which were ready. A crucial question is, when compared with the alternatives, to what extent do the overheads of blocking system-calls damage performance — or, to what extent is performance improved[10].

The first benchmark used is a simple test that measures the raw overhead of blocking system-calls. This overhead is measured, at various levels of parallel granularity, by timing N parallel blocking calls. The underlying C function called is '`select()`', with a variable (possibly zero) timeout. Although the '`select`' call is not likely to be the main purpose for an application's use of blocking calls, it is certainly a possibility.

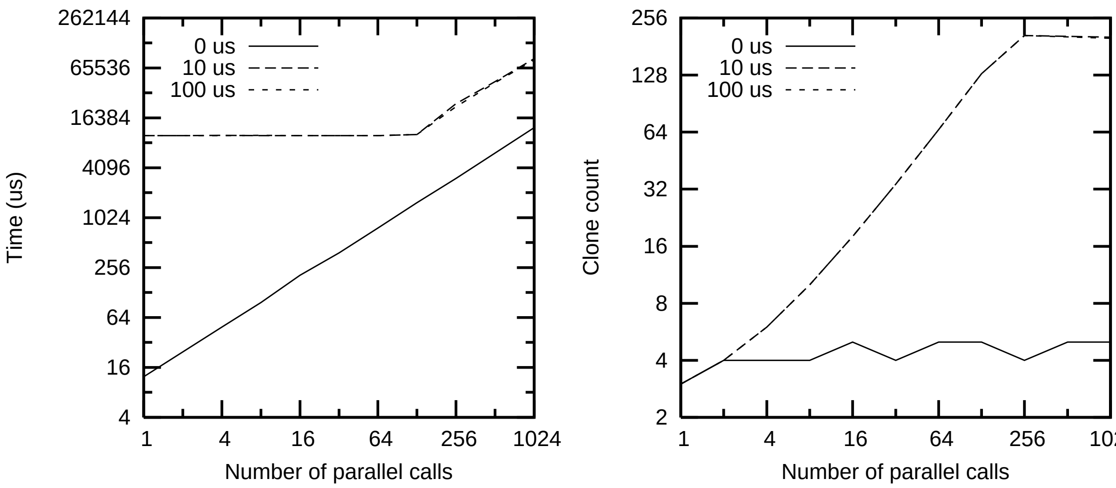


Figure 6.7: Blocking system-call overheads for `select()` with various timeouts

Figure 6.7 shows the results of this benchmark, for delays of 0, 10 and 100 micro-seconds. The time required for x parallel blocking calls is measured, as well as the maximum number of clone processes that were used. The main body of the benchmark is trivial, simply:

```
INT t.start, t.stop, ns.per.group:
TIMER tim:
SEQ
  tim ? t.start
  SEQ i = 0 FOR N.LOOPS
    PAR j = 0 FOR PAR.COUNT
      B.select (DELAY)
  tim ? t.stop
  VAL INT64 nstime IS (INT64 (t.stop - t.start)) * (1000 (INT64)):
  ns.per.group := INT (nstime / (INT64 N.LOOPS))
```

As can be clearly seen in the results, a 0 μs timeout behaves substantially differently from `select()` called with either 10 μs or 100 μs timeouts. This is primarily an effect of the 10 ms Linux kernel internal timer. Finer timeouts are potentially possible, using the '`nanosleep`' system-call. However, actually getting timeouts finer than 10 ms on Linux (by having the kernel perform busy waits – of 2ms or less) requires superuser privileges to change the default scheduling policy — which would actually have a negative effect by restricting the number of clone processes that could be scheduled.

The minimum overhead reported is approximately 12 μs, on the 800 MHz Pentium-3 used for this benchmark, at a parallel granularity of one. The overhead of a single `select()` call from `occam`

[10]Even if performance is damaged in some ways, just having the facility for multiple parallel blocking system-calls is justification enough.

in this benchmark is approximately 1 μs, setting the minimum time for the blocking system-call dispatch-collect cycle at 11 μs. A lot of this overhead can be attributed to the (Linux) kernel calls involved in dispatching and collecting processes; in particular, the overhead and latency of the signalling mechanism used to notify the run-time kernel of completed blocking calls.

Network Performance

The previous `select()`-based benchmark does not really represent a real-life case, but is useful for measuring the raw overhead of blocking calls. A fairly typical activity of many UNIX processes is selecting from multiple sockets, followed by sequential processing of data from ready sockets.

Given that `select()` is easily available to occam, applications have a choice of how to handle activity from multiple clients. At the finest level of granularity, each client can be handled by its own occam process. At the other end of the scale, the application could make a single `select()` call then process the active sockets either sequentially or in parallel. For socket reads, sequential processing after a `select()` will be more efficient, since data is being retrieved from kernel buffers directly.

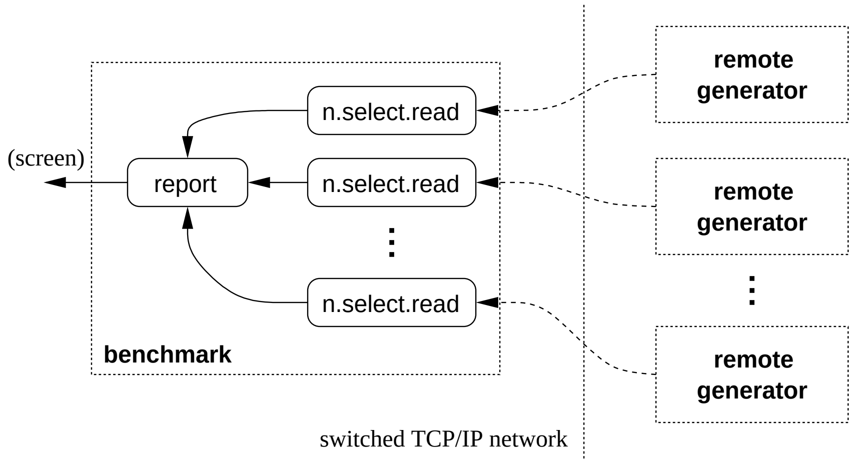


Figure 6.8: Blocking system-call TCP/IP network benchmark layout

Figure 6.8 shows the structure of this benchmark, with one occam process per connection. At coarser granularities, each occam '`n.select.read`' handles multiple remote generators (not shown). The "remote generator" processes are simple C programs that act as rate-controllable traffic generators. Connections are made outwards from the occam benchmark to these generators, that start generating traffic as soon as requested. Once all the required generators are active, the occam benchmark program simply measures the amount of data received within a given period of time (typically 5 seconds in this benchmark).

The C function called (as the blocking system-call), is implemented using '`select()`' followed by a '`read()`' on all the active sockets, before returning to occam. When calling '`select`' with multiple sockets, any number of sockets may be ready on return (between one and the number passed). At low CPU loadings, each call to '`select()`' with multiple sockets may result in only one or two being ready, increasing the number of 'blocking calls' required to handle each client. At higher CPU loads, with multiple sockets per '`select`', more sockets will be ready for reading per cycle, reducing the number of blocking calls performed. At some point, a 'minimal' overhead will be reached, with each call to '`select`' returning immediately with all sockets ready.

The maximum throughput attainable depends on several factors. The application controllable aspects are the frequency and size of communication, as well as some (OS) kernel networking parameters (such as send and receive buffer sizes). What cannot be controlled, however, is the network. At the simplest level this means having sufficient bandwidth, tolerable latency and 'fair' switching fabrics.

Figure 6.9 shows the results of this benchmark for two different networks. On the left are results for a relatively wide network, with eight traffic generators running on eight Sun Sparc workstations, and the results collected on a 550 MHz P3 four hops away. The results on the right are for a significantly smaller network, with two traffic generators running on each of two machines and the results collected on a 400 MHz P2.

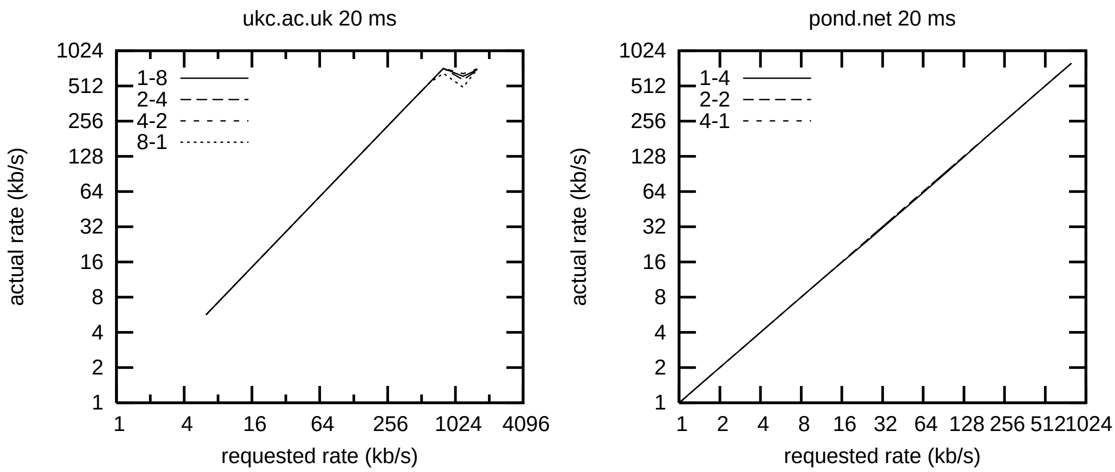


Figure 6.9: Blocking system-call network benchmark results

Unfortunately, these results only show that a either a slower CPU or faster network is required, as the performance for all cases is roughly similar. At some transfer rate, the CPU will become fully loaded and the lines will separate.

6.3.9 Interacting with occam Programs

Traditionally, KRoC programs have been connected to their surrounding environment by the use of three channels: '`keyboard`', '`screen`' and '`error`'. These represent the three standard UNIX I/O streams, '`stdin`', '`stdout`' and '`stderr`' respectively. In the original KRoC/Linux, the other ends of these channels were connected to *pseudo-processes* — functions in the run-time kernel that masqueraded as occam processes (as far as channel communication and scheduling were concerned).

Handling for the screen and error channels is relatively simple, but the keyboard channel is more problematic. The solution in KRoC [WW96] was to have a separate UNIX process reading from the keyboard, connected by a pair of pipes to the main run-time kernel process. The keyboard process reads a character from '`stdin`', signals the run-time kernel (whose signal-handler simply sets the 'keyboard' synch-flag), writes to the 'data' pipe then reads from the 'acknowledge' pipe. This arrangement is shown in figure 6.10.

The acknowledgement pipe is needed to prevent reading from '`stdin`' before the occam process has consumed the key-press — in retrospect, a form of the extended rendezvous (section 3.7). This

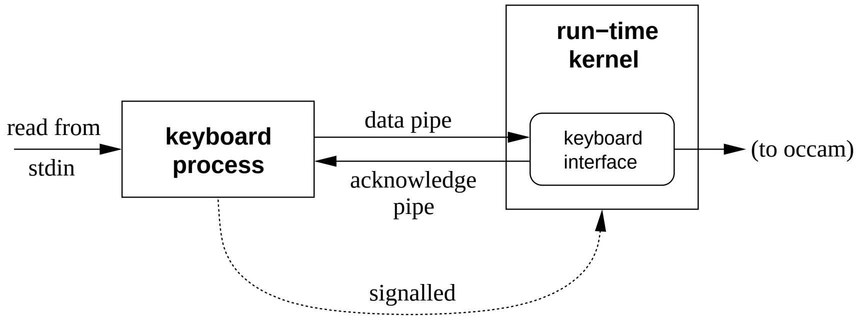


Figure 6.10: The original KRoC keyboard process

method was used in the original KRoC/Linux and still exists in current versions, but only when blocking system calls are unavailable.

The introduction of blocking system calls provided a practical keyboard-handling alternative, primarily in the interests of removing the pseudo-processes, which were fairly ungraceful bits of code. Additionally, the cost of two pipe reads and two pipe writes per input character is needlessly excessive, but was the only practical and portable technique at that time. With the availability of blocking system-calls, the keyboard pseudo-process can be transformed into a real `occam` process, that performs a blocking call to read from the keyboard before communicating it to the `occam` program.

This approach has been taken in recent KRoC/Linux versions, along with real `occam` processes that replace the screen and error pseudo-processes. The modified handling for the screen and error channels is used regardless of whether blocking system-calls are available (which largely depends on whether the running Linux kernel version supports the '`clone()`' system call[11]). The new arrangement of interface processes is shown in figure 6.11.

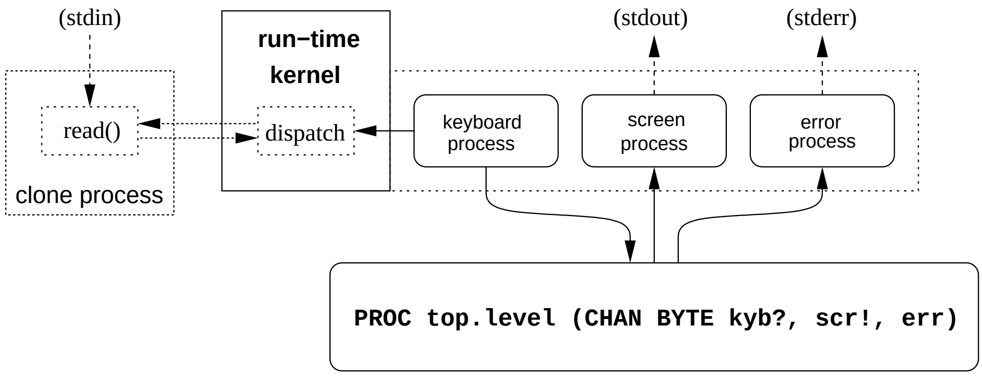


Figure 6.11: New `occam` keyboard, screen and error handling process

The advantages of this are twofold. Firstly, the keyboard synchronisation-flag (for cases when blocking system-calls are not available), and the blocking system-call synchronisation flag, can share

[11] The number of Linux installations which cannot support blocking system-calls is ever decreasing, but occasionally one crops up — some (mostly non-technical) people/organisations prefer to use old but working Linux installations, rather than take the risk of upgrading and ending up with a non-functional system.

the same space in the synchronisation word (section 6.2.3) — the new implementation of the keyboard process makes them mutually exclusive. Secondly, modification of any of the three occam processes is trivial — the occam sources are edited, the interface code re-generated[12], and finally the run-time kernel re-compiled.

One reason for modifying the occam interface code would be to allow the screen output to block in the background — the standard implementation simply writes, if that blocks, the run-time kernel blocks. As an example, the error-channel handling process is currently implemented as:

```
#PRAGMA EXTERNAL "PROC C.write.error (VAL BYTE ch) = 0"

PROC kroc.error.process (CHAN BYTE in?)
  WHILE TRUE
    BYTE ch:
    SEQ
      in ? ch
      C.write.error (ch)
:
```

Where '`C.write.error`' is an external C function defined in CCSP that simply writes characters to '`stderr`'. A blocking version could be created by substituting the name '`B.write.error`' for '`C.write.error`'. The screen process is less trivial since it line-buffers output by default, flushing the buffer when it fills up or sees one of the characters: newline ('`*n`'), carriage-return ('`*c`'), or the special flush character ('`*#FF`').

When the run-time system starts up, it allocates workspace for these processes, wires in the top-level KRoC channels and puts them on the run-queue. The top-level `PROC` is then connected to these channels and invoked directly through some suitable in-line assembler. When this call returns, the run-time system exits — regardless of the state of the keyboard, screen and error handling processes. To ensure that the screen output buffer is flushed before exit, tranx86 inserts code to send the flush character to the screen channel before the top-level `PROC` returns. This can be disabled with a command-line flag if a different screen-handling process is to be used.

To correctly detect a deadlocked program, the run-time kernel needs to inspect the three top-level channels. If any application-defined occam processes are blocked communicating on these channels, then the system is not deadlocked. However, if all three channels are either empty or contain their respective handler occam processes — and there are no pending timeouts or blocking system-calls (except possibly the keyboard process, which is specially ignored for this) — then deadlock is reported and the run-time system exits.

6.4 User Defined Channels

User-defined channels allow the placement of arbitrary C code behind occam channel operations, providing a mechanism for *external-channels*. This is certainly not a new idea: Vella [Vel98, VW99] provided external channels over networks in a SPARC version of KRoC, allowing large networks of occam processes to be distributed over networks of workstations. A similar mechanism also existed in the original CCSP [Moo99] (in the C interface), designed largely to support the external

[12] Before distribution, the occam interfaces are reduced to assembler by the translator. Both the occam source and generated assembler are shipped however. This stops the (stand-alone) CCSP source tree from requiring a functional KRoC. The source distribution '`build`' script does include support to generate the assembler at the appropriate point however, should it ever be required.

communication layer in MESH [BDv99a, BDv99b]. (CCSP and MESH were initially developed as a joint-project between CERN and UKC, before diverging).

The support for external channels added here is substantially more flexible than previous versions — largely as a direct result of the blocking system-calls extension (section 6.3). Currently, supporting the `ALT` for external events involves significant application design, as described in section 6.3.3. User-defined channels provide this support directly — with minimal application involvement.

Previous implementations of user-defined channels have imposed a slight run-time cost for ordinary occam channel communication. This arises from the way they are implemented — by setting the least-significant bit in the channel address to one. When communicating, the run-time kernel must mask this bit off, checking for an external channel, before making an appropriate call to external channel functionality. Although this overhead is small (around 4 machine cycles), it is one which we would wish to avoid if possible — in the interests of low overheads.

In previous versions of KRoC, external channel capability was a build-time option, and in initial releases of the new KRoC/Linux, this support was removed. The support for user-defined channels described here makes a compromise for run-time cost. Run-time kernel entry-points are provided specifically for external channel handling, leaving the ordinary communication instructions alone. With a compile-time flag, the translator will generate checking code around a communication instructions to redirect external channels to the alternative kernel entry-points. This flag only need be applied when compiling occam sources which use external channels. In some (rare) cases, this can lead to run-time failure — for example, when an external-channel is passed to an occam `PROC` compiled without the support required to handle it. Debugging support is available to help track these down, however, described in section 7.3.

Unlike previous external channel implementations, which were primarily designed for supporting synchronised channel communication over TCP/IP networks, user-defined channels provide support for any arbitrary external communication mechanism, of which networked channel communication is just one. Such networking of channels is still of primary interest however, but not within the scope of this thesis. Figure 6.12 shows an example of such connectivity.

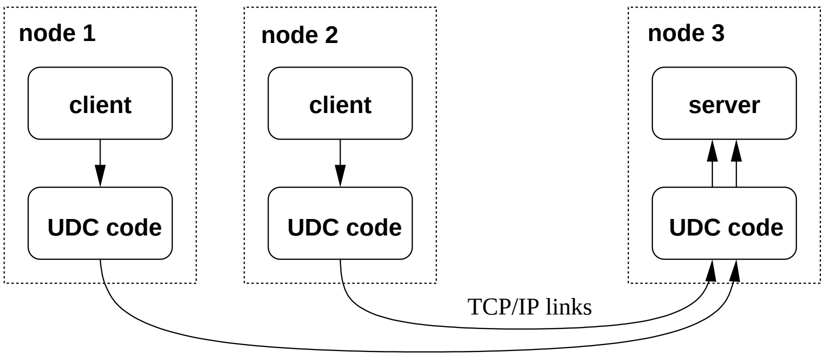


Figure 6.12: Example distributed occam network using user-defined channels

The code required to use a user-defined channel in occam is relatively trivial, consisting of calls to create and free external channels, plus a special language mechanism to *attach* an external channel to an occam name — described in section 6.4.1. The code needed to implement a user-defined channel is substantially more complex, but within the grasp of any competent C programmer. The user documentation for user-defined channels [Bar02] provides in-depth examples for creating user-defined channels. A brief overview is given here in section 6.4.2.

The implementation of user-defined channels is non-trivial, comprising of modifications to most parts of the KRoC/Linux system. For the occam compiler, these are a new set of instructions output by the compiler to handle user-defined channels (described in section 6.4.3), plus some additional support for parsing the language syntax described in section 6.4.1. The translator performs much of the work for handling user-defined channels at run-time, by inserting checks as described in section 6.4.4. The run-time handling of user-defined channels is described in section 6.4.5.

6.4.1 The occam Interface

In order to access a user-defined channel, an occam program must do two things. Firstly, the external channel must be allocated and initialised, that is returned as an integer address for occam. Secondly, the acquired address must be *bound* to an occam channel. Additionally, after use, the user-defined channel may be freed. Such proper destruction of resources is indeed encouraged; to promote good programming practice and prevent memory leaks (within the external C code).

The KRoC/Linux distribution includes an example '`softchan`' user-defined channel, that implements a simple channel (examples of buffered and socket channels are also provided). The '`softchan`' is accessed through the following (library-provided) `PROC`s:

```
PROC udc.softchan.alloc (RESULT INT addr)
PROC udc.softchan.free (INT addr)
```

These simply map onto underlying C functions that perform the required setup and initialisation. Once an address has been acquired using '`udc.softchan.alloc`', it is bound to an occam channel either by a '`PLACE`' statement or '`PLACED`' declaration. For example:

```
INT addr:
SEQ
  -- allocate channel
  udc.softchan.alloc (addr)

  -- use channel
  CHAN INT c:
  PLACE c AT addr:          -- AT is optional
  PAR
    producer (c!)
    consumer (c?)

  -- use it again
  PLACED CHAN INT c addr:
  PAR
    producer (c!)
    consumer (c?)

  -- free channel
  udc.softchan.free (addr)
```

Semantically, the placed '`c`' channel is no different from an ordinary occam channel. For other user-defined channels, such as the example buffered channel, the semantics change, and occam programs may need to be aware of this.

6.4.2 The C Interface

The C interface to user-defined channels is centred around a C data-structure ('ext_chan_t'), that holds all the information needed to support a user-defined channel. When a user-defined channel is created (by a C call from occam), parts of this structure must be initialised:

```
typedef struct {
    ...  private fields for occam/CCSP

    unsigned int magic;
    int flags;

    int (*chan_verify)(ext_chan_t *, unsigned int);
    int (*chan_read)(ext_chan_t *, char *, int);
    int (*chan_write)(ext_chan_t *, char *, int);
    int (*chan_alt_enable)(ext_chan_t *);
    int (*chan_alt_disable)(ext_chan_t *);
    int (*chan_min)(ext_chan_t *, void **);
    int (*chan_mout)(ext_chan_t *, void **);
    int (*chan_min64)(ext_chan_t *, unsigned long long *);
    int (*chan_mout64)(ext_chan_t *, unsigned long long *);
    int (*chan_minn)(ext_chan_t *, unsigned int *, int);
    int (*chan_moutn)(ext_chan_t *, unsigned int *, int);

    void *userptr;
} ext_chan_t;
```

The first group of fields (hidden) are private to the run-time implementation (described in section 6.4.5). The special 'magic' field should be initialised to the constant 'UDC_MAGIC'. This is used to ensure the integrity of a user-defined channel for some operations at run-time. The 'flags' field is a bitwise-or of flags that control the involvement of blocking system-calls (section 6.3) for channel operations: 'UDC_NONE' is the default value, 'UDC_BLOCKING_INPUT' and 'UDC_BLOCKING_OUTPUT' specify that channel input and output should use blocking system-calls. 'UDC_BLOCKING_ALT' specifies that channel 'ALT's should also be handled using blocking system-calls.

The third group of fields are pointers to functions that provide the desired functionality for communication and channel verification. The verification function 'chan_verify' must always be provided, even if it does nothing (ignoring it is not encouraged however). The unsigned integer argument passed is a hash-code for the underlying occam channel protocol. This can be used to ensure channel-protocol compatibility at run-time. The remaining fields are set to functions that provide the actual communication support. The user-documentation [Bar02] describes the functionality of all these fields in detail.

The final 'userptr' field is optional space allocated for a user-defined channel to store its own (user-defined) state. To simplify the process of initialisation, a run-time function ("ccsp_udc_alloc_extchan (int eb)") is provided that allocates and initialises an 'ext_chan_t' structure to a default state, with additional space (of 'eb' bytes) pointed to by 'userptr'. Similarly, a "ccsp_udc_free_extchan" function is provided to free it.

A few other user-callable functions are provided for use at run-time, mainly to resume suspended occam processes and handle blocking ALTs. The operation of these are described in sections 6.4.5 and 6.4.6.

6.4.3 Generating Code for User Defined Channels

Once initialised in the C world, an occam program uses a PLACE statement or PLACED declaration to attach a user-defined channel to an occam name, as shown in section 6.4.1. For example, using the traditional 'PLACE' form:

```
CHAN INT c:
PLACE c AT addr
...  process using 'c'
```

Unlike ordinary occam channels (that get initialised to 'NULL' after declaration), user-defined channels (indicated to the compiler by a run-time computed PLACEment — 'addr') cause the name 'c' to be a pointer to the channel — rather than the channel itself. The initialisation code for these stores the computed address in the channel name, before calling the external channel verification function. The above occam fragment, for example, will generate the following (annotated) code:

```
LD addr              -- load address
LDC prothash         -- load protocol hash-code
EXTVRFY              -- external channel verify
LD addr              -- load address
LDC 1
OR                   -- set the lsb to indicate an external channel
ST c                 -- store in channel name
```

The 'prothash' constant is generated internally by the compiler. The programmer interface for user-defined channels defines various constants which represent the protocol hash-codes for common protocols (such as INT, BYTE and INT::[]BYTE), described in [Bar02].

After this initialisation, the compiler treats 'c' as an ordinary channel pointer. Where input, output and ALTs are performed directly on an external (PLACEd) channel, the compiler generates explicit external communication instructions: EXTIN, EXTOUT, EXTENBC, EXTDISC and various others (for external MOBILE communication). The full list is given in Table B.4, with details in Appendix B.1.4.

6.4.4 Compile-Time Translation for User Defined Channels

In cases where the external nature of a channel is unknown (typically in PROC formal parameters), the compiler generates ordinary channel communication instructions, leaving it up to the translator (tranx86) to check whether a channel is external or not. The interception of such instructions by the translator is controlled by a run-time flag, so the overhead need not be incurred where external channels will not be used.

The run-time check inserted by the translator checks the least-significant bit of the channel address. If set, the bit is removed and the corresponding external channel entry-point used. Otherwise the existing channel operation is performed unaltered. For example, the following occam fragment:

```
INT x:
SEQ
  c ? x
  ...  process using x
```

generates the (annotated) code:

```
AJW -1                        -- room for x
LD c                          -- load channel
LD ADDRESSOF x                -- load target address
LDC 4                         -- load count
IN                            -- channel input
...  code for process using x
AJW 1                         -- lose x
```

The translator turns this into the appropriate code to interface with the run-time kernel, inserting an external-channel check if desired. The code below shows an approximation of this output:

```
    subl    $4, %ebp            ; AJW -1
    movl    8(%ebp), %eax       ; LD c
    leal    4(%ebp), %ebx       ; LD ADDRESSOF x
    movl    $0f, -4(%ebp)       ; save return address (for IN)
    movb    %al, %dl            ; copy bottom 8 bits of channel address
    andb    $0xfc, %al          ; mask off bottom 2 bits of channel address
    andb    $0x03, %dl          ; mask in bottom 2 bits of channel address
    jz      _Y_in32             ; conditional jump to ordinary input entry-point
    jmp     _Y_extin32          ; jump to external input entry-point
0:
    addl    $4, %ebp            ; AJW 1
```

Similar checks are made for other communication instructions, including ALTs. In places where the compiler drops explicit external channel instructions ('EXTIN' for example), the bottom 2 bits must still be masked off before entering the run-time kernel.

6.4.5 Run-Time Handling for User Defined Channels

The previously hidden fields of the 'ext_chan_t' structure are:

```
void *chanword;               // channel word
unsigned int chantype;        // protocol hashcode
void *chanbxalt;              // used for blocking ALTs
void *chansync;               // used for blocking ALTs
```

The 'chanword' field behaves like the traditional occam channel-word. If the channel is empty (inactive), 'chanword' is set to null (technically 'NotProcess.p'). Otherwise this field points at the workspace of a suspended occam process. The 'chantype' field is used for extra internal checks during channel verification, and holds the channel protocol hash-code generated by the compiler (supplied to the run-time system through the 'EXTVRFY' instruction).

The 'chanbxalt' and 'chansync' fields are used for the implementation of blocking ALTs, that requires careful programming in order to avoid race-hazards.

The kernel entry-points for external communication provide the necessary checks and glue-code to perform the external operation. As a general rule, all the linked C functions which make up the external channel (function-pointers in the 'ext_chan_t' structure) return an integer which indicates the action to be performed by the occam kernel — either to suspend the invoking process or to continue running (on successful communication). If the operation is flagged as 'blocking' in the 'flags' field, then a blocking system-call is made to the user code – whose return value is simply discarded (the operation always runs to completion).

Algorithm 6.7 shows the algorithm used for external channel input. External channel output and mobile communications follow along similar lines. Mobile communication is slightly different

Algorithm 6.7: External channel input algorithm

```
// Wptr : invoking workspace pointer
external.channel.input (chan.addr, dest.addr, count):
if (chan.addr = NotProcess.p) or (chan.addr[magic] ≠ UDC_MAGIC) then
   error (invalid channel)
else if chan.addr[chan_read] = null then
   error (channel not for input)
end if
if chan.addr[flags] and BLOCKING_INPUT then
   dispatch3 (Wptr, Wptr[Iptr], chan.addr, dest.addr, count, chan.addr[chan_read])
else
   T ⇐ chan.addr[chan_read] (chan.addr, dest.addr, count)
   if T ≠ 0 then
      // suspend occam process in channel and reschedule
      chan.addr[chanword] ⇐ Wptr
      reschedule
   end if
end if
```

in that no size field is given. Since the default implementation for mobile communication works by pointer-swapping, the size of the data pointed at is not required. For both static and dynamic mobile communication, user-defined channel code gets passed a pointer to the workspace location where the scoped-in mobilespace pointer (section 4.2.3) is allocated. For static mobiles, this is a single word in workspace which points into the statically allocated mobilespace. For dynamic mobile arrays, there are $n + 1$ words in the process workspace — the pointer to the dynamically allocated block first, followed by n dimension counts. For the external versions of '`MIN64`' and '`MOUT64`', the number of dimensions is fixed at 1. For external versions of '`MINN`' and '`MOUTN`', the number of dimensions is passed as a parameter to the user-defined channel code.

To provide access to the sizes of mobile variables (the whole fixed-size for static mobiles, and the base-type size for dynamic mobile arrays), a compiler option is provided which allocates an extra word in the occam workspace for mobile variables. When a mobile variable comes into scope, this extra word is initialised to the correct size. By default, when compiling occam code with user-defined channel support (the '`-e`' flag to '`kroc`'), this *mobile-size-field* is enabled. For user-defined channel code, static mobile communication functions get passed a pointer to two words in workspace — the pointer into mobilespace and the size of the underlying type. For dynamic mobile communication, the base-type size is found in the $(n + 1)^{\text{th}}$ workspace offset (0 being the pointer, followed by n dimension sizes).

To reschedule an occam process that is blocked in a user-defined channel, a call to "`ccsp_udc_resume_process (ext_chan_t*)`" is made, passing a pointer to the user-defined channel in which that process is blocked. Algorithm 6.8 describes the operation of this function, that must be called from within the context of an occam process. Usually, the call will be made during user-defined input or output code (to reschedule the other side).

In all cases, the user-defined channel code must handle all the aspects of communication, including storing destination and source pointers if necessary. The example programs provided in the KRoC/Linux distribution define private structures which hold the necessary state. The size of this state is passed as an argument to "`ccsp_udc_alloc_extchan`" which stores a pointer to the required size in the '`userptr`' field. Subsequently, operations on the channel have easy access to this state through that '`userptr`'.

Algorithm 6.8: External channel process resume algorithm

```
// chan.addr : address of ext_chan_t structure
if (chan.addr = NotProcess.p) or (chan.addr[magic] ≠ UDC_MAGIC) then
  error (invalid channel)
else if chan.addr[flags] bitand BLOCKING then
  error (attempting to resume blocking thread)
else if chan.addr[chanword] = NotProcess.p then
  error (channel empty)
end if
P ⇐ chan.addr[chanword]
if P and 1 then
  // something ALTing
  P ⇐ (P and 1)
  if P[Ptr] = Waiting.p then
    queue P
  end if
  P[Ptr] ⇐ Ready.p
else
  queue P
end if
chan.addr[chanword] ⇐ NotProcess.p
```

6.4.6 ALTing on User Defined Channels

The mechanisms provided to support `ALT`s on user-defined channels are significantly more complex than simple input and output. Providing a mechanism for `ALT`ing against external events is a powerful feature. For example, reading data from multiple TCP connections or timing-out can easily be done using the socket channel (‘`sockchan`’):

```
[2]INT addr:
SEQ
  ...  setup addr[0] and addr[1] for socket channel
  PLACED CHAN INT::[]BYTE c AT addr[0]:
  PLACED CHAN INT::[]BYTE d AT addr[1]:
  WHILE TRUE
    TIMER tim:
    INT t:
    [DATA.SIZE]BYTE data:
    INT len:
    SEQ
      tim ? t                  -- read current time
      t := t PLUS 1000000      -- timeout in 1 second
      PRI ALT
        c ? len::data
          ...  process [data FOR len]
        d ? len::data
          ...  process [data FOR len]
        tim ? AFTER t
          ...  timed out (no data read)
```

The `ALT` construct accesses user-defined channel code through the '`EXTENBC`' and '`EXTNDISC`' (section 3.9.2) instructions, which in turn invoke the '`chan_alt_enable`' and '`chan_alt_disable`' functions in the user-defined channel code. Algorithms 6.9 and 6.10 show the algorithms used to implement '`EXTENBC`' and '`EXTNDISC`' respectively.

Algorithm 6.9: External channel `ALT` enabling algorithm

```
// Wptr : invoking (ALTing) workspace pointer
external.channel.enable (chan.addr, guard):
if (chan.addr = NotProcess.p) or (chan.addr[magic] ≠ UDC_MAGIC) then
  error (invalid channel)
else if chan.addr[chan_alt_enable] = null then
  error (channel not for ALTing)
end if
if guard then
  if chan.addr[flags] bitand BLOCKING_ALT then
    chan.addr[chanbxalt] ⇐ NotProcess.p
    chan.addr[chansync] ⇐ NotProcess.p
  end if
  T ⇐ chan.addr[chan_alt_enable] (chan.addr)
  if T then
    // external channel ready
    Wptr[Ptr] ⇐ Ready.p
  else
    // store ALTing process in channel
    chan.addr[chanword] ⇐ (Wptr bitor 1)
  end if
end if
return guard
```

The requirements for a user-defined `ALT` are either blocking or non-blocking. In both cases, the enabling function returns a flag indicating whether the channel is ready, as does the disabling function.

To handle a blocking `ALT`, the user-code inside the enabling function makes a call to "`ccsp_udc_start_alter`", that dispatches a blocking system-call to handle the `ALT`. A corresponding function is used in the disabling function to terminate the blocking `ALT` — "`ccsp_udc_kill_alter`".

The blocking `ALT` mechanism is engineered in such a way that if the blocking call returns whilst the `ALT`ing process is blocked (in '`ALTWT`' or '`TALTWT`'), then that `ALT` is optionally woken-up, with the external channel ready. The user code dispatches the blocking `ALT` process by calling "`ccsp_udc_start_alter`", providing a pointer and argument to a C function which will perform the blocking call.

In the socket channel, for example, this is implemented by a simple '`select()`' call on the socket. The C function that performs the blocking call takes two parameters: the user-argument supplied to the 'start-alter' function, and a pointer to a *wake-flag*. This flag, initially set to zero, is examined when the blocking call finishes. If set to 1, and the `ALT`ing process is blocked (waiting), then that `ALT`er is rescheduled. Otherwise the `ALT`ing process is left alone.

In order to use the blocking system-calls extension for `ALT`ing, a small *pseudo-process* is created. The purpose of this pseudo-process is to provide an occam process context, that is scheduled when the blocking call completes. A blocking `ALT` always finishes in one of two ways. Either the call finishes naturally, in which case the pseudo-process is scheduled, or it is killed from within the

Algorithm 6.10: External channel `ALT` (reversed) disabling algorithm

```
// Wptr : invoking workspace pointer
external.channel.disable (chan.addr, guard, proc.addr):
load return.address
if (chan.addr = NotProcess.p) or (chan.addr[magic] ≠ UDC_MAGIC) then
   error (invalid channel)
else if chan.addr[chan_alt_disable] = null then
   error (channel not for ALTing)
end if
if guard then
   T ⇐ chan.addr[chan_alt_disable] (chan.addr)
   if T then
      // channel ready
      Wptr[Temp] ⇐ proc.addr
      fired ⇐ 1
   else
      if chan.addr[flags] bitand BLOCKING_ALT then
         // channel not ready, blocking call terminated – wait for it
         if chan.addr[chansync] = NotProcess.p then
            chan.addr[chanbxalt] ⇐ return.address
            chan.addr[chansync] ⇐ Wptr
            reschedule
         end if
         chan.addr[chansync] ⇐ NotProcess.p
      end if
      chan.addr[chanword] ⇐ NotProcess.p
      fired ⇐ 0
   end if
else
   fired ⇐ 0
end if
return fired
```

channel disabling code by a call to "`ccsp_udc_kill_alter`" — that simply invokes '`_killcall()`' to terminate the blocking call.

Figure 6.13 shows the workspace layout and initialised values for the pseudo-process. This workspace also contains the 'magic channel' used to terminate blocking system-calls (section 6.3.7). A pointer to this is stored at offset 2 in the workspace, with the pointer to the (blocking) user function at offset 1. The "`ccsp_udc_start_alter`" function dispatches a blocking system-call with the context of the pseudo-process. A small stub function is called first, that re-arranges the parameters slightly, in order to provide the user function with the supplied argument (at workspace offset 3) and a pointer to the wake-flag (at workspace offset 4). The return address for the pseudo-process is set to the address of a small section of jump-code, that re-enters the run-time kernel through the '`_X_udc_fallout`' entry-point, making it appear that the pseudo-process invoked it.

The algorithm used to support the completion of a blocking `ALT`, implemented by '`_X_udc_fallout`', is shown in algorithm 6.11. This code is run always, regardless of whether it was triggered by a '`_killcall()`' termination on the blocking-call, or whether the blocking call finished naturally. Algorithms 6.10 and 6.11 ensure that the blocking `ALT` has completed at the point where the channel is disabled.

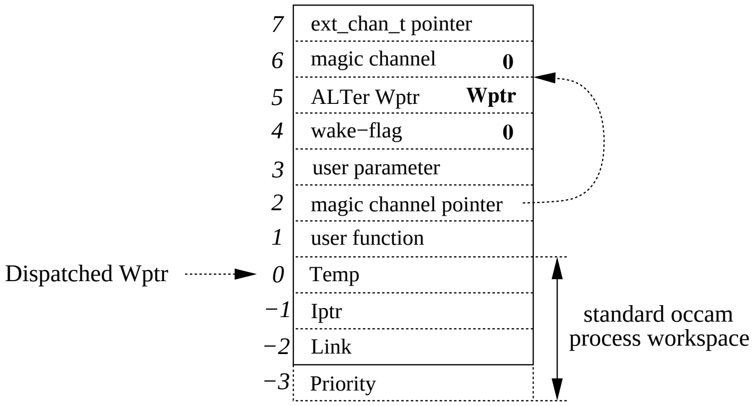


Figure 6.13: Workspace layout for user-defined channel blocking ALT pseudo-process

To illustrate how these algorithms interact, consider the following simple occam fragment:

```
ALT
  c ? x
    P (x)
  d ? y
    Q (y)
```

where 'c' is an external (user-defined) channel, and 'd' is an ordinary occam (soft) channel. Although 'PRI ALT' may be a better choice for run-time efficiency (it will use the enhanced ALT enabling sequence — section 3.9.3), the plain 'ALT' generates less code. For the above occam fragment, the enabling and disabling code is:

```
ALT                         -- begin ALT
--{{{  ALT enabling
LD c                        -- load channel pointer
LDC 1                       -- load TRUE pre-condition
EXTENBC                     -- enable external channel
LD d                        -- load channel pointer
LDC 1                       -- load TRUE pre-condition
ENBC                        -- enable channel
--}}}
ALTWT                       -- ALT wait
--{{{  ALT disabling
LD c                        -- load channel pointer
LDC 1                       -- load TRUE pre-condition
LD ADDRESSOF :L1            -- load process address
EXTNDISC                    -- disable external channel
LD d                        -- load channel pointer
LDC 1                       -- load TRUE pre-condition
LD ADDRESSOF :L2            -- load process address
NDISC                       -- disable channel
--}}}
ALTEND                      -- finish ALT
```

Algorithm 6.11: External channel blocking `ALT` completion algorithm

```
// Wptr : workspace pointer (of pseudo-process)
if Wptr[4] = 1 then
  // resume any suspended ALT
  T ⇐ Wptr[5]
  if T[Ptr] = Waiting.p then
    T[Ptr] ⇐ Ready.p
    queue T
  end if
end if
E ⇐ Wptr[7]
Wptr ⇐ Wptr −4
... free pseudo-process workspace
Wptr ⇐ NotProcess.p
if E[flags] bitand BLOCKING_ALT then
  if E[chansync] ≠ NotProcess.p then
    // NDISC process is blocked here, take over from it
    return.address ⇐ E[chanbxalt]
    Wptr ⇐ E[chansync]
    E[chansync] ⇐ NotProcess.p
    E[chanbxalt] ⇐ NotProcess.p
    E[chanword] ⇐ NotProcess.p
    fired ⇐ 1
    return fired
  end if
  E[chansync] ⇐ E
end if
reschedule
```

This is followed by the code for the guarded processes themselves, jumped to from within the '`ALTEND`' instruction:

```
:L1
  ...  code for "c ? x -> P(x)"
  J :L3
:L2
  ...  code for "d ? y -> Q(y)"
  J :L3
:L3
```

Figure 6.14 shows the run-time behaviour for such code, showing both cases for termination — the 'early synchronisation' (where the blocking `ALT` finishes before the disabling sequence gets called), and the 'late synchronisation' (where the disabling sequence kills the blocking `ALT`).

One feature which becomes possible using blocking `ALT`s is that of external `ALT`-able *output* events. Because the occam compiler disallows output guards in `ALT`s, such external outputs must be masqueraded as inputs for `ALT`ing[13].

[13] A possible route for `ALT`ing external output would be to allow, specifically, user-defined channel outputs in `ALT`s, generating almost the same code. This has not been implemented however, as it does not fit naturally with occam (that has no support whatsoever for output guards in `ALT`s).

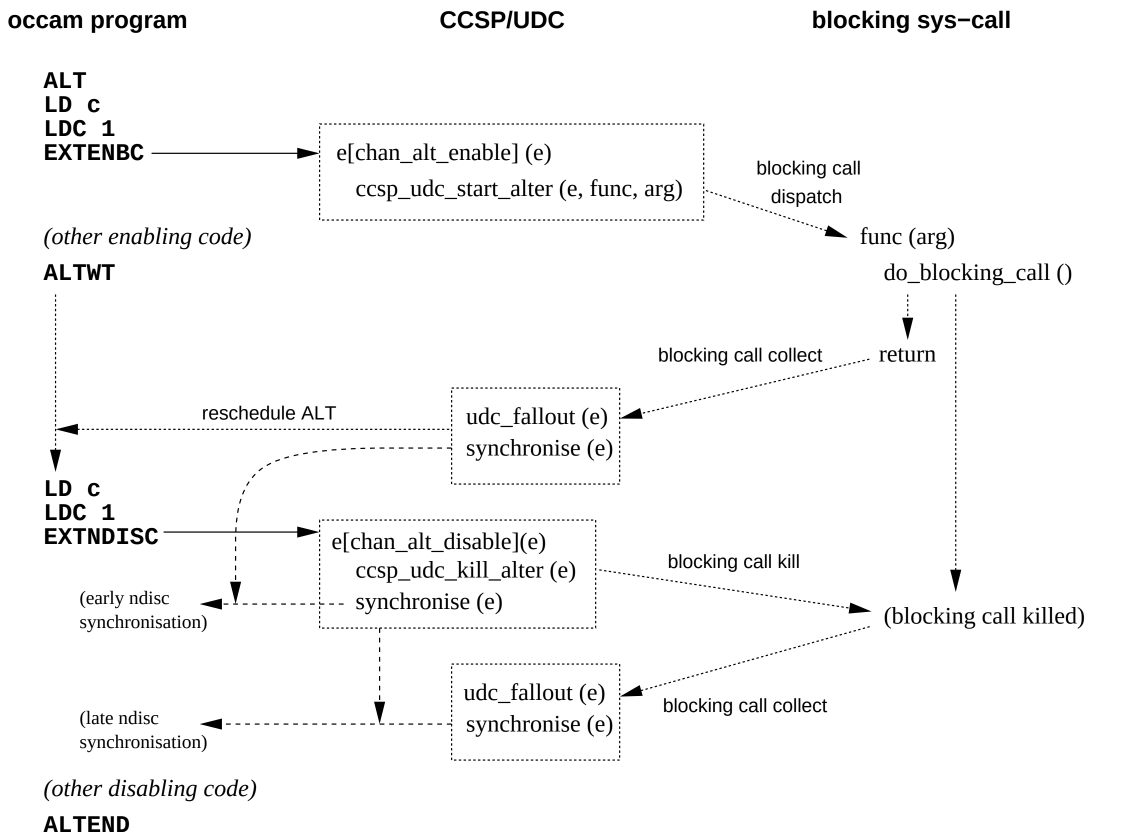


Figure 6.14: External channel blocking `ALT` run-time behaviour

To correctly disguise an output as an input, a *static mobile* user-defined channel must be used. For example:

```
DATA TYPE PACKET IS MOBILE [1500]BYTE:
INT addr:
SEQ
  ...  initialise special user-defined channel
  PLACED CHAN PACKET c AT addr:
  WHILE TRUE
    PACKET data:
    SEQ
      ...  populate ‘data’
      PRI ALT
        c ? data
          ...  ‘data’ output and still defined
        tim ? AFTER timeout
          ...  timed-out.  ‘data’ not output, but still defined
```

This ensures that the pointer passed to the underlying C function (‘`chan_min`’) is indeed ‘`data`’. The pointer actually passed to the user code is one into the `occam` workspace, that contains a pointer to ‘`data`’, and optionally a mobile-size-field.

Using dynamically sized mobile variables is not an option, since the compiler generates code that frees the dynamic mobile before performing the input. Ordinary channel input is not recommended either, since the compiler could, potentially (for pathological cases), allocate a temporary for the input and assign it to '`data`' later. For a user-defined channel performing output, this would mean the pointer passed would point at truely uninitialised data.

6.5 Dynamic Loadable Processes

This section presents a mechanism which enables the dynamic loading and connection (through a channel interface) of pre-compiled occam processes. Unlike the dynamic process mechanisms presented in chapter 5, this extension does not require advance knowledge about the process being loaded. The extension uses a UNIX mechanism for the dynamic loading of program code (the dynamic linker — '`ld`'), and requires no changes in the occam compiler.

The perceived use of this functionality is for building module-based programs, that consist of some core components supplimented by run-time loaded modules for specific functionality.

Once loaded and connected, a dynamically loaded process (possibly a network of sub-processes) becomes part of the overall network itself. This allows it to load and connect other dynamically loadable processes (including other instances of itself). Although not strictly intentional, this provides a method for parallel self-recursion, mutual-recursion and any cyclic call structure in general. Only self-recursion is allowed directly within the occam language (discussed in section 5.1.4).

Dynamically loadable processes are contained within *shared object* ('`.so`') libraries. A single library may contain an arbitrary number of dynamic processes. In order to generate a library, a special flag is passed to the '`kroc`' wrapper-script which causes the creation of a shared-object, rather than an executable, from occam sources. Only a few changes are required when translating the occam compiler output in order to build a dynamic process, described in section 6.5.1.

A set of `FUNCTION`s and `PROC`s are provided for occam programs, that load and unload dynamic process libraries, load `PROC`s from libraries (and free them again), and 'activate' (execute) loaded `PROC`s. These are described in section 6.5.2.

One additional feature engineered into the dynamic processes extension is the ability to suspend and resume the dynamic process. Once suspended, the dynamic process's state can be written to disk, or accessed as an array of bytes, allowing it to be communicated over networks. In a similar way, a previously suspended process may be loaded from disk and resumed, or a remotely sent process recieved and resumed. This leads to several useful features, described with the details of suspending and resuming in section 6.5.3. The implemenation of this mechanism is described in section 6.5.4.

6.5.1 Generating Dynamic Processes

Dynamic process libraries are created by giving the '`-l`' flag to the '`kroc`' command. This flag also selects the '`-c`' (compile only) option — dynamic libraries may not contain the outermost top-level process for an occam program. When generating code for top-level `PROC`s (those which are defined at the left-most level of indentation), the compiler outputs special constants which define the workspace, vectorspace and mobilespace usage for those `PROC`s.

When translating ordinary occam, tranx86 ignores most of these directives, and only outputs usage information for the last `PROC` — where the occam program will start. This information is encoded in three constants: '`_wsbytes`', '`_vsbytes`' and '`_msbytes`'.

For dynamically loadable processes, the handling of these constants is changed such that the memory usage for *all* top-level PROCs is included in the translated code, and placed into the shared-object ('.so') file. The names of the generated memory usage constants follow the PROC name. An occam process called 'thing' will generate the entry-point 'O_thing' and the constants '_thing_wsbytes', '_thing_vsbytes' and '_thing_msbytes'. All memory usage information is output, even if zero.

The PROC interface to loadable dynamic processes is fixed — an array of input channels and an array of output channels, all carrying the special 'ANY' protocol. For the programmer, this means that the *actual* channel-only interface must be re-typed and re-arranged, on both sides. For example, a very simple dynamic process that has a single 'BYTE' output-channel could be implemented by:

```
PROC test.process (VAL DPROCESS me, []CHAN ANY in?, out!)
  CHAN BYTE out! RETYPES out[0]!:
  VAL []BYTE str IS "Hello, dynamic world!*n":

  SEQ i = 0 FOR SIZE str
    out ! str[i]
:
```

The leading 'VAL DPROCESS me' is a process handle for the process itself — similar to 'this' in Java and C++ — used for suspending and other operations.

6.5.2 Loading and Running Dynamic Processes

Dynamic process libraries are handled at run-time using the Linux/UNIX dynamic-linker, 'libdl'. This allows a program to dynamically load a '.so' file, then extract pointers to symbols within it (using the 'dlopen()' and 'dlsym()' functions). For the occam programmer, all this functionality is managed by the run-time kernel, and accessed through a number of run-time functions — that are presented as INLINE PROCs to occam:

```
PROC ccsp.openlib (VAL []BYTE filename, RESULT INT handle)
PROC ccsp.loadproc (VAL INT handle, VAL []BYTE procname, RESULT DPROCESS p)
PROC ccsp.runproc (DPROCESS p, []CHAN OF ANY in, out, RESULT INT result)
PROC ccsp.freeproc (DPROCESS p)
PROC ccsp.closelib (VAL INT handle)

PROC ccsp.run (VAL []BYTE procname, VAL INT libhandle, []CHAN OF ANY in, out,
               RESULT INT result)
```

To open a dynamic library, an occam program simply calls 'ccsp.openlib', passing the filename[14] as an argument. If the library is opened successfully, an integer handle to it is returned. On failure, this handle is set to zero. A load failure, other than invalid file, occurs when symbols in the dynamic library cannot be resolved. For occam, this can happen when a dynamic process has '#USE'd an occam library, but is not linked against it — either in the dynamic process or the main occam program.

To load a process from a dynamic library, an occam program calls 'ccsp.loadproc', passing the relevant library handle and the textual name of the process. On success, the 'DPROCESS p' parameter is assigned a handle for that loaded process, or 'NOTPROCESS.D' (zero) on failure.

[14] The filename passed to the dynamic linker can optionally be path-free, in which case it must be reachable via the 'LD_LIBRARY_PATH' environment-variable or the system-wide configuration in '/etc/ld.so.conf'.

Once loaded, a process is attached to the existing occam network and scheduled by calling 'ccsp.runproc'. The process handle and channel interface are passed as parameters, with the outcome (after completion) stored in the 'result' parameter. On success, this will be either 'DPROCESS.FINISHED' or 'DPROCESS.SUSPENDED'. If the process generates a run-time error, it is deactivated and 'DPROCESS.FAULTED' is returned.

The last of these, 'ccsp.run', is simply a wrapper around 'ccsp.loadproc', 'ccsp.runproc' and 'ccsp.freeproc', with some basic error handling. With the exception of 'ccsp.runproc', these occam PROCs call underlying C functions in the run-time kernel that perform the relevant dynamic-library operations.

For each dynamic process loaded, a structure is allocated in the run-time kernel that maintains essential house-keeping information — for instance, pointers to the allocated workspace, vectorspace and mobilespace for the dynamic process. A pointer to this structure is returned to the occam program when it successfully loads a dynamic process (with 'ccsp.loadproc'). The run-time kernel keeps track of all loaded dynamic processes, as both a flat-list and as a tree — where descendants indicate dynamic processes started from within another dynamic process. The list at the top-level incurs the obvious $O(n)$ cost when searching for a particular process. A balanced tree (AVL for example) would provide a more efficient search at around $O(log_2 n)$. However, a process need only be searched for when a run-time error occurs, making it a rather low-priority optimisation.

The PROC call to 'ccsp.runproc' is intercepted by the translator, that generates code for a special run-time entry-point, 'KR.run'. The operation of this call is to enqueue the dynamic process and suspend the current process, resuming when the dynamic process terminates normally, suspends itself or causes a run-time error.

Assuming compilation into a dynamic library called 'test.so', the earlier 'test.process' can be dynamically loaded and executed with:

```
#INCLUDE "dynproc.inc"

PROC test (CHAN BYTE screen!)
  INT handle:
  SEQ
    ccsp.openlib ("./test.so", handle)
    IF
      handle <> 0
        [1]CHAN ANY test.out! RETYPES [scr!]
        INT res:
        SEQ
          ccsp.run ("test.process", handle, [], test.out!, res)
          ...  take action based on outcome in "res"
          ccsp.closelib (handle)
      TRUE
        SKIP
:
```

The usefulness of this dynamic process loading mechanism hinges on the fact that nothing about the process (or network of processes) being launched need be known in advance — with the exception of its channel interface, that is abstracted into the two arrays of ANY channels. No input channels are required by this process, so an empty-array specifier is used (introduced in section 3.5.2).

Once connected into an existing process network, no additional overheads are incurred at run-time for the dynamic process. Mobile data structures, arrays, channel bundle ends and processes

(Chapter 4) may be communicated through this channel interface, providing a low-cost method for transferring data in and out of a dynamically loaded process.

The occam web-server [Bar03] does exactly this when handling dynamically loadable '`.ogi`' modules. The connection between the web-server and an OGI module is comprised of two tagged-protocol channels, that carry mobile connection structures and higher-level control information. Figure 6.15 shows an example of the run-time network that might be created in the web-server.

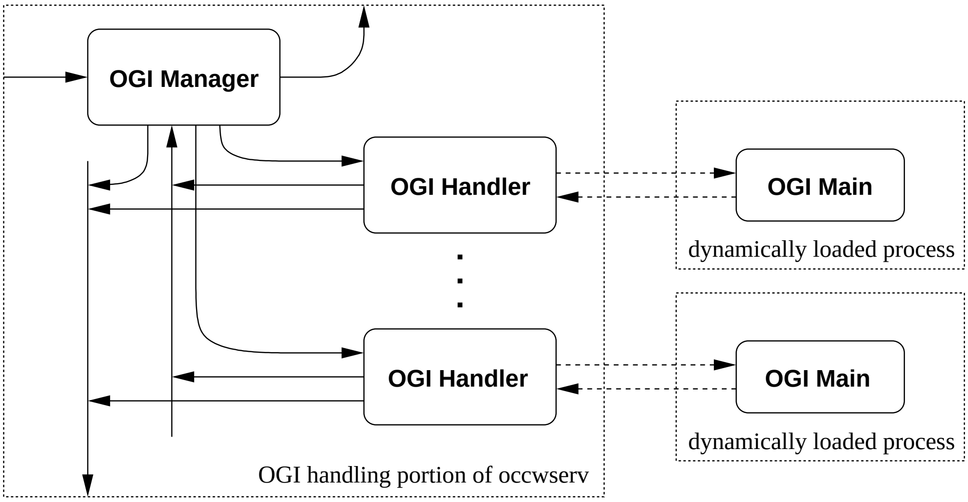


Figure 6.15: Example connectivity of dynamically loaded processes

In a similar way to other dynamic process extensions, mobilespace allocated for a dynamically loaded process must never be returned to the memory pool for re-use. The mobilespace for dynamically loaded processes is managed by the run-time kernel, which maintains a list of allocated mobilespaces. When a dynamic process is loaded, a hash-code for its mobilespace is generated. This is subsequently used to check for available mobilespaces, and to free the mobilespace to the correct list after use. The hash-code is formed using the library name, the process name and the memory requirements of the process. It is envisaged that this will be sufficiently unique to prevent the invalid recycling of mobilespaces.

6.5.3 Suspending and Resuming Dynamic Processes

One of the more interesting aspects of these dynamic processes is their ability to suspend themselves, and possibly be resumed later on. Futhermore, once suspended, the state of the process can be saved to disk, duplicated, or communicated over a network, for example. Correspondingly, a dynamic process may be initialised from a previously saved, assigned or received state.

To suspend, a dynamic process calls the '`ccsp.suspendproc`' procedure, that takes the process's own identifier (passed as a parameter into the launched `PROC`), and a integer used to indicate the result. This result will either be the constant '`DPROCESS.ACTIVE`', if another part of the dynamic process is active (on the run-queue, timer-queue, or is waiting for the completion of a blocking system call); or '`DPROCESS.RESUMED`' if the process suspends successfully and is later resumed. Of course, from the process' point of view, '`ccsp.suspendproc`' may never return — if the process is subsequently destroyed (or 'forgotten').

Therefore, dynamic loadable processes that intend to be suspended and possibly later resumed, potentially after duplication or migration, must be designed with this in mind — whilst the application may spin calling 'ccsp.suspendproc' (and looping when 'DPROCESS.ACTIVE' is returned), this is not good programming practice. A dynamically loaded network should (ideally) arrange for itself to be completely blocked *internally* (that includes the top-level channels), before attempting to suspend itself.

A simple example of suspending and resuming dynamic processes is provided in the KRoC distribution — 'testsuspend.occ', with the dynamic process in 'suspendable.occ'. The dynamic process, 'test.process', is essentially a simple clock that generates BOOL ticks at a specified (and controllable) rate. The protocol providing control over the process is defined as:

```
PROTOCOL CLOCK.PROTO
  CASE
    reset; INT
    stop
    suspend
    bounds
    overflow
:
```

The (purely sequential) example process is connected through three-channels, recovered (from the 'ANY' interface) with the following:

```
PROC test.process (VAL DPROCESS me, []CHAN ANY in?, out!)
  CHAN CLOCK.PROTO in? RETYPES in[0]?:
  CHAN BOOL sync.out! RETYPES out[1]!:
  CHAN BYTE out! RETYPES out[0]!:
  ...  process using "in?", "out!" and "sync.out!"
:
```

The re-typing of these channels replaces the names 'in' and 'out', that is quite sensible in this example — since ANY channels cannot be used for communication directly. The 'sync.out!' channel it *not* the tick — it is used to synchronise with the external network (merely for application convenience). The 'BYTE' channel 'out!' is multiplexed to the screen, and is where the clock reports tick events (as well as other diagnostic information).

The clock process ('test.process') initially synchronises on 'sync.out!', before printing a simple 'hello' message and entering its (terminating) main-loop. After initialisation, the clock is silent, waiting for an initial 'reset'. Once sent, the clock begins ticking at the rate (in microseconds) specified by the 'INT'. Once it has started ticking, the clock only stops when it is terminated with 'stop' — from the clock's point of view anyway. When it receives the 'suspend' message, 'test.process' calls 'ccsp.suspendproc', and if resumed continues ticking. For this example process, 'DPROCESS.ACTIVE' should never be returned — since it consists entirely of ALTing sequential code. The occam code for this handling of suspend and resume is simple:

```
INT result:
SEQ
  ccsp.suspendproc (me, result)
  IF
    result = DPROCESS.RESUMED
      sync.out ! TRUE
    TRUE
      -- error, should never happen
      STOP
```

When suspended, the process that called 'ccsp.runproc' resumes, with a result indicating why the dynamic process finished — either it terminated, faulted, or was suspended. When suspended, the test-rig allows the *state* of a dynamic process to be written out to a file, and conversely, have the suspended state *replaced* by loading it from a file. This functionality is provided through two (somewhat specialised) occam PROCs:

```
PROC ccsp.writeproc (DPROCESS p, VAL []BYTE filename, RESULT INT result)
PROC ccsp.readproc (DPROCESS p, VAL []BYTE filename, RESULT INT result)
```

The 'DPROCESS' passed to 'ccsp.readproc' must be an already initialised dynamic process, and the same process that was used with 'ccsp.writeproc' — i.e. 'ccsp.readproc' *will not* load the dynamic process itself, only the saved state. Thus any application wishing to use this functionality in depth must arrange for code (the dynamic process' '.so' file) to be available where necessary. Along similar lines, but providing a slightly different technique for saving and restore dynamic process states, are the PROCs:

```
PROC ccsp.sizeproc (DPROCESS p, RESULT INT bytes)
PROC ccsp.savestate (DPROCESS p, []BYTE data, RESULT INT result)
PROC ccsp.loadstate (DPROCESS p, VAL []BYTE data, RESULT INT result)
```

These allow the state of a dynamic process to be managed by the application. Once turned into a BYTE array using 'ccsp.savestate', an application could communicate this over a TCP/IP network, save the state to disk, or duplicate the state.

6.5.4 Implementing Suspend and Resume

In general, the implementation of dynamic processes is non-trivial, and this is definitely true for suspend and resume support. The *state* of an occam process is contained entirely within its workspace, vectorspace and mobilespace. For the majority of occam processes, this state is entirely internal — i.e. processes that do not hold references to memory external to their local state. Whilst this applies to standard occam, many of the extensions presented so far damage this in some way. In particular dynamic-mobiles — including dynamic mobile arrays, mobile channel-ends, and slightly speculatively, mobile processes. Furthermore, a dynamic process may hold references to other dynamic processes that it loaded and is responsible for.

In the absence of non-local pointers, the workspace and vectorspace of a dynamic process consist entirely of data and pointers to other (local) memory regions (channels containing blocked processes, for example). To correctly save and restore these memories, pointers must first be reduced to relative offsets and recorded separately when saving, to be restored later (which will most likely be at a different memory address).

In the first implementation of process suspension, pointers were located by simply scanning the memory of the dynamic process and reducing/recording those values that appeared to be valid pointers. However, although this approach works, it has a potentially serious deficiency — if the process has values in its memories that look like pointers, these will be reduced and recorded too. Thus when the process is resumed, at different memory locations, these non-pointer values will be changed (incorrectly). The solution, that also permits the saving and restoring of mobilespace (not currently supported), is to generate a "workspace map" of the process, indicating where pointers reside and the type they point to (for example, channel-word, or dynamic-mobile).

The workspace maps generated can be used by the run-time system to appropriately suspend and restore regions of workspace that contain pointers.

Generating Workspace Maps

The compiler has been slightly modified to generate workspace layout information, on a `PROC`-by-`PROC` basis, when a particular command-line flag is given. Before the code for a `PROC` is generated, its workspace map (of pointer objects, such as channels and dynamic mobiles), is generated as an ETC special (in a similar form to mobilespace maps). Broadly speaking, this extra information does not affect, in any way, the occam code being compiled.

When compiling occam in this 'suspendable' mode, the allocation of objects in workspace (and vectorspace) is adjusted so that memory used for pointers is never re-used for non-pointers, or pointers of a substantially different nature. While this may mean a larger memory footprint for occam processes, it does guarantee the correct suspending and resuming of pointers in a process's workspace. In some cases, the effect of the modified workspace allocation may be dramatic — for example, when sequential code is unable to re-use workspace because execution may be suspended whilst in *any* of those sequential processes.

In practice, the effect of this can be significant enough to discourage suspending this way, and have the application implement it by other means. This could easily be done by having the dynamic process network able to save and restore its own state from a file (using the provided occam file library – '`filelib`').

Figure 6.16 shows an example `PROC` and its corresponding workspace map.

```
PROC foo (CHAN INT out!)
  MOBILE []BYTE data:
  [10]CHAN INT c:
  SEQ
    ...  body of process
:
```

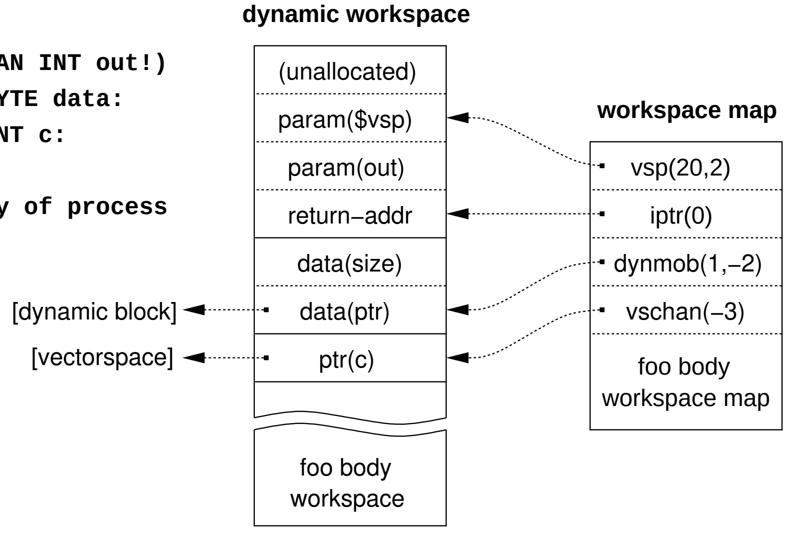


Figure 6.16: Dynamic process workspace map example

6.6 CPU Timer Support

This section presents a relatively small but significant extension to the KRoC/Linux system, designed for the Intel Pentium (and above) architectures, that significantly improves the handling and resolution of occam `TIMER`s.

Given that the various versions of KRoC are generally targetted for UNIX/POSIX based systems, the only sensible (and portable) timer and timeout implementations available are those provided by the POSIX standard. POSIX defines various timer functions, namely '`alarm`', '`setitimer`' and '`getitimer`', as well as the signalling mechanisms that these use. The mechanism at the UNIX level is simple — set a timeout (usually in seconds and microseconds), and *at least* that time later, a signal is delivered to the process (usually '`SIGALRM`' — alarm clock).

Although this mechanism is fully functional, it suffers from large overheads (from the system-calls and signal delivery) and possible inaccuracies (dependent on the underlying UNIX timer resolution — POSIX specifies that alarm signals should be delivered on or after their timeout [Int96]).

Intel Pentium and later processors contain a cycle-counter, set to zero on CPU reset. This increments at the CPU core clock frequency — effectively just a trivial 64-bit counter, incremented by the clock signal and loaded into the ALU when required. The '`rdtsc`' instruction is used to load this value into two processor registers, and is typically accurate to within a few cycles (instruction re-ordering inside the CPU affects the actual time of the load — avoidable if necessary by serialising execution around that instruction).

The motivation for using CPU timers is discussed in section 6.6.1, along with some background on previous support for these. The implementation, mostly concerned with occam's '`TIMER`' and '`AFTER`' keywords, is covered in section 6.6.2. By current CPU standards, a micro-second is rather a long time (possibly representing several thousand instructions[15]). In light of this, details of a 64-bit timer interface for occam are covered in section 6.6.3, although this has not been implemented.

6.6.1 Motivation

The motivation for using CPU timers is simple — reading the time via '`rdtsc`' is significantly cheaper than making a kernel-call that does the same job[16]. And if timer checking can be made extremely low-cost, the use of POSIX signals for timeout handling can be avoided (since these have a significant run-time cost).

Using the CPU cycle-counter for application timing is not a new idea. MESH [BDv99a] uses the CPU cycle-counter (and busy-waits) to perform timing, as did the C version of CCSP, before that support was removed (since they were both formed from a common core). The handling of CPU timers in these is not particularly suited to occam, where a micro-second, 32-bit time value is more suitable.

An overhead always traditionally incurred by KRoC is that of '`SIGALRM`' handling, used to implement the timer-queue and whose signal-handler sets one of the synchronisation flags (shown in figure 6.3 on page 149). Setting the synchronisation flag has a neglible cost, in comparison to that of (Linux) kernel signal delivery mechanisms. Additionally, an existing alarm may have to be cancelled and a new one set, when processes are either added to or removed from the start of the timer-queue (that is ordered by time, soonest first). When processes are being frequently rescheduled on the timer-queue (repeating time-out `ALT`s, for example), this overhead becomes noticable — due directly to the increased frequency of (OS) kernel interaction.

[15]In theory, a 3.06 Giga-Hertz processor (as is currently the fastest available), can process a little over three thousand instructions per microsecond. However, given the relative speeds of memory and I/O devices, this is typically not the case. A processor will often spend much of its time idling, for instance while waiting for memory to be loaded into the cache. As core frequency speeds have increased, much effort has gone into processor design in an attempt to minimise these delays, including predictive branching, extensive instruction re-ordering (at the micro-code level) and a relatively deep memory-write pipeline.

[16]A sizeable amount of effort has gone into Linux, as far as this is concerned. Typically, calls to '`gettimeofday()`' now perform no kernel call and read values from memory directly — memory that visible to both the kernel (read/write) and user-land processes (read-only).

6.6.2 Implementation

In KRoC, the occam TIMER is a signed 32-bit wrapping counter that counts microseconds, giving a maximum delay period of approximately 35 minutes. The CPU cycle-counter increments at the core clock frequency, which is generally unsuitable for applications. To calculate the CPU frequency, a mixture of 'rdtsc' and kernel time-reading calls is required. Obtaining an accurate result requires several seconds of calibration time, so KRoC provides a small utility that does this, placing the CPU frequency (MHz) in a file which is later read by the run-time system. The code that performs the calibration is derived from similar code in MESH, authored by Marcel Boosten [BDv99a]. More recent Linux kernels provide the CPU frequency as part of the information in the 'proc' file-system. If this is available the calibration program simply reads the value, rather than performing the calibration.

At run-time, the 64-bit value read using 'rdtsc' must be convered into a 32-bit occam TIMER value. A technique suggested by David Wood has been used to do this: (floating-point) divide 2^{32} by the CPU frequency in MHz and store the result as a 32-bit integer. To convert the 64-bit value, multiply it by the stored 32-bit integer and take the top 32-bits of the result. This calculation, including the 'rdtsc' instruction, can be performed relatively quickly (in tens of clock cycles).

The integer multiply instruction on the i386 multiplies two 32-bit quantites to give a 64-bit one (in two 32-bit registers), so the conversion is performed using two multiply instructions, an add and a few moves, before placing the resulting value in memory (typically an occam INT variable).

For complete support of TIMERs in occam, implementations (such as KRoC) need to provide support for 5 instructions: 'LDTIMER' to read the current time; 'TIN' to wait until a specified time; and 'ENBT', 'DIST' and 'TALTWT' to support timer ALTs. By default, these instructions simply translate to run-time kernel calls, passing arguments and results as appropriate. The "inline-timers" flag to tranx86 causes full inlining of 'LDTIMER', and 'ENBT' (as well as the associated extended behaviours for enhanced ALT enabling and ALT pre-enabling, as discussed in section 3.9.3); and partial inlining of 'TIN'.

Timer Loads

The implementation of the 'LDTIMER' instruction, when inlining is enabled, is as follows (leaving the result in the 'EAX' register):

```
rdtsc                       // read time into edx:eax
movl    %edx, %ebx          // save high 32-bits
mull    glob_cpufactor      // multiply low 32-bits by "glob_cpufactor",
                            //     high 32-bits left in edx
movl    %ebx, %eax          // load saved high 32-bits
movl    %edx, %ebx          // save high 32-bits of last multiply
mull    glob_cpufactor      // multiply previously saved bits by "glob_cpufactor"
addl    %ebx, %eax          // add high 32-bits of previous result,
                            //     leaving the whole result in "eax"

movl    %eax, 4(%ebp)       // save in local variable
```

A reasonable amount of effort has gone into making this highly efficient, with effectively the same code inside the run-time kernel when inlining is not enabled. The memory location referenced by 'glob_cpufactor' contains the earlier computed multiply factor.

When inlined, this sequence of instructions takes approxiately 47ns on an 800 MHz P3 (< 38 cycles). When called inside the run-time kernel, it takes approximately 71ns (< 57 cycles).

Timer Waits

The 'TIN' instruction waits until a specified time, or if that time has already passed, returns immediately. The existing algorithm, based on similar code in the Sparc KRoC [WW96, Woo97], first checks to see if the time has expired, and if so, does nothing. Otherwise the current process is added, in time-order, to the timer queue and a reschedule takes place.

Waiting for a time that has already past takes approximately 75ns on an 800 MHz P3 when using CPU timers, implemented in the "X_tin" kernel entry-point. Inlining the whole of this instruction, to improve efficiency, is not really an option — given the relatively complex operation of timer-queue manipulation. Instead, the case where the time has already passed is inlined, such that the run-time kernel is only entered if the process needs to wait. With this inlining enabled, the cost of waiting for a time already past is reduced to around 50 ns, comparable with the earlier timer-read inlining.

The inlined part of the 'TIN' instruction (when enabled with the '-iT' command-line flag), simply reads the current time and checks to see if the timeout (placed in 'Areg') has passed. The code generated for this is similar to the earlier code for an inlined timer read, for example, with a target time in 'Areg':

```
    rdtsc
    movl    %edx, %ecx
    mull    glob_cpufactor
    movl    %ecx, %eax
    movl    %edx, %ecx
    mull    glob_cpufactor
    addl    %ecx, %eax          // leaves "now" in %eax

    subl    %Areg, %eax         // subtract target time
    jns     0f                  // jump if time passed

    ...  add to timer queue and deschedule
0:
```

When the jump is taken, this code takes approximately 50 ns to execute. Efficiency is paramount here since similar code will be used to check the timer-queue for ready processes whenever the run-time kernel reschedules (in order to avoid the overheads incurred by a signalling timer implementation).

6.6.3 64-Bit Timers

The standard occam timer counts in micro-seconds. However, the resolution of the CPU timer is much higher — approximately the clock speed. Given that most modern PCs now have clock periods of *less* than a nano-second, the micro-second timer suddenly seems rather outdated.

The use of 32-bit integers to manpulate time in occam partially restricts the maximum resolution of the TIMER, if the timer period is not to become too small to implement useful delays. For instance, with nano-second resolution for TIMERs, the maximum timeout available using a 32-bit integer is a little over 2.1 seconds, that might be insufficient for many applications.

To overcome these limitations, it is proposed that a 'TIMER64' type be added to occam, only to be used with 64-bit integers ('INT64' in occam). The idea was initially suggested by David Wood and Peter Welch, after discussions about the relative length of a 'modern' micro-second. A timer this wide can support both a high resolution and acceptable timeout periods. If pico-second (10^{-12}) resolution is used, the maximum timeout period is effectively 106 days — more than sufficient. A nano-second `TIMER64` would allow a maximum timeout of almost 3 centuries — probably more than sufficient for most software requirements!

In the interest of efficiency in the implementation, the most appropriate `TIMER64` resolution might be the one where the top 32-bits increment once every second, and the lower 32-bits are scaled appropriately. The accuracy of such a timer would depend on the underlying CPU speed, and on any instruction re-ordering around timer reads.

6.6.4 Optimising Process Timeouts

This section proposes a modification to the way in which process timeouts are handled, that could potentially improve the performance of some applications.

When CPU timers are enabled and there are processes waiting on the timer queue, the timeout is checked whenever a reschedule takes place. Although the overhead for this is small (around 40 CPU cycles on a P3), it significantly impacts the context-switch time — around 70 to 110 nano-seconds ordinarily (depending on optimisations). For much of the time this overhead is too small to be of any significance, until the parallel granularity of a system causes the scheduling overhead to become significant (on the 800 MHz P3 on which most times here are measured, scheduling overheads would account for 50% of the CPU time at approximately 3–4 *million* process schedules per second).

In cases where an application makes few, distant timeouts, the impact on scheduling due to the usage of CPU timers is an unwanted burden. Indeed, for these cases, the aggregate overheads of using a signalling timer implementation would likely be smaller than the frequent, but small, overhead incurred by active CPU timers.

The solution proposed here is to use two timer queues; one for short-term timeouts that can be checked on reschedule; and one for long-term timeouts, that can use the kernel timeout signalling mechanism. This is shown in figure 6.17.

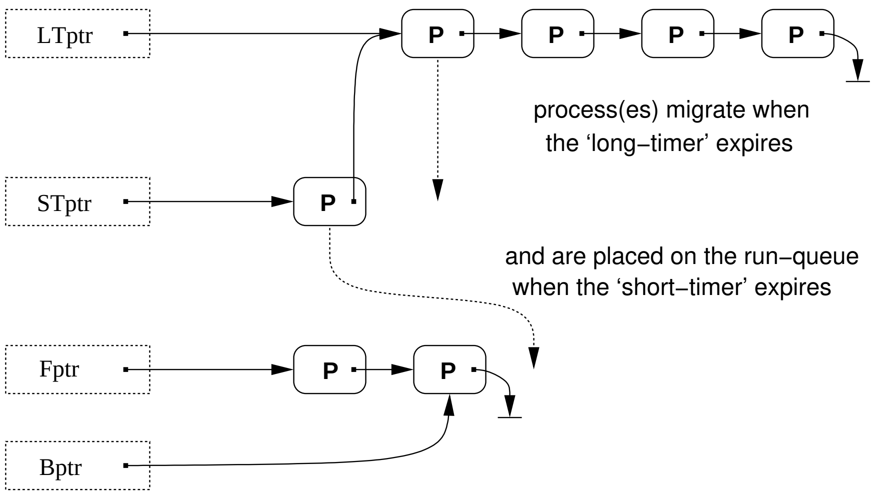


Figure 6.17: Using two timer queues to implement timeouts

The queues are maintained such that the last process in ‘*STptr*’ (if present) links to the first process in ‘*LTptr*’ (or null). This allows all waiting processes to be scanned in a single-loop, starting at *STptr*. It also optimises the process of migrating a process from *LTptr* to *STptr*, which then simply involves moving *LTptr* along.

A constant time value determines the 'lifetime' for any process on *STptr*. The exact value used depends largely on the speed of the system — and could be calculated at run-time from the CPU speed. For fast systems, the overhead of polling the first process on *STptr* is substantially decreased — it is a short series of highly localised instructions. The overhead of kernel calls (to handle signalled timeouts) does not decrease at nearly the same rate, with respect to processor speed. Thus, for fast systems, better overall throughput can be obtained by allowing processes to live on *STptr* for a long time (e.g. 200 ms). As CPU speed decreases, the overhead of polling *STptr* increases. Thus, it is more favourable to keep processes on *LTptr* — in an attempt to keep *STptr* empty.

Currently, this is an entirely theoretical implementation, but one with good prospects. With a real implementation, the effects of different *STptr* lifetimes on different types of application could be examined. Given appropriate care in such an implementation, the difference in overheads between the existing CPU-timer implementation and this implementation, with a maximum lifetime for *STptr*, should be minimal. Manipulation of the (OS) timer underlying *LTptr* can often be deferred, to be handled by the scheduler when it sees an appropriate synchronisation flag.

6.7 Direct OS Kernel Support for CSP

One of the useful features about open-source operating-systems, particularly Linux, is the ability to easily create kernel modifications for performing particular tasks. The philosophy of doing this in the first place is somewhat debatable — on one hand it allows for significant performance improvements for certain programs (the Linux-kernel web-server and NFS-server for example, the latter also having user-land help, for authentication and other tasks which are more suited to a user-land implementation). On the other hand however, there are issues about recompiling kernels, and the level to which someone else's code may be trusted (if that code is not included in the operating-system by default — possibly because it was rejected by the community).

This section describes the CSP driver — a Linux kernel *device driver* that provides support for named channels (that can be used as a mechanism for inter-process communication (IPC), between distinct occam programs[17]), described in section 6.7.1.

The CSP driver provides specific support for KRoC, making easier the task of pausing safely (traditionally performed by the '`safe_pause()`' function in KRoC [WW96]), and reducing the complexity and overheads of handling blocking system-calls (section 6.3). These are described in sections 6.7.2 and 6.7.3 respectively.

The choice of implementing this functionality in a device-driver, as opposed to a more core-kernel component such as a new system-call, is two-fold. Firstly, the vast collection of available device drivers[18] means, inevitably, that some will contain bugs and may crash at run-time. Linux has evolved to handle these, by reporting the error and shutting down the driver, leaving the kernel as a whole un-harmed. The faulty device driver, however, will likely be unusable from then on, cleanable only with a reboot. Secondly, there is a large body of literature available regarding device-driver authoring for Linux, mostly in on-line documents and in the driver sources themselves, but also in several books, [RC01, Kha02] and others.

[17] The CSP driver can also be used with C programs — threaded or otherwise

[18] Several hundred device drivers in the standard kernel sources, and numerous others are available elsewhere — some provided in binary-only format for licensing reasons.

6.7.1 Named Channels for Inter-Process Communication

One aspect of communication lacking from occam is the ability to easily communicate with other occam *programs* on the same system. Existing UNIX mechanisms for IPC [Ste92, Ste98] cover a whole range of methods: pipes, UNIX sockets, named pipes (FIFOs), terminal master/slave pairs, signals, semaphores and shared memory. POSIX threads [Int96, But97] add another layer of IPC mechanisms, but generally specialised (and sometimes restricted) for use within same-memory processes (the threads). None of these *directly* support the abstraction of a simple occam channel — which is a synchronised point-to-point unbuffered communications link. They can, however, be used to implement the channel semantics[19]. The original KRoC used a pair of pipes to implement a channel between the keyboard process and run-time kernel (discussed in section 6.3.9). A POSIX threads based implementation of CSP support for C, by Beton [Bet00], uses thread synchroniation mechanisms to provide channel support.

The CSP-driver for Linux simplifies this whole issue, by providing direct support for *named channels*. These channels allow distinct OS-level processes to communicate and synchronise with CSP semantics. In particular, they allow separate occam programs to communicate directly — when combined with the user-defined channel support[20] (section 6.4).

One percieved use for this mechanism is to allow remote control of the occam web-server, through named channels. Remote monitoring processes (to watch connections real-time, for example) could also be attached. There is even the possibility of attaching to the web-server for the purpose of communicating client connections. However, this would involve the communication of file-descriptors between UNIX processes, a non-trivial (but possible) operation. Implementing support for this within the Linux CSP-driver is a possibility, but has not been investigated (it is unclear whether the effort would justify the result).

After a standard '`open()`' call on the device (currently '`/dev/cspdrv`'), the CSP driver is accessed using the IOCTL calls shown in Table 6.6. These implement all the support required for channels, including `ALT`s.

Constant	**Value**	**Parameter**
`IOCTL_GET_VERSION`	0x1000	`char *`
`IOCTL_GET_CHANNEL`	0x1001	`ucsp_chaninfo *`
`IOCTL_TYPE_CHANNEL`	0x1002	`ucsp_chaninfo *`
`IOCTL_RELEASE_CHANNEL`	0x1003	`ucsp_chaninfo *`
`IOCTL_READ_CHANNEL`	0x1010	`ucsp_datainfo *`
`IOCTL_WRITE_CHANNEL`	0x1011	`ucsp_datainfo *`
`IOCTL_ENABLE_CHANNEL`	0x1012	`ucsp_altinfo *`
`IOCTL_DISABLE_CHANNEL`	0x1013	`ucsp_altinfo *`
`IOCTL_ALTWT`	0x1014	`ucsp_altinfo *`

Table 6.6: Channel IOCTL calls for the Linux CSP-driver

The '`GET_VERSION`' ioctl is used to query the version of the device-driver. If parts of the interface change in the future, this will allow programs to automatically select the correct interface, based on version. The '`char*`' parameter passed is a pointer to a string (of at least 32 bytes) where the version information is stored.

[19] Conversely, the behaviour of existing IPC mechanisms can easily be modelled in occam/CSP.

[20] The user-defined channel library required to interface with the CSP driver is now part of the standard KRoC/Linux distribution, and provided as part of the blocking system-calls library (since it relies significantly on that functionality), called '`cspdrvlib`'.

Accessing Channels

The ioctl calls, 'GET_CHANNEL', 'TYPE_CHANNEL' and 'RELEASE_CHANNEL' provide access to a channel, and use the 'ucsp_chaninfo' structure, defined as:

```
typedef struct {
    char name[32];      // channel name for GET_CHANNEL
    int dir;            // direction for GET_CHANNEL, hashcode for TYPE_CHANNEL
    int chanid;         // returned channel-id for GET_CHANNEL, param for others
} ucsp_chaninfo;
```

To access a channel, a process puts the desired channel name in 'name', direction in 'dir' and calls the 'GET_CHANNEL' ioctl. If the channel does not exist, it is created. The direction field ('dir') determines how a process will use the channel, and possibly how it interacts with others. Table 6.7 shows the various constants defined for the direction field.

Constant	Description
CSP_CHANNEL_READER	Use channel for unshared reading
CSP_CHANNEL_WRITER	Use channel for unshared writing
CSP_CHANNEL_ANY_READER	Use channel for shared reading (..-to-any)
CSP_CHANNEL_ANY_WRITER	Use channel for shared writing (any-to-..)
CSP_CHANNEL_MANY_READER	Use channel for synchronised reading (..-to-many)
CSP_CHANNEL_MANY_WRITER	Use channel for synchronised writing (many-to-..)

Table 6.7: Channel direction constants for the CSP-driver

If there is any compatibility error when 'getting' the channel, the ioctl call fails. For example, if one process aquires a channel with the flag 'CSP_CHANNEL_READER', any attempt by another process to access the same channel in *any* read mode will result in a error (the first reader is exclusive).

Before a channel can be used for communication, the 'IOCTL_TYPE_CHANNEL' ioctl must be called passing an appropriately unique value. For occam programs, this will be the 'PROTOCOL.HASH' of an occam channel. When channels are initially created inside the driver their type is unknown. The first process to call 'IOCTL_TYPE_CHANNEL' sets the type of the channel, for the entire lifetime of that channel (that may continue to exist after the process which created it terminates). Any subsequent call to 'IOCTL_TYPE_CHANNEL' is simply checked against the stored value.

To release a channel, a process fills a 'ucsp_chaninfo' structure (only the 'chanid' field) and calls ioctl on the device with 'IOCTL_RELEASE_CHANNEL'. The channel is only destroyed (inside the driver) if this was the last process using it. Forgetting to call this ioctl before program exit is not fatal — when the device is 'close()'d (which happens automatically if the program exited with it open), any 'accessed' channels are released (and destroyed if necessary).

Reading and Writing

The ioctl calls 'IOCTL_READ_CHANNEL' and 'IOCTL_WRITE_CHANNEL' implement basic communication. Both of these use the 'ucsp_datainfo' structure, defined as:

```
typedef struct {
    int chanid;         // channel identifier
    char *data;         // pointer to data to read/write
    int length;         // size of communication (must match)
} ucsp_datainfo;
```

The driver enforces CSP semantics on the use of these ioctl calls. When one process calls 'IOCTL_READ_CHANNEL', it will remain blocked until another process calls 'IOCTL_WRITE_CHANNEL' (assuming a non-'many' communication).

The application simply sets this structure up with the appropriate 'chanid', 'data' (to read or write) and 'length' (in bytes), then calls the relevant ioctl. The driver checks for same-sized communication and gives an error to both if incorrect. The way the driver works means that if one communicating process aborts (crashes, for instance), the other will be left 'stuck'. If this happens during sequential protocol communication (that is implemented as many individual communications), the next attempt at communication (from a new or restarted occam program) will likely result in an error.

Alting on Named Channels

Support for ALTing is provided by the 'IOCTL_ENABLE_CHANNEL' and 'IOCTL_DISABLE_CHANNEL' ioctl calls. These use the 'ucsp_altinfo' structure, defined as:

```
typedef struct {
    int chanid;         // channel identifier
    int ready;          // channel ready (returned)
} ucsp_altinfo;
```

The underlying implementation of these 'enable' and 'disable' operations is different to that of the normal transputer. The 'enable' operation only tests a channel for ready-ness, without leaving a reference to the invoking process in the channel. Instead, the 'IOCTL_ALTWT' ioctl call is used, which performs the blocking part of the ALT.

Things are done this way primarily to fit with the user-defined channel mechanism (section 6.4). When the enabling function returns with a not-ready indication, a blocking system-call is dispatched to perform the 'IOCTL_ALTWT' operation for that particular channel. This blocking call either finishes normally (by a remote party attempting communication), or it is terminated from within the user-defined channel disabling code (as shown in figure 6.14 on page 179).

For C processes, the 'select()' system-call can be used on the CSP-driver file descriptor. This provides a mechanism for ALTing over *all* the channels in use by a particular process. Channels accessed for reading (input) are ALTed on from the 'select' read-set. When used in the write-set, channels accessed for writing (output) are ALTed on. This creates a possibility for false synchronisations — where two (or more) processes 'select()', and 'wake-up' as a result of each others offers to communicate, but where one or more of the processes fail to engage in that particular communication.

Ordinarily, this could result in a process being blocked in communication (from which the remote party has withdrawn), but where the real intention is to ALT. Unfortunately there is no neat solution to this, for C processes at least. Either the application must guarantee to only perform ALTing one-way (which could be on outputs) — and match that behaviour with remote processes — or explicitly handle failed synchronisations.

To aid the latter, a process may set the non-blocking option on the CSP-driver descriptor. Instead of blocking, read and write ioctl operations will return the standard 'EWOULDBLOCK' error ('EAGAIN' on some systems). This should be used cautiously however, since two C processes in non-blocking mode will never be able to communicate. If an application requires a mixture of blocking/non-blocking behaviours, it must open the CSP-driver device twice, with the non-blocking option set on one.

6.7.2 Support for Sleeping

Currently, periods of idleness in KRoC are handled by the 'safe_pause' function. This simply blocks the run-time system in the OS kernel until an interrupt is received — typically for either a timeout or a completed blocking system-call.

To ensure the safe delivery of signals, the run-time kernel must prevent the delivery of signals as it goes to sleep, but allow signals to be delivered whilst it is sleeping. This is implemented in KRoC [WW96] through the use of the POSIX 'sigprocmask()' and 'sigsuspend()' system-calls. After the run-time system has woken up, the signal mask must be restored by calling 'sigprocmask()' again.

The end result is a cost of two or three system-calls — incurred even if the kernel does not sleep (a signal might be delivered just as the run-time system enters 'safe_pause', for example).

By providing this support within the CSP-driver, the overhead can be reduced to a single system-call (to the CSP-driver), that automatically takes care of any aynchronous signalling. Two IOCTL calls are used to perform 'sleeping' actions, shown in table 6.8.

Constant	Value	Parameter
IOCTL_SAFE_PAUSE	0x1020	ucsp_waitinfo *
IOCTL_REALTIME_PAUSE	0x1021	ucsp_waitinfo *

Table 6.8: Timeout IOCTL calls for the Linux CSP-driver

At a basic level, these two IOCTL calls just put the run-time kernel to sleep, until a signal is delivered. When CPU timers (section 6.6) are in use, the signalling timeout mechanism is only used during safe-pause — that also incurs a slight additional overhead for setting up and possibly cancelling the timer. The CSP-driver IOCTLs for sleeping accommodate this, by allowing a timeout to be specified in the passed 'ucsp_waitinfo' structure:

```
typedef struct {
    int *sf_addr;           // sync-flag address
    unsigned long timeout;  // timeout in micro-seconds
} ucsp_waitinfo;
```

The 'sf_addr' field is set to the address of the KRoC synch-word, allowing the kernel to check for events before sleeping. The kernel has the advantage that it is non-preemptable, thus no signals will be delivered whilst executing inside the CSP-driver.

If a timeout is required, this is placed in the 'timeout' field (in micro-seconds). Otherwise, the timeout field is set to zero. The 'SAFE_PAUSE' IOCTL uses the standard kernel timing mechanism (that has a resolution of approximately 100 Hz). The 'REALTIME_PAUSE' IOCTL uses a combination of this and busy-waiting, in an attempt to produce significantly more accurate timeouts where required.

Figure 6.18 shows the results from a simple test program that measures the actual time required for a delay ('AFTER') timeout.

6.7.3 Support for Blocking System Calls

One of the major overheads in the current KRoC/Linux is that of blocking system-calls. The round-trip time for a blocking call, using the standard implementation, is approximately 9 μs on an

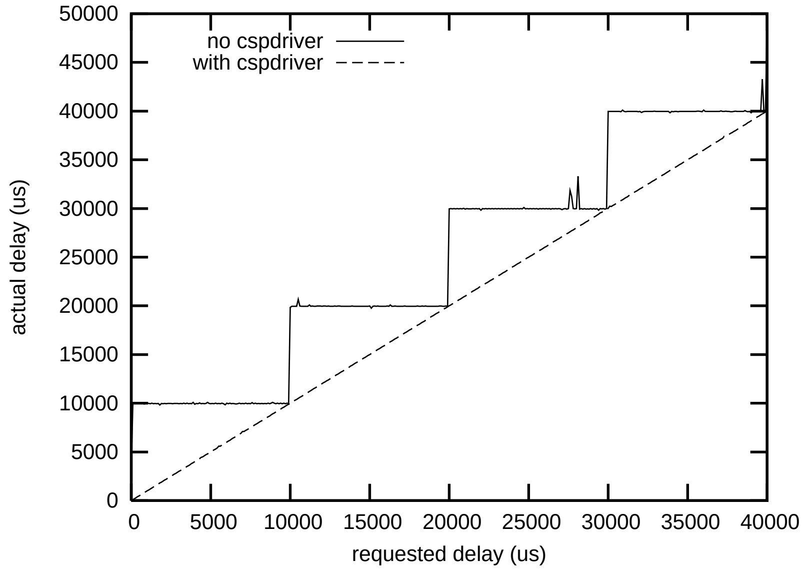


Figure 6.18: KRoC timeouts with and without kernel support

800 MHz P3. Much of this can be attributed to the semaphore and signalling operations that the blocking 'clone' processes use to synchronise with the main kernel thread.

Use of the CSP-driver for handling the synchronisation between the clones and the main KRoC kernel can significantly reduce this overhead. To handle this, another set of IOCTL calls have been added to the driver, shown in table 6.9.

Constant	Value	Parameter
`IOCTL_BSC_SETUP`	0x1030	`uscp_bsciinfo *`
`IOCTL_BSC_CALLIN`	0x1031	`ucsp_bscdinfo *`
`IOCTL_BSC_DISPATCH`	0x1032	`ucsp_bscdinfo *`
`IOCTL_BSC_GET_BUFCOUNT`	0x1033	`int *`

Table 6.9: Blocking system-call IOCTL calls for the Linux CSP-driver

The increased level of 'multi-process' control (in the CSP driver) enables blocking-call 'dispatches' to be *queued* by the driver. In the user-space implementation of blocking system-calls, a request can only be dispatched if the previous one had been taken (locked by a form of spin-lock that reschedules if the previous blocking-call had not been dispatched).

Before the CSP driver can be used to handle blocking system-calls, it must be suitably initialised. This is done using the '`BSC_SETUP`' and '`GET_BUFCOUNT`' IOCTLs. The '`BSC_SETUP`' call uses the '`ucsp_bsciinfo`' structure, defined as:

```
typedef struct {
    unsigned int *sf_addr;  // sync-flag address
    unsigned int bsc_bits;  // blocking flag bits
    unsigned int bsc_mask;  // blocking flag mask
} ucsp_bsciinfo;
```

This provides the information that the kernel needs to set the KRoC blocking-call synch-flag asynchronously — i.e. without the use of a signal.

The two IOCTL calls that perform the dispatch and collection algorithms are '`BSC_CALLIN`' and '`BSC_DISPATCH`' respectively. Both of these use the '`ucsp_bscdinfo`' structure, defined as:

```
typedef struct {
    void *address;          // pointer to bscinfo (input/output)
    int length;             // size to communicate
    int resync;             // resync flag for clones
} ucsp_bscdinfo;
```

The '`address`' and '`length`' fields, in the main run-time kernel process, hold the address of the '`bsc_thread`' structure that holds the information for a particular blocking call (defined on page 156). Much of the information in the '`bsc_thread`' structure is used by the clone processes locally (to hold their own state). Therefore the '`length`' field is adjusted to only cover the information required for dispatching (the '`bsc_thread`' structure is (re)-organised to allow this).

A clone process collects a call using a similar process, but places the address of its own '`bsc_thread`' structure in the '`address`' field. If the clone had previously just completed a call, the '`resync`' field is set to non-zero. This works in conjunction with the spin-locked queue used to hold 'finished' calls (in terms of the occam processes making them) — the CSP-driver only provides the synchronisation (and communication) between the main kernel process and the blocking-call clone processeso

The obvious advantage to this method is that when a clone finishes and calls '`BSC_CALLIN`', if the run-time kernel is still sleeping (in '`SAFE_PAUSE`') then it can be immediately resumed to reschedule the previously blocking occam process — without the overhead of any signalling.

Chapter 7

Further Extensions

This chapter covers a number of extensions and enhancements that provide for: easy co-existence of C and occam (concurrently); process priority in occam; a mechanism for debugging run-time errors; and support for higher level channel and data interaction.

One feature which has become particularly desirable, partly as a consequence of other extensions, is the ability to have external (C) processes that can communicate with occam processes directly. The original CCSP supported both C and occam, but not concurrently. The run-time kernel was either compiled for occam or for C. By the time of the first KRoC/Linux 1.3.3-pre release, in August 2002, all support for the C version (including support for external channels) had been removed from CCSP. Section 7.1 describes the C-interface extension ('CIF'), which provides a limited CCSP-based C API for writing concurrent code in C, *and* allows that code to interact concurrently with occam processes. The run-time cost of interoperability rests solely on the C interface; occam processes suffer no additional costs as a result.

Section 7.2 describes the addition of process priority to occam. The Transputer provided two levels of process priority — high and low. The hardware engineering was such that a high-priority process could preempt a low-priority process, typically used for for handling hardware 'events' on the Transputer (via the event pins). To date, KRoC implementations have ignored this priority — quietly implementing '`PRI PAR`' as just '`PAR`'. The implementation of priority in KRoC/Linux provides 32 levels of process priority, but not through '`PRI PAR`' (although consideration is given to this).

Debugging occam programs has traditionally been difficult, although in the large, we can guarantee that all run-time errors[1] are trapped. However, when such an error occurs, the run-time kernel just prints a basic informational message and exits. Section 7.3 presents an implementation of post-mortem debugging (debugging after a run-time error has occurred, not the steps leading up to it), that takes many ideas, in particular the technique for deadlock debugging, from work by Wood for debugging in the Sparc version of KRoC [WB00].

Section 7.4 describes the support for a mechanism that allows communication to be 'extracted' and 'inserted' into occam programs. That is, support for the decoding and encoding of channels, allowing ordinary occam communication to be transported at a higher-level (over a TCP/IP based network, for example).

Finally, section 7.5 describes a pre-processor added to the occam compiler, a useful feature long

[1]This only extends to errors in occam programs. We cannot make guarantees about the safety of linked-in C code, or even the integrity of the run-time kernel itself. The occam compiler may also be a source of error, though run-time errors caused by this usually result in trappable conditions or CG-test failure.

missing from occam. Various pre-processors already exist for occam, but are somewhat detached from the compiler. The pre-processor described here is implemented in the lexical analyser (lexer) of the occam compiler, allowing finer points such as indentation to be taken into consideration.

7.1 Concurrent C and occam

The previous Chapter has presented extensions that allow occam and C code to interact in various ways. The one type of interaction *not* supported by these, however, is that of direct communication between occam and C *processes*. The 'CIF' (C interface) extension provides a mechanism for this. The implementation is somewhat obscure however — rather than directly interacting with the CCSP run-time kernel, the CIF extension disguises C processes as occam processes during scheduling and communication. This allows the run-time kernel to treat them as ordinary occam processes — it is, in fact, unable to tell the difference between the two — causing no additional overheads for existing occam programs.

The API provided to C processes is a subset of that provided with the original CCSP [Moo99, Moo00a], which is largely based on the traditional Inmos C API for programming the Transputer. This new iteration of the API does have some minor differences however — largely to provide support for the various occam extensions presented in Chapter 3. Support for blocking system calls (section 6.3) is also provided.

There are both good and bad reasons for wanting to write certain code in C, as opposed to occam. The ability to write C which can interact with occam makes easier the task of interacting with hardware at a low-level — possibly with architecture specific inline assembler. However, by using C, the automatic race-hazard and alias free benefits of occam are lost. Additionally, to avoid overheads for occam processes, C processes incur additional overheads masquerading as occam processes. These overheads are small however (in tens of nano-seconds/CPU-cycles).

Figure 7.1 shows an example network, in which the '`hdd.control`' process is probably more suited to a C implementation, rather than an occam one. The CIF extension is used in this way for certain hardware interaction in RMoX. Additionally, CIF provides a method for augmenting existing C code, allowing it to be interfaced with occam.

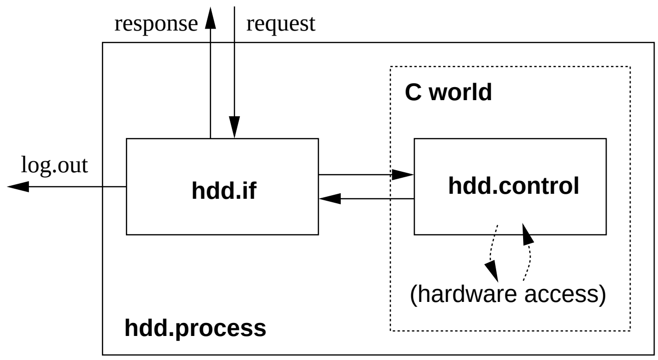


Figure 7.1: Process network for an example hard-disk interface

The implementation of this extension is done using a library of C functions, with basic support for calling this type of C process from occam. Only a minor modification has been made to the run-time kernel — to support blocking system calls. The creation and execution of C processes is described in section 7.1.1. The implementation of the C/occam interface is discussed in section 7.1.2.

7.1.1 Creating and Running C Processes

Concurrent C processes are created wholly in the C world, through the use of either the 'ProcAlloc' or 'ProcInit' library calls. To create a C process from occam, a small external C stub function must be used. This is due, in part, to the lack of a function-pointer type and variable-argument functionality in occam. For example, the code required to create a (serial) C version of the traditional 'integrate' process, is:

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

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

void real_create_integrate (int *raddr, Channel *in, Channel *out)
{
    Process *p = ProcAlloc (integrate, 1024, 2, in, out);

    *raddr = (int)p;
    return;
}

/* PROC create.integrate (INT raddr, CHAN INT in?, out!) */

void _create_integrate (int *ws)
{
    real_create_integrate ((int *)(ws[0]), (Channel *)(ws[1]), (Channel *)(ws[2]));
}
```

The 'real_create_integrate' function calls 'ProcAlloc' to create a new process, that starts at 'integrate' with 1024 bytes of C stack space, and passes the two channel parameters given. This is accessed by an occam program through the external declaration:

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

In the transition between occam and C, protocol information for the channel is lost. Because of this, the programmer must be particularly careful to use channels correctly, otherwise serious run-time errors may result. Another requirement on programmers is that they use the returned address *carefully* in occam— if the external function is called after some of its parameters go out of scope, undefined behaviour will result.

Inside the occam program, a special PROC ('cifccsp.startprocess(addr)') is called to run the process allocated in the C world. This simply schedules the C process for execution and waits for it to terminate. If desired, the C process can be run again, with the parameters unchanged, or optionally altered using the 'ProcParam()' C function.

The correct way to interface and run the above 'integrate' from an occam program would be:

```
#USE "course.lib"
#INCLUDE "cifccsp.inc"

PROC int.writer (CHAN INT in?, CHAN BYTE out!)
  WHILE TRUE
    INT v:
    SEQ
      in ? v
      out.int (v, 0, out!)
      out ! '*n'
:

PROC test (CHAN BYTE screen!)
  CHAN INT c, d:
  PAR
    numbers (c!)
    INT addr:
    SEQ
      C.create.integrate (addr, c?, d!)
      cifccsp.startprocess (addr)
    int.writer (d?, screen!)
:
```

The process network for this program is shown in figure 7.2.

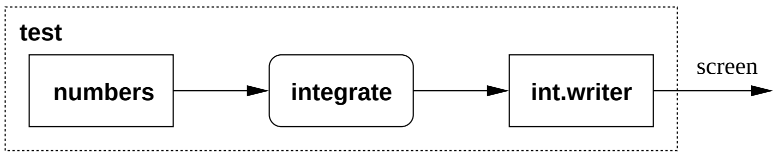


Figure 7.2: Example C integrate in an occam process network

7.1.2 Masquerading as occam

In order to be scheduled alongside other occam processes, a C process must pretend to be an occam process when entering the run-time kernel. Inside the '`ProcAlloc()`' function (part of CIF), a block of dynamic workspace is allocated for the C process, as well as a C stack. Figure 7.3 shows the layout of the allocated workspace.

The workspace slots from -6 to 2 inclusive provide the occam process state space required to perform timeouts, communication, `ALT`s and `PAR` constructs. The magic constant placed in slot 3 is used to check the integrity of the process, to ensure that the workspace has not accidently become corrupt (by an erroneous C process). Slots 4 to 6 are used for the C stack handling. Whilst the C process is running (in its own stack), the stack-pointer for the occam kernel is stored in workspace slot 6 ('CCSP SP'). When the C process is not running, its stack pointer is stored in slot 4. Slot 5 points at the base for the C process's stack, used for freeing it in '`ProcAllocClean()`'.

Workspace slots 8 and 9 are used as temporary locations to store the current run-queue front and back pointers whilst the C process is running. For occam processes, these values are kept in processor registers. Slot 10 is used to save the workspace-pointer for the occam process that called

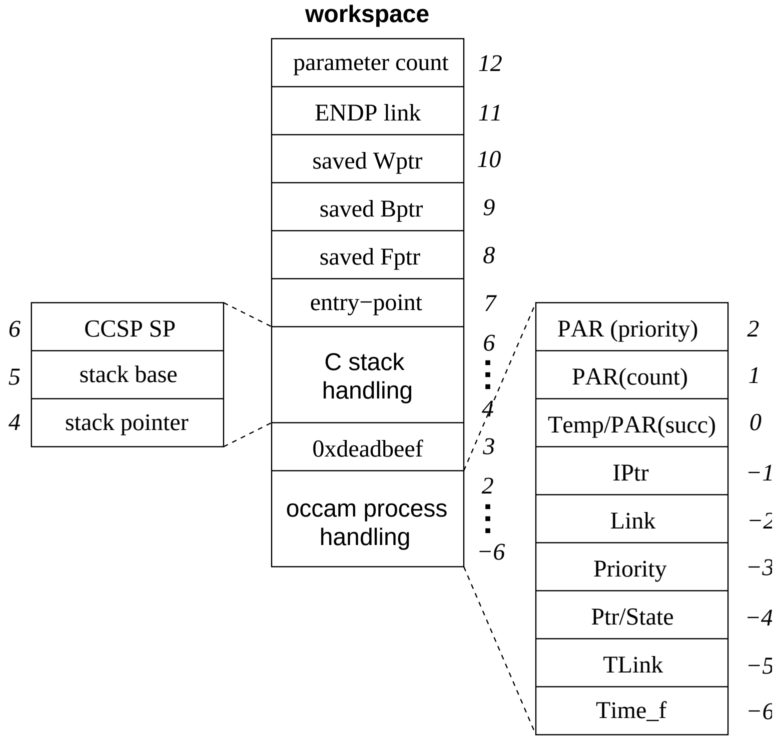


Figure 7.3: Workspace layout for a concurrent C process

'`cifccsp.startprocess`'. If the C process was started using '`ProcPar()`' (or other similar call), this will be 'NotProcess.p', and workspace slot 11 contains a link back to the process performing the '`ProcPar()`'. Finally, workspace slot 12 contains the parameter count for the called C process. This is used to ensure that '`ProcParam()`' is called with the correct number of parameters (the same as that given to '`ProcAlloc()`').

'`ProcAlloc()`' places parameters directly in the newly allocated C stack, along with the '`Process`' pointer (given as the first argument). The occam state is then initialised to resume at a stub function, written in target assembly. This stub function, when scheduled, saves the occam state, switches stacks, then calls the C process (whose parameters are already in the stack). When (and if) the C function returns, the occam run-time state is restored and the process shut-down (which will possibly involve rescheduling a blocked '`ProcPar()`' or '`cifccsp.startprocess`').

The '`cifccsp.startprocess`' `PROC` is a small piece of inline Transputer assembly, that schedules a C process for execution before suspending itself. The code for this is:

```
INLINE PROC cifccsp.startprocess (INT addr)
  ASM
    LDLP 0            -- load current Wptr
    LD addr           -- load C process Wptr
    STNL 10           -- store current Wptr in C process workspace
    LD addr           -- load C process Wptr
    RUNP              -- run process
    STOPP             -- stop (this) process
:
```

To handle the saving and restoring of the occam state, two (i386) assembler macros are used:

```
#define CIF_SAVE_OCCAM_STATE
    movl    %esp, 24(%ebp)      /* save occam-kernel stack pointer */
    movl    %esi, 32(%ebp)      /* save Fptr */
    movl    %edi, 36(%ebp)      /* save Bptr */

#define CIF_RESTORE_OCCAM_STATE
    movl    24(%ebp), %esp      /* restore occam-kernel stack pointer */
    movl    32(%ebp), %esi      /* restore Fptr */
    movl    36(%ebp), %edi      /* restore Bptr */
```

Another pair of assembler macros handle the saving and restoring of the C process state:

```
#define CIF_SAVE_C_STATE
    movl    cifccsp_wptr, %ebp  /* load Wptr */
    movl    %esp, 16(%ebp)      /* save C process stack pointer */
    movl    $0f, 28(%ebp)       /* save return address (forward local label) */

#define CIF_RESTORE_C_STATE
    movl    16(%ebp), %esp      /* load C process stack pointer */
    movl    28(%ebp), %eax      /* load return address */
    movl    %ebp, cifccsp_wptr  /* store Wptr */
```

When a C process first starts up — the first time it is scheduled for execution — the occam run-time state is stored, the C state loaded and the user-function called. This happens all within the '`occstub_entry_point`' assembler routine.

Once running, a C process has complete control until it enters the occam run-time kernel — where it may be rescheduled. A set of assembler macros provide code which transfers control from a C process to the run-time kernel. Some of these, such as '`ChanInInt()`', are used directly by user C code — principally communication instructions. Other entry-points are called from within CIF's own C routines — for example, '`ProcPar()`' will call the '`StartProcess()`' macro to place a C process on the run-queue.

The C macros that provide access to the run-time kernel from a C process follow a standard form. '`ChanInChar()`', for example, is:

```
#define occstub_ChanInChar(C,P)
    { int CIF_dummy1, CIF_dummy2;
        __asm__ __volatile__ ("
            "       pushl   %%ebp                                   \n"
            CIF_SAVE_C_STATE
            CIF_RESTORE_OCCAM_STATE
            "       pushl   %%eax           # dest addr             \n"
            "       pushl   %%ebx           # chan addr             \n"
            "       pushl   $1              # count                 \n"
            "       pushl   $occstub_resume_point                   \n"
            "       jmp     _X_in                                   \n"
            "0:     popl    %%ebp                                   \n"
            : "=a" (CIF_dummy1), "=b" (CIF_dummy2) /* no outputs */
            : "a" (P), "b" (C)
            : "ecx", "edx", "esi", "edi", "cc", "memory");
    }
```

When executed from a user C process, this code pushes the 'EBP' register onto its own C stack, calls the 'CIF_SAVE_C_STATE' macro, followed by the 'CIF_RESTORE_OCCAM_STATE' macro, and finally the code required to interface with the run-time kernel. For this example, that uses the '_X_in' entry-point, arguments are passed on the stack. Other kernel entry conventions (described in section 6.2.2) may require different argument-passing mechanisms.

Inside the 'CIF_SAVE_C_STATE' macro, the (forward) address of label '0:' is saved in the C process's workspace. This label is where the invoking process resumes from — which only needs to restore the earlier saved 'EBP' register. The return-address for the C process in the occam kernel is set to another assembler routine, 'occstub_resume_point'. This has a relatively trivial implementation:

```
occstub_resume_point:
    CIF_SAVE_OCCAM_STATE
    CIF_RESTORE_C_STATE
    jmp      *%eax              /* jump directly to resume point */
```

More complex functions (such as 'ProcPar()' and 'ProcAlt()') are implemented in C, using the lower-level assembly macros to interface with the run-time kernel where needed.

7.2 Process Priority

A useful feature for application programs, and a major requirement in some real-time systems, is the ability to prioritise. Standard CSP [Hoa85] does not include a treatment of any priority, even though 'PRI ALT' (and also 'PRI PAR') have existed in occam for many years. More recently, the extended CSP variant $\mathcal{CSPP}$ [Law02] does provide semantics for priority, and priority issues in general.

The priority requirement for many applications relies on *process priority*. That is, if two or more processes of differing priorities are available for scheduling, higher priority processes will be scheduled first. Furthermore, a major requirement of many real-time control applications is the ability to have a set of cyclic control processes, managed so that each process completes its cycle within a fixed time. Process priority can be used to achieve this.

The original Transputer hardware supported two levels of priority, low and high, with fast pre-emptive scheduling. The original KRoC did not support priority, quietly implementing 'PRI PAR' as just 'PAR'. This is not sufficient for real-time applications which require more than one control process — even at low processor loadings. Efficient classical solutions, such as *deadline* and *rate-monotonic* scheduling [LL73], require multiple time-varying priorities.

Priority can be engineered in the application however, as described in [SWB90] and [Wel90], where process scheduling is controlled through communication. The granularity of priority-changing in such systems is dependant on the frequency of interaction with the 'scheduler' process. Too frequent and the overhead will be high. Too scarce and the system may not respond in a timely manner when higher priority processes become runnable. The Transputer implementation (hardware) provided *pre-emptive* scheduling of high-priority processes, triggered typically from an event pin on the chip [Inm93] — that could be connected to hardware devices directly.

The addition of priority presented here, previously described in [BW02a], provides 32 levels of process priority for KRoC/Linux. Rather than using 'PRI PAR' to control priority, a number of compiler built-ins are used, leaving 'PRI PAR' ineffective. These built-ins are described in section 7.2.1.

The implementation of this priority, described in section 7.2.2, is mainly handled in the run-time kernel (CCSP), but requires changes to all parts of the KRoC system (due to a slight change in the layout of a process' workspace). The implementation, whose basic ideas follow those proposed by

Ploeg et. al. in [PSBR94], aims for small overheads so that it is practically useful. The performance of this priority implementation is discussed in section 7.2.3.

7.2.1 The occam Interface

The 32-levels of priority are accessed directly, using various compiler built-ins. The highest priority is 0 and the lowest 31, limited in order to be efficient in parts of the implementation. A process may only inspect or manipulate its own priority — changing the priority of other processes is not supported. The built-ins provided are:

```
INT FUNCTION GETPRI ()
PROC SETPRI (VAL INT p)
PROC INCPRI ()
PROC DECPRI ()
```

The three `PROC`s '`SETPRI`', '`INCPRI`' and '`DECPRI`' are treated as wholly side-effecting, and therefore cannot be used in `FUNCTION` bodies. This restriction can only be relaxed if a structured mechanism that allows a process to *temporarily* change its priority is provided. '`GETPRI`' is wholly non-side-effecting however, and can be safely used in expressions.

The behaviour for out-of-range priorities is deemed to be implementation dependent. In the implementation presented here (in section 7.2.2), out-of-range values are quitely ignored, mapping to the highest and lowest priorities as appropriate.

The two `PROC`s '`INCPRI`' and '`DECPRI`' are provided mainly for convenience. The implementations are simply '`SETPRI (GETPRI() - 1)`' and '`SETPRI (GETPRI() + 1)`' respectively.

A process is free to change its own priority arbitrarily — but as mentioned, may not change the priority of any other process. Changing from a lower to a higher priority will always generally succeed immediately, however, there is the possibility that another (higher priority) process became runnable, which would be scheduled first. Changing from a higher to a lower priority will cause a reschedule (and deactivation of the current process) if any other process is waiting (runnable) at the target priority level or higher.

7.2.2 Implementing Priority Handling

The implementation of priority is handled largely in the run-time kernel. The occam compiler's access to priority is handled through two new Transputer instructions: '`GETPRI`' and '`SETPRI`' (detailed in section B.1.5). These are called directly from the compiler built-ins '`GETPRI()`' and '`SETPRI(p)`'.

The most significant change required to support process priority is the addition of a *process-priority* slot in the occam workspace. This has been inserted between the existing 'Link' and 'Pointer' fields, as shown in figure 7.4. The "below workspace" slots are only used when a process is blocked (either on communication or other synchronsation), or waiting (on the run-queue).

The only changes to the compiler, besides the addition of the priority-related compiler built-ins and new instruction support, are slight adjustments to process workspace requirements (for the extra slot), and the addition of a '*SavedPriority*' slot for `PAR`. This extra slot in the `PAR` implementation ensures that a process's priority is saved and restored around a `PAR` block — necessary since the last process in a `PAR` runs in the workspace of the process performing the `PAR` (for efficiency).

The modified CCSP run-time kernel (examined extensively in Chapter 6) maintains 32 separate run-queues, one for each of the priority levels. Priority is managed through the use of two (global) kernel variables: '*PPriority*' and '*PState*'. *PPriority* holds the priority level for any currently running process — i.e. it is what is returned when an occam program calls '`GETPRI()`'. *PState* is a bit-field

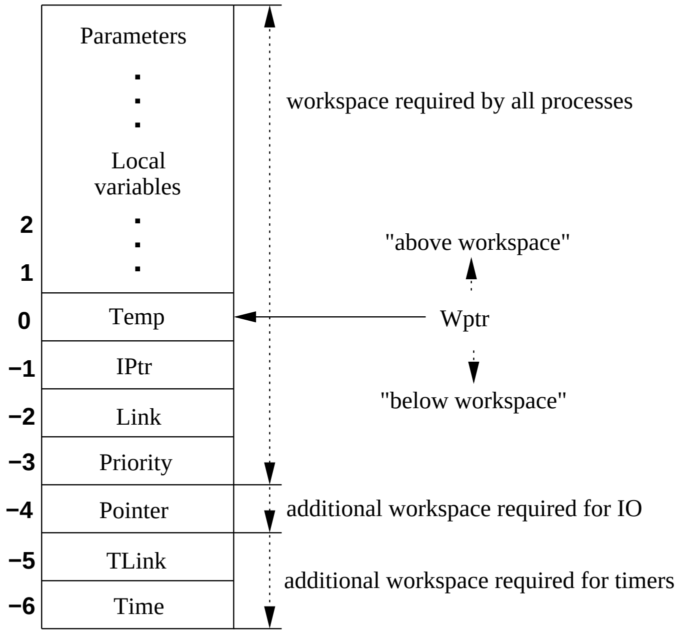


Figure 7.4: occam process workspace layout (with priority field)

(32-bits wide) that indicates at what priorities runnable processes exist. A single-cycle instruction (on i386 based architectures at least) can be used to find the left-most or right-most set-bit in the *PState* word, used when rescheduling to determine the priority of the next "best" process.

The various priority run-queues (organised into two arrays of front-pointers and back-pointers) are only used to *hold* the various run-queues. The '`Fptr`' and '`Bptr`' variables (held in registers during the execution of occam code) are still used, holding the appropriate run-queue. These are loaded and stored in the two arrays when changing priority. Figure 7.5 shows these arrays, along with the priority-related variables and relevant run-time registers.

In practice, the *PState* word is only examined when the current priority-level runs out of processes, or when the completion of an external event makes a higher-priority process runnable. A similar approach, for keeping track of the priority-levels at which runnable processes exist, is used in MESH [BDv99a, BDv99b].

Changing Priority

For the majority of the time, the currently running process has the highest priority (possibly alongside others, with which it will be scheduled). A process only becomes runnable (hence a potential priority change point) on certain well-defined conditions: when a new process is created using the '`STARTP`' instruction; when an existing process is scheduled using '`RUNP`'; when a process becomes runnable as a result of channel communication; or, when a process becomes runnable as a result of an external event (for example, a simple timeout or blocking system-call completion). In addition to these four, the completion of a `PAR` is also a potential priority-change point.

The case for '`STARTP`' is largely uninteresting: the new process inherits the priority of the current process. '`RUNP`' is used to place an existing process on the run queue. With priority enabled, the '*Priority*' field of the process being enqueued is used to select the correct run-queue (if not the

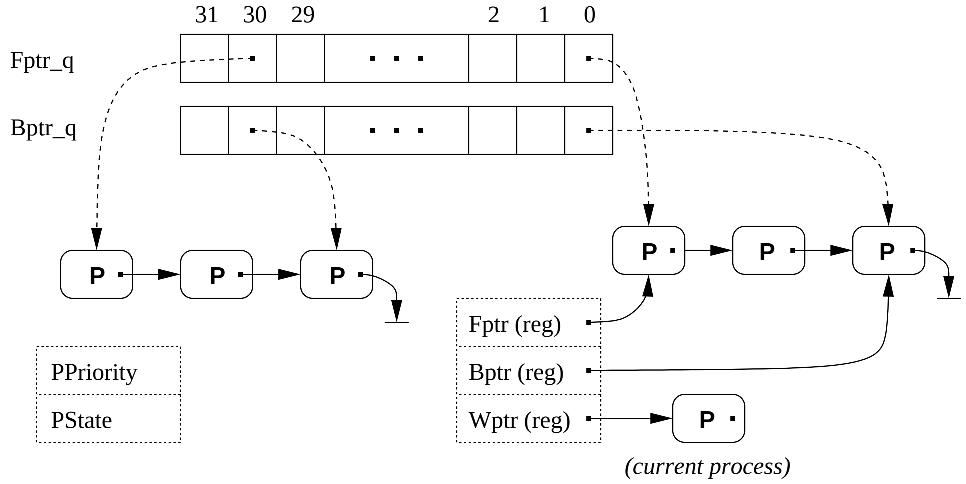


Figure 7.5: Priority run-queues and priority related kernel variables

current priority — stored in '*PPriority*'). If the enqueued process is of a higher priority, the current process is descheduled, priority levels changed and the newly enqueued process scheduled. Channel communication follows suite, since after communicating the blocked process is rescheduled — it is, in effect, '`RUNP`'d.

The handling of external events (notified through the synchronisation flags, shown in figure 6.3 on page 149) has been explicitly modified to handle priority. Traditionally, timeouts have been implemented using the UNIX kernel-timers (accessed through the '`setitimer()`' system call, or equivalent). When a timeout expires, a signal is raised and the '`tim_sync`' synchronisation flag is set. The addition of CPU-timer support (covered in section 6.6) changes this slightly — the signal is only arranged for when the run-time system sleeps (because it has nothing to run). The completion of a blocking system call (section 6.3) raises a signal, whose handler sets the '`block_sync`' synchronisation flag.

Whenever the scheduler proper is invoked, it checks the synchronisation flags for activity and processes them accordingly. Processes made runnable as a result have their '*Priority*' field inspected and are added to the appropriate run-queue. Rather than immediately changing priority level if a higher-priority process becomes runnable, an extra synchronisation flag, '`priority_sync`', is set. This is used to indicate that a higher priority process *may* be available, and is checked after the other synchronisation flags.

After processing other synchronisations flags, if the '`priority_sync`' flag is set, the '*PState*' word is scanned to find the highest-priority runnable process. If this is different than the current '*PPriority*', the current run-queue is saved and the new one loaded (in addition to modifying '*PPriority*'). Algorithm 7.1 shows the algorithm used to change priorities, and will typically be used with "**bit.scan.left** (PState)" as the new priority.

The '`priority_sync`' flag is also set under some other (less obvious) conditions to indicate that a scan should be made of '*PState*' at the next opportunity. This happens, for example, when a process is queued at a higher priority, but the current priority cannot be changed immediately — for instructions that manipulate the run-queue but may not be able to deschedule. The priority flag is also set when a process finishes, and is the last process in the current run-queue, or when '`ENDP`' (end process) completes a `PAR` barrier and enqueues the blocked process. The scheduler will see this flag on the next reschedule and change to a higher priority if needed.

Algorithm 7.1: Priority change algorithm

```
// PPriority : current priority, −1 if none
// PState : priority state field
// Fptr.Q, Bptr.Q : priority run-queues
// Fptr, Bptr : current run-queue
priority.change (new.pri):
if PPriority = −1 then
  PPriority ⇐ new.pri
  Fptr ⇐ Fptr.Q[PPriority]
  Bptr ⇐ Bptr.Q[PPriority]
else if PPriority ≠ new.pri then
  Fptr.Q[PPriority] ⇐ Fptr
  Bptr.Q[PPriority] ⇐ Bptr
  PPriority ⇐ new.pri
  if Fptr = NotProcess.p then
    PState ⇐ ( PState bit.and ( bit.not (1 ≪ PPriority )))
  else
    PState ⇐ ( PState bit.or (1 ≪ PPriority ))
  end if
  Fptr ⇐ Fptr.Q[PPriority]
  Bptr ⇐ Bptr.Q[PPriority]
end if
```

Enqueuing Multiple Processes

The user-defined types that provide higher-levels of process synchronisation [WW97] (in particular '`BUCKET`' and '`BARRIER`'), use the Transputer instructions '`SAVEL`' (save current run-queue at supplied address), and '`STLF`', '`STLB`' (store low-priority front and back pointers), to join a queue of blocked processes to the run-queue at $O(1)$ cost.

This does not mix well with priority support. For a synchronisation type such as the barrier, processes arrive (synchronise) and are placed on a process queue inside the `BARRIER` type:

```
DATA TYPE BARRIER
  RECORD
    INT Fptr, Bptr:
    INT active, count:
:
```

When the last process synchronises (using '`synchronise.barrier`'), processes blocked on the queue held by '`Fptr`' and '`Bptr`' are enqueued onto the current run-queue en-masse. If all the processes waiting on this queue are at the same priority as process that completes the barrier, then there is no problem — the processes will be enqueued onto the correct (current) run-queue. Where processes of differing priorities synchronise on the barrier, this approach causes them to "inherit" the priority of the process that completes the barrier synchronisaion.

There are two problems that need to be solved here. Firstly, how to load and store the queue pointers for priorities other than *PPriority*. And secondly, how to handle the queueing of processes within types such as '`BARRIER`'. The Transputer provided *six* instructions for manipulating the run-queue: '`SAVEH`' and '`SAVEL`' for retrieving the *high-priority* and *low-priority* run-queue pointers respecitively; '`STHF`' and '`STHB`' for storing the high-priority run-queue front and back pointers; and finally, '`STLF`' and '`STLB`' for storing the low-priority run-queue pointers. KRoC based applications only ever make use of low-priority run-queue instructions — since the original KRoC targets did not support priority. Attempts to use the high-priority queue instructions with KRoC will result in run-time traps (the various KRoC translators produce run-time error generation code for these instructions).

Rather than adding new instructions to support the loading and storing of the various priority run-queues, the high-priority run-queue instructions ('SAVEH', 'STHF' and 'STHB') have been re-used[2]. For the Transputer, these instructions take a single argument in '*Areg*' — a pointer to two INTs for 'SAVEH', and the INTs themselves for 'STHF' and 'STHB'.

These instructions have been modified to take an additional argument — a priority level, in '*Areg*', with the original argument in 'Breg'. These modified instructions allow the various priority queues to be manipulated, without affecting the existing (low-priority) instructions, that always operate on the *current* run-queue. If the modified high-priority instructions are called with '*Areg*' equal to '*PPriority*', then the current run-queue is modified.

Within the 'BARIER' type and similar user-defined types (that manipulate processes), provision must be made for handling the potentially varying priority of the synchronising processes. This can either be handled as processes synchronise, by placing them on different queues (one for each priority level) within the BARRIER; or by retaining the single queue, and enqueuing the processes individually (respecting their priority) when the barrier completes.

The choice of which approach is taken depends on the likely number of processes involved. Handling the priority of each process individually when the barrier completes incurs an $O(n)$ cost, where n is the number of *active* barrier processes. The other approach, of maintaining 32 separate run-queues in the barrier (or as many priority levels are supported), incurs a constant cost, approximately 32 times that of a single enqueue ($O(32)$). Implementations can optimise somewhat, by keeping track of which priority queues contain processes (a local *PState* for the BARRIER), reducing this cost to $O(m)$, where m is the number of priorites in use within the BARRIER.

The blocking system-calls extension (covered in section 6.3) takes the approach of handling each process individually. This is primarily because the number of completed blocking calls is expected to be small — perhaps 4 or 5. The overheads of blocking calls themselves impose a limit on the frequency of dispatching and collecting.

Writing Compatible Code

For distribution-shipped occam sources, such as BUCKET and BARRIER, it is desirable to have code that will work both with and without priority support. The "normal" way to handle this would be the creation of multiple sources, one of which is selected (via a symbolic link, or name-prefix/name-suffix mechanism, for example), depending on whether priority is enabled. This works and is acceptable — since it is currently the only way. It is, however, not entirely desirable — essentially it means maintaining multiple source files, or having a pre-processor generate the various files from a single source.

Section 7.5 presents a compiler-based pre-processor for occam. This can be used to efficiently manage code that needs to behave differently with and without priority ('BARRIER' and 'BUCKET' are just two particular examples). As discussed in that section, the pre-processor defines various constants depending on the current configuration and compilation options. One of these is 'PROCESS.PRIORITY', defined to be the number of priority levels supported, but only if the KRoC

[2]The compiler never generates these instructions directly, and they are unhandled in most KRoC implementations. Thus, recycling them is largely safe.

system was built with priority support enabled. The following occam fragment shows the earlier 'BARRIER' data-type declaration, modified to use multiple run-queues if priority is enabled:

```
DATA TYPE BARRIER
  RECORD
    #IF DEFINED (PROCESS.PRIORITY)
      #RELAX                                    -- adjust indentation
      [##PROCESS.PRIORITY]INT Fptr.q, Bptr.q:
      INT Q.state:
    #ELSE
      #RELAX                                    -- adjust indentation
      INT Fptr, Bptr:
    #ENDIF
    INT active, count:
:
```

The somewhat peculiar "##PROCESS.PRIORITY" is used to retrieve the value of that pre-processor define, literally replacing it with the value it represents. This is covered in more detail in section 7.5.

Similar use of the pre-processor can be employed in general code, at arbitrary points (similar to C's pre-processor), to change program behaviour if priority is enabled.

7.2.3 Performance

Figure 7.6 shows the process network and loop-body code for the priority benchmark. Two processes, 'A' and 'B', repeatedly communicate with the third process 'C' but change priority before/after each communication. This particular arrangement of priority changing and communication results in a deterministic scheduling order for the processes involved — giving a known number of priority change operations per cycle. The 'C' process is run at the lowest priority level (31), while the producers ('A' and 'B') alternate between 1 and 4. Generally speaking, however, priority *cannot* be used to guarantee deterministic scheduling.

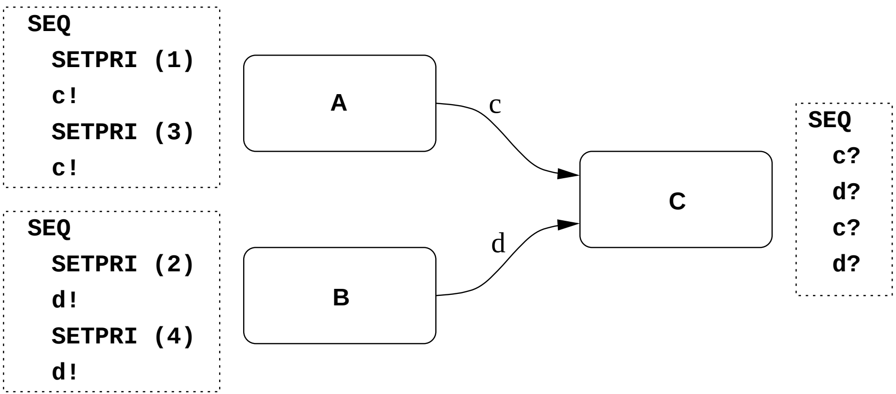


Figure 7.6: Priority benchmark process network

Figure 7.7 shows the execution trace for this benchmark, indicating the timed (and looping) section. The priority overheads are calculated from the difference beween the code shown and a priority-free version.

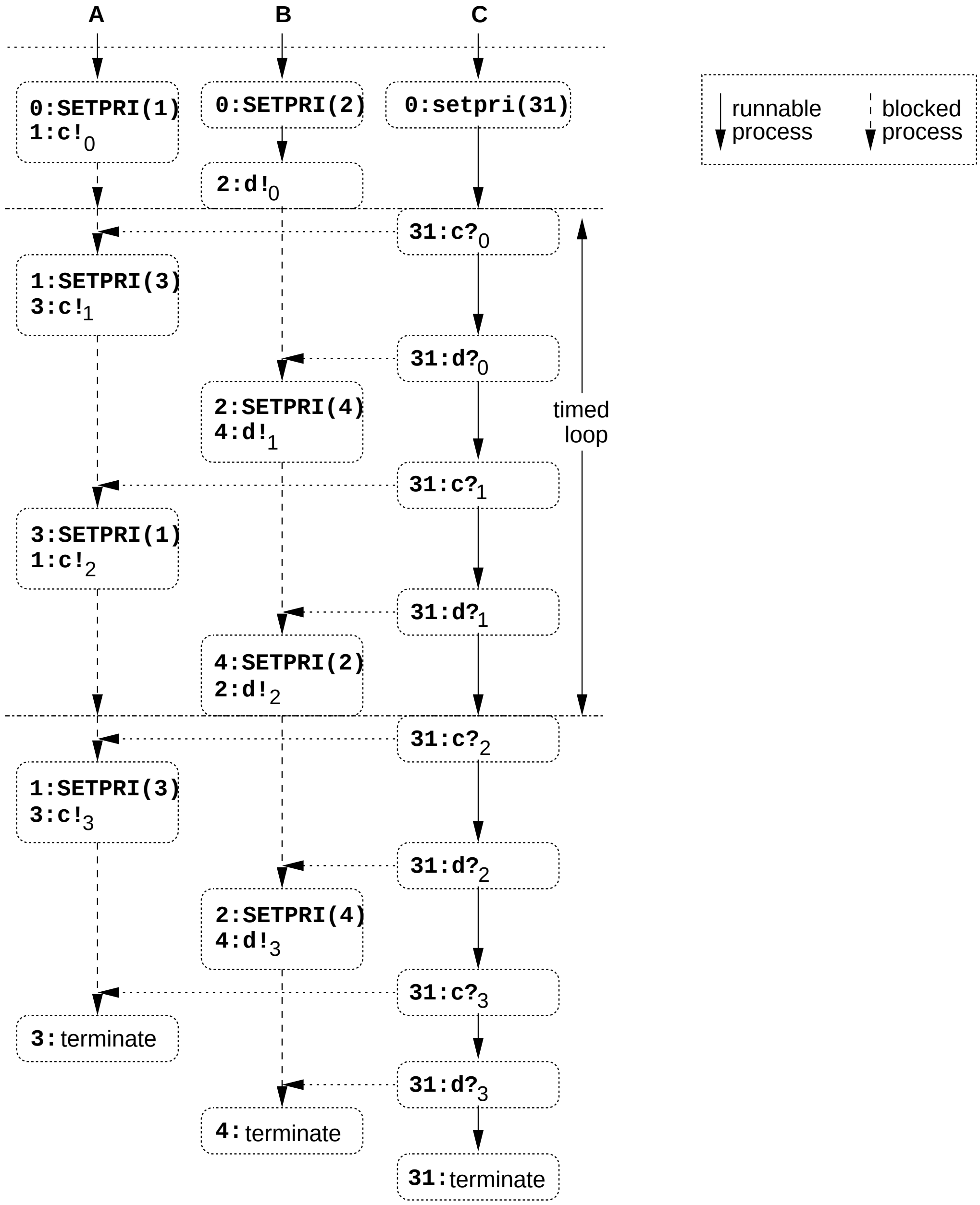


Figure 7.7: Execution trace for priority benchmark program

The loop involves a total of 8 context switches; 4 for rescheduling when a process blocks in a channel (the 'A' and 'B' processes), and 4 for rescheduling a higher-priority process when 'C' communicates. Every context switch involves changing priority levels, as do the four calls to 'SETPRI'. The non-prioritised version also has 8 context-switches per cycle — for any rescheduling policy[3]. Calls to 'SETPRI' do not result in context switches, only a priority level change.

The loop overhead (difference) for this benchmark is 752 ns, measured on an 800 MHz P3.

[3]To what extent this is true in general is currently unknown.

Overall, the run-queue and priority are changed 12 times, giving an average overhead of 63 ns for a priority-level change (around 50 machine cycles). This is actually over-estimated, since it includes the overhead of a kernel-call for '`SETPRI`'.

7.3 Post-Mortem Debugging

The addition of support for debugging occam programs greatly enhances the value of KRoC as a programming environment. Traditionally with KRoC, run-time errors result in a simple error message before the program exits. This behaviour is known as 'halt' error-mode, as opposed to 'stop' error-mode, where processes are simply descheduled when they generate run-time errors. In some ways 'stop' error-mode is the more natural of the two — a run-time error within a parallel system only affects the process that caused it, and it may be desirable to allow the rest of the system to continue running.

The majority of KRoC systems do not support 'stop' error-mode, however, and attempts to enable it result in internal compiler errors (largely virtual-transputer stack overflows).

Most run-time errors that can befall an occam program are generated explicitly by the compiler, with the exception of integer-overflow and divide-by-zero errors. The remaining errors are caused by one of four virtual-transputer instructions: '`CSUB0`', '`CCNT1`', '`SETERR`' or '`FPCHKERR`'. The first two are range-checking instructions, that are typically used before array subscriptions, to ensure that accesses fall within the array bounds. The '`SETERR`' instruction is used to cause an immediate error, and is generated when '`STOP`' is compiled in 'halt' error-mode. On the Transputer, this instruction halts the processor, and therefore cannot be used in 'stop' error-mode.

In 'stop' error-mode, the compiler will generate the less destructive '`STOPP`' (deschedule current process) instruction for cases where '`STOP`' is encountered in occam— that includes the implicit '`STOP`' inserted for non-default handling '`IF`' and '`CASE`' constructs, as well as for wholly disabled '`ALT`'s. The final run-time error cause is from '`FPCHKERR`', that is used to check for any pending error from the floating-point unit (and causing a run-time error if appropriate).

Support for post-mortem debugging was added first to the Sparc version of KRoC, and later the Linux version, as described in [WB00]. This form of debugging, for the two versions of KRoC that implement it, uses the compiler in 'halt' error-mode. Run-time errors are processed largely in the standard manner — the run-time kernel will be invoked (via an error-generating instruction) to process the error and exit.

Post-mortem debugging provides detailed information for the user when a run-time error occurs, implemented with a combination of special code inserted by the translator, and corresponding handling in the run-time kernel. The precise details of how this is done differ for the Sparc and Linux versions of KRoC, but provide the same information (crash diagnostics) to the user. Additionally, when deadlock occurs (no runnable processes), the global workspace is examined to find processes blocked in channel communication[4].

As far as handling 'stop' error-mode goes, the compiler's own implementation (designed with the Transputer in mind) is not entirely useful. For reporting errors to the user, interaction with the run-time kernel (indicating an error condition) is perferable to a simple '`STOPP`' (which the compiler generates for errors in 'stop' error-mode).

The work here adds to that presented in [WB00], by providing an implementation for 'stop'

[4]There is currently no support for locating processes blocked on anything other than channel communication, such as semaphore '`CLAIM`'s. This is missing largely because it does not fit will with the existing implementation (that utilises the process's '`Link`' field).

error-mode. When a process generates an error, that error is reported (when debugging is enabled), the process is descheduled and other processes continue running. Generally speaking, this will lead to deadlock, unless the system is explicitly designed to be aware of potentially stopping processes (that it cannot easily detect). This leads naturally into another problem — how to detect the failure of a component, that will no longer respond to channel interaction. Some consideration is given to this under future work, discussed in section 8.4.1, related to work done by Welch in [Wel89].

7.3.1 Standard Run-Time Errors

As noted, the standard run-time errors fall into four categories: integer overflow and integer divide-by-zero; range-check errors (typically on array accesses); explicitly generated errors (‘STOP’); and floating-point errors. The handling of these, for both ‘halt’ and ‘stop’ error mode is examined in the following sections, although the difference between the two is relatively minor (due to the implementation of ‘stop’ error-mode outside the compiler[5]).

For all these errors, the PROC or FUNCTION where the error occurred is reported, along with the filename and line-number. For example:

```
KRoC: overflow on operation ADC in PROC "rte1" in file "rte1.occ" near line 8
OCCAM STOPped at 0804ddf4
```

The first line of output is from the post-mortem debugger, the second is produced in ‘halt’ error-mode, indicating where execution stopped. In ‘stop’ error-mode, this second line is never produced. The run-time system continues scheduling processes, or deadlocks if none are runnable.

For some errors, such as the above, information is also provided about the operation that caused the error — ‘ADC’ in this example.

Debugging Information

For any post-mortem debugging, the textual information regarding PROC/FUNCTION names and file-names must be available to the run-time system. Both the Sparc and i386 implementations follow a similar method, originally devised by Wood (for the Sparc implementation): strings are placed in the generated code, whose addresses are associated with one or more debugging records. A debugging record contains all the information necessary to report an error — such as the specific error (operation) and the occam line-number, along with the file-name and procedure-name.

In the Sparc implementation, debugging records are assembled in the data segment, and occupy approximately 8 words (32 bytes). When error-handling code is generated, the address of an associated debug record is loaded and the (debugging) error-routine inside the run-time kernel executed.

Debugging in the Sparc KRoC also reports the line-number and file-names within temporary files (the compiler output and translator output). This is less relevant for the i386 version of KRoC, that uses binary intermediate code (ETC), and by default uses a pipe to the assembler, rather than invoking the assembler on a temporary file. For the Sparc version, the overhead of accessing this information and entering the run-time kernel (without the conditional branch around it) is 3 instructions (12 bytes of machine code), as shown in [WB00]:

[5] Fixing the compiler to correctly handle ‘stop’ error-mode (for KRoC targets at least) is in many cases unnecessary — the run-time kernel currently handles *all* run-time errors, so can simply schedule another occam process instead of exiting when an error occurs.

```
        cmp     %l0,%i4             !- error if negative
        blu     9f
        sethi   %hi(0f),%o0         !*
        call    $$debug
        or      %o0,%lo(0f),%o0     !*

        .data                       !* debugging record
0:      .word 0x12002000,17,31,58,LP0,LF0,LS0,LD0
LP0:    .asciz "out_repeat"
LF0:    .asciz "small_utils.lib"
LS0:    .asciz "nos.kt8"
LD0:    .asciz "nos.s"
        .text
9:                                  !* continue program
```

The Linux version is different, building debugging records directly in the code before jumping to an error handling routine. The Linux debugging record is generally much smaller — only 8 bytes, compared to the Sparc implementation's 32. This is mainly because there is less information to record, no intermediate file locations are recorded, but also due to a different method of handling procedure and file names. For example:

```
        jno     0f                  # jump if not overflow
        movl    $0x8000008, %edx    # first word of debug record
        movl    $0, %ecx            # second word
        jmp     L1
0:                                  # continue program
```

The code here occupies exactly 12 bytes (not counting the initial conditional jump), 8 of which are the debug record. The code at label '`L1`' loads pointers to the procedure and file name tables — that the debug record indexes — before calling an entry-point in the run-time kernel.

The name tables are organised in a simple way: first the number of entries, followed by that many offsets (from the table start) to the actual data, followed by the data itself. For the earlier error reported in "`rte1.occ`", this table is:

```
L4:
        .long   0x00000001, 0x00000008
        .byte   0x72, 0x74, 0x65, 0x31, 0x00, 0x00, 0x00, 0x00
L3:
        .long   0x00000001, 0x00000008
        .byte   0x72, 0x74, 0x65, 0x31, 0x2e, 0x6f, 0x63, 0x63
        .byte   0x00, 0x00, 0x00, 0x00
```

The strings in the above (aligned and null-terminated) are "`rte1`" in the table at '`L4`', and "`rte1.occ`" in the table at '`L3`'.

Range-Check Errors

Range-check errors are generated by the instructions '`CSUB0`', '`CCNT1`', '`CSNGL`' and '`CWORD`'. Other checking instructions exist, particularly in the T9000 instruction set, but these are never used by KRoC. The checking instructions generated for KRoC are typically to ensure that array accesses are within bounds, and to ensure that non-constant replicator counts are not negative, or for some, zero — a replicated '`ALT`' of count 0 is equivalent to '`STOP`', as is the similar '`IF`'.

Some additional range checks are inserted by the translator for the ETC specials encoding constant shift operations — i.e. if the shift is negative or greater than the word size (32-bits).

Without debugging, the code generated checks for the appropriate condition (optimising known values away where possible), leaving a conditional jump either to the translated instruction, or to a generic error handling routine in the run-time kernel. When debugging is enabled, the error-path (previously a jump/call to the run-time kernel) loads various constants onto the stack (8 words representing line number, operation, file and procedure name), and calls the locally generated setup code. This setup code, as described previously, simply further loads the constant table addresses (of file and procedure strings) and enters the run-time kernel.

The debugging overhead for range check operations is just 12-bytes (2 instructions)[6]. The check itself varies in size, depending on the particular check and any known constant values for the operands. For many range-checks, 2 instructions are generated, 1 for some and 3 for others, typically around 6 bytes of machine-code.

Explicit `STOP`s

'`STOP`' in occam results in the compiler generating the '`SETERR`' instruction — that set the 'error-flag' on the Transputer, which would then handle the error appropriately in hardware (based on the error mode). Without debugging, this is translated into a simple run-time kernel call. With debugging enabled, four words are pushed onto the stack (two words of debugging information, and two addresses), before calling the run-time kernel. This adds four extra instructions to the generated code, up to 20 bytes of machine-code.

Since explicitly written `STOP`s are substantially less common than range-checks or overflow-checks (for example), there is less of a need to keep the 'local' code to a minimum.

Integer Errors

Integer errors, like range-errors, are a result of common programming mistakes. Unlike many languages, occam considers the wrapping of variable values to be an error, unless explicitly 'avoided' (as must be done when manipulating time values). Many operations in occam are capable of generating integer-related errors, some less obvious than others.

Table 7.1 lists the various *errors* that are generated (and reported), named to be consistent with the instructions that generate them.

Unlike the original Transputer hardware, Intel (i386) processors do not generate a trap (hardware exception) when overflows occur. Instead, the 'overflow' flag is set in the condition-codes register, which can be easily checked after the operation. Without debugging, the handling of integer overflows is largely a simple conditional-jump into the run-time kernel. Some of the Transputer instructions, such as '`FMUL`' and the floating-point size checks, do not map cleanly onto i386 instructions. For these, the overflow checks are explicit, but not more than a few instructions.

The checking of floating-point values, for the translation of '`FPCHKI32`' and '`FPCHKI64`', are handled by 'dropping-in' floating-point constants for the minimum and maximum integer values. These are placed alongside the debugging jump-code in the generated output.

[6] In actual fact, it is often considerably less than 12 bytes when constants are '`push`'d onto the stack, rather than '`mov`'ed into registers. For errors in a main compilation unit, the 'file-number' will be zero, leaving one of the debugging words as just the 16-bits of 'proc-number' (often small).

Code	Description
ADD	overflow on integer addition
ADC	overflow on constant integer addition
SUB	overflow on integer subtraction
MUL	overflow on integer multiply
LADD	overflow on long (64-bit) addition
LSUB	overflow on long (64-bit) subtraction
LDIV	overflow on long (64-bit) division
FMUL	overflow on fractional multiply
FPCHKI32	overflow for real conversion to a 32-bit integer
FPCHKI64	overflow for real conversion to a 64-bit integer

Table 7.1: Run-time integer errors

Floating-Point Errors

In the original KRoC/Linux, much of the support for floating-point handling (both operation and error) was missing. Certainly, many of the CG-tests (compiler test-suite) failed due to missing floating-point instruction translations.

Floating-point arithmetic, and its associated error handling, are now fully supported by KRoC/Linux. The majority of Transputer floating-point instructions translate relatively cleanly to the Intel FPU (from i387 upwards). The hardware arrangement inside Intel chips places the FPU as a completely separate execution unit, allowing the FPU to run concurrently alongside integer code. Any exchange of data between the FPU and integer units requires synchronisation, that is handled automatically by the processor when required. When operations are entirely FPU and memory related, the integer unit need not be involved, until it comes to the checking of errors, that is.

FPU errors are handled in one of two ways — either by the raising of an explicit hardware trap (delivered to the application as a POSIX signal – '`SIGFPE`'), or by the setting of a flag in the FPU's status register. Without debugging, traps are used to handle floating-point errors. This results in the run-time system printing an appropriate error message and exiting.

With debugging, floating-point errors are handled in software using explicit checks. The Transputer also contained an asynchronous floating-point unit, but handles errors differently. In the Transputer, errors are allowed to accumulate in the floating-point unit, as they are in the Intel FPU (provided that the hardware-trap error mechanism is turned off). After a sequence of floating-point operations, the occam compiler generates the '`FPCHKERR`' instruction that, on the Transputer, checks for an FPU error and either halts or stops if an error had occurred.

The similar operation on the Intel involves reading the FPU status register, that indicates what errors, if any, have occurred. These are cleared by writing to the FPU control register. Table 7.2 shows the various floating-point errors reported by the Intel FPU (and hence those reported by KRoC/Linux).

Without debugging, the '`FPCHKERR`' instruction translates to the Intel '`fwait`' instruction. This forces synchronisation with the FPU, that will cause a hardware trap if an error occurred. With debugging, '`FPCHKERR`' is translated into a series of instructions that read the FPU status, mask off the unwanted bits, and if the result is non-zero, generates a jump into the floating-point error jump-code (with two registers filled with the required debugging information, including the error bits). The floating-point error routine in CCSP simply prints the error (along with the file, `PROC`

Error	Description
invalid-op	an invalid/unsupported FPU op-code was used
denormal	a denormalised operand was used (not necessarily an error)
div-by-zero	a floating-point divide-by-zero was attempted
overflow	overflow occurred
underflow	underflow occurred
inexact	an inexact result was produced

Table 7.2: Intel floating-point errors reported by KRoC/Linux

and line where the error occurred), then either terminates or continues (depending on the error-mode). To ensure that the correct location in the file is reported (or as near to as possible), the translator keeps track of where the last (error-generating) floating-point instruction occurred. This information is used when constructing the debugging record for '`FPCHKERR`', rather than the location of that instruction itself.

7.3.2 Low-Level Debugging

Occasionally, fatal run-time memory errors occur, resulting in the somewhat unfriendly "segmentation fault" error message. These are either the result of errors (bugs) in the KRoC system itself, or the result of damaged external interaction from within occam. In either case, the provision for debugging already described is unable to provide information for these errors[7].

To make debugging information available at *any* point, the translator constantly updates four global variables with debugging information (whenever the line, `PROC` or file changes, and immediately after descheduling instructions). Two of these global variables are used to hold pointers to the current file-name and `PROC`-name tables; another holds the indexes and line-number. The last global debugging variable is currently spare, and set to a magic-constant (until a future use for it is found).

When the run-time system encounters a fatal error, these debug variables are examined, and if set, the debugging information contained in them is printed. The majority of the time, this information represents the exact position of execution within an occam program. When externally called C code causes a fatal error, the location of the external call is reported.

7.3.3 Deadlock Detection

When concurrent occam/CSP systems do go wrong (typically through programmer error), the result is usually deadlock. The run-time system reports deadlock when it has no runnable processes, that includes processes waiting on the timer-queue or waiting for external IO completion (blocking system-calls).

In many occam applications, parts of a system can deadlock, without deadlocking the rest of the system — although the overall operation may be adversely affected. Finding the cause of deadlock is typically not easy. A common approach is the addition of debugging messages to an occam system, in an attempt to trace the events leading up to deadlock.

[7]It is conceivable that with sufficiently 'clever' analysis, run-time checks on pointer accesses could be made. Such checks would not be unlike those found in the C dialect "Cyclone" [JMG+02], that provides a 'safer' C (with extensive compile-time and run-time checks).

Post-mortem debugging for deadlock was first added to KRoC/Sparc by Wood [WB00]. In this implementation, when the run-time system detects deadlock, it *scans* the entire occam workspace looking for 'blocked' processes. A simple yet novel approach is used for this: before performing a blocking communication, the translator inserts code that sets the process's 'Link' field back to that process's 'Wptr'. When deadlock occurs, these 'markers' are searched for and 'stuck' processes reported.

The KRoC/Linux implementation uses a very similar approach, but with a slightly different implementation (to follow the exiting post-mortem implementations here). For example, with debugging enabled, simple 'INT' output generates code approximately equivalent to:

```
    ; begin OUTWORD translation
    movl    %Wptr, Link(%Wptr)        ; store Wptr in the Link field
    movl    $0f, Iptr(%Wptr)          ; store return address in Iptr field
    jmp     _Y_in8                    ; jump into run-time kernel
0:
    jmp     L19                       ; jump over debugging record
    .byte   0x0d, 0x00, 0x03, 0x04
    .byte   0x00, 0x00, 0x00, 0x00
    .byte   0xde, 0xad, 0xbe, 0xef
    jmp     L9                        ; jump to debugging jump-code
L19:
    movl    $0, Link(%Wptr)           ; clear Link field
    ...  translation continues
```

The debugging record includes the line number of the communication, and indexes into the local file and PROC name tables. Also included is a magic constant, that helps identify 'correct' cases.

When the run-time system deadlocks, the global workspace is scanned for words that contain a pointer back two words (the Link from a process to itself). When found, what ought to be the return address is checked to ensure that it references code, and not other memory. If this is the case, the code at the return-address is examined, as it should contain a debugging record placed between two jump instructions — which is checked by looking for the 'jump' op-codes and the magic constant '0xdeadbeef'. If all is safe up to this point, the run-time system executes the jump-instruction following the debug record. This is to some simple local code that places addresses for the appropriate file/PROC name table in processor registers (picked up by the run-time system when that setup returns).

If all goes well, the information contained in the debugging record is displayed — as it is a blocked process. If any checks fail, the potential debug-record is ignored and the search for others continues.

Unfortunately, this mechanism only works for processes that are blocked on *communication* (including ALTs). The same mechanism does not work for processes waiting on SHARED channel-type CLAIMs however. These use a semaphore-based implementation for queueing processes, that uses the 'Link' field internally for maintaining the queue. Being able to detect deadlocked processes blocked on these CLAIMs is useful however. One possible approach would be the explicit identification of such queues to the run-time system, either using a similar memory-trick in the semaphore, or by making explicit 'new_queue' and 'free_queue' calls to the run-time portion of the post-mortem debugger.

Something similar to this already happens for dynamically loadable processes (section 6.5), that have their (dynamic) workspaces explicitly identified to the run-time system. This enables the post-mortem debugger to search for deadlocked processes within these dynamic processes.

Another incomplete part of post-mortem debugging is the detection of deadlock within FORKed,

n-replicated, and recursive parallel processes (covered in Chapter 5). Again, this would require informing the run-time system about such dynamic memories — which it cannot easily detect without compiler (or translator) support.

7.4 Support for Higher Level Interaction

New facilities have been added to the occam compiler that allow channel communications to be extracted from, and inserted into, occam processes. This extension was developed to make more transparent the transport of channel communications over TCP/IP networks, currently being investigated by Schweigler et al. [SBW03].

The extended rendezvous (section 3.7) provides the necessary mechanisms to handle the semantics of channel communication over TCP/IP networks — any communication can simply be extended for the duration of the network transaction, controlled by the process(es) providing the networking support. Figure 7.8 shows an example arrangement of three processes distributed across three nodes, with a fixed network infrastructure.

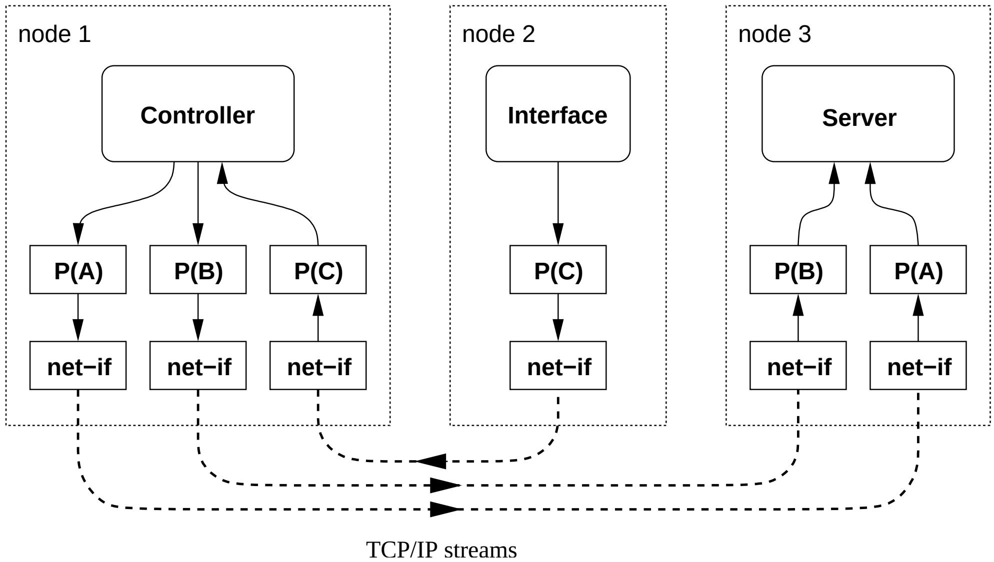


Figure 7.8: Example distributed occam process network

This is just one method of providing network connectivity — many others are possible. The processes shown in figure 7.8 use two common 'net-if' (network interface) processes, that handle the transporting of data over a TCP/IP stream with extended synchronisation. These two processes might have the PROC signatures:

```
PROC net.if.recv (VAL []BYTE local.host, VAL INT local.port, CHAN []MOBILE BYTE out!)
PROC net.if.send (VAL []BYTE remote.host, VAL INT remote.port, CHAN []MOBILE BYTE in?)
```

These PROCs are connected through channels of dynamic mobile arrays (of bytes), used principally to reduce communication overheads within the local network. Other protocols are certainly possible: an INT counted array of BYTEs, for example. It is the responsibility of the 'net-if' components to ensure that data written to a '`net.if.send`' emerges same-sized from '`net.if.recv`' — TCP/IP is a stream protocol, not a datagram protocol.

The protocol-conversion components ('P(a)', 'P(b)' and 'P(c)') are intended to convert to and from a user protocol type (communicated to/from the application components). Writing these protocol converters is surprisingly messy however, and will involve the copying of data from the application into a suitably sized mobile BYTE array. In addition to this, the protocol-converters must be modified whenever the protocol on which they operate changes (for example, the addition of a tag in a variant protocol). For example, a simple sequential protocol and its (outgoing) converter-component might be:

```
PROTOCOL TIMED.DATA IS INT; REAL64:

PROC pcvt.timed.data (CHAN TIMED.DATA in?, CHAN MOBILE []BYTE out!)
  WHILE TRUE
    INT d.t:
    REAL64 d.v:
    in ?? d.t; d.v
      MOBILE []BYTE array:
      SEQ

        array := MOBILE [12]BYTE

        VAL [4]BYTE a.t RETYPES d.t:
        [array FOR 4] := a.t

        VAL [8]BYTE a.v RETYPES d.v:
        [array FROM 4] := a.v

        out ! array
:
```

Having to resort to writing code like this is a burden, and becomes a problem if we wish to construct a generic networking infrastructure, since users of those facilities will need to provide their own protocol-conversion processes (for any PROTOCOLs or DATA TYPEs that they define). Such code must be rigorously correct, if not to introduce hard-to-find bugs (for example, the correct placement of data within the output array). The above example is non-terminating; however, adding support for this is relatively trivial (the addition of an ALT).

For larger data-types, the overhead of copying (in and out of the externally communicated mobile array) becomes damaging to performance, and in many cases, this overhead is unnecessary. If the communication is correctly extended, the outputting application process will not complete the output until the remote (application) process has received the data. This allows a potentially significant optimisation in the outputting side: using a pointer to the data in the network communication process, rather than the data itself — thus avoiding any copying between the application and the low-level call to send data over a network.

Extracting a pointer in occam is non-trivial — and impossible without the use of inline (extended Transputer) ASM blocks. This leads to horribly complex protocol converters, discouraging the use of any such distribution mechanism. To access pointers from communication easily requires interaction outside of occam. For instance, specific programming in the run-time kernel (to intercept communication) has already been used by Vella [VW99, Vel98] to provide network communication. The flexibility of such infrastructures is limited however, since all network handling is performed by the run-time system. The user-defined channel (section 6.4) and C-interface (section 7.1) extensions presented in this thesis allow for more generic external interaction, but require a good knowledge of

C programming.

To address these issues, built-in PROCs have been added to the occam compiler that provide seamless support for protocol-conversion, providing automatic implementations for the 'P(a)', 'P(b)' and 'P(c)' processes from figure 7.8. Section 7.4.1 describes how these converters are accessed and used; the implementation is covered in section 7.4.3. Having the compiler generate the code required is significantly beneficial: it can handle *any* defined protocol, and will always be correct[8].

7.4.1 Accessing and Using Protocol Converters

The compiler-generated protocol converters are accessed through the two special PROCs:

```
PROC DECODE.CHANNEL (CHAN * in?, CHAN ** term?, CHAN *** decode.out!)
PROC ENCODE.CHANNEL (CHAN *** encode.in?, CHAN ** term?, CHAN * out!)
```

These have a somewhat generic interface, allowing channels carrying various protocols (represented by asterisks) to be connected. The single asterisk protocol ('*') is the user/application protocol. This is the most generic, allowing any channel to be connected, with the exception that channels carrying mobile channel-types (channel-bundle ends) may not be connected. This restriction is imposed, currently, because there is an issue of how these should be handled, particularly when SHARED channel-ends are involved. These issues are not directly within the scope of this thesis, but once addressed, support for them can be added mostly transparently — some thoughts are offered in section 8.4.2 however.

The two-asterisk channel protocol ('**') is for the termination channel. The way in which the implementation handles this channel (it never actually communicates), permits the use of any non-mobile, single-item protocol. For most purposes, this will typically be either 'INT' or 'BOOL'.

The three-asterisk protocol ('***') represents the higher-level protocol, consisting of two INTs. The channel used can either be of a user-defined PROTOCOL consisting of the two INTs, or of a fixed-size array: [2]INT. The two values communicated are a pointer to the originally outputted data, and the size of that data (in bytes). Communicating a pointer is safe, provided that the output from 'DECODE.CHANNEL' is extended as far as its actual transmission on the network (or, more likely, until it is transferred to a buffer in the TCP/IP stack). This is the only requirement of 'DECODE.CHANNEL'. To retain single-channel synchronisation, the communication must be further extended to the remote 'ENCODE.CHANNEL' process and into the remote application process.

The receiving side, using 'ENCODE.CHANNEL', is slightly more complicated, requiring that the pointer and size communicated to it (on 'encode.in') is that of a *dynamic mobile array* (of BYTEs). This fits the intended usage quite nicely however, since data received at the network-level will most likely be placed in a BYTE array.

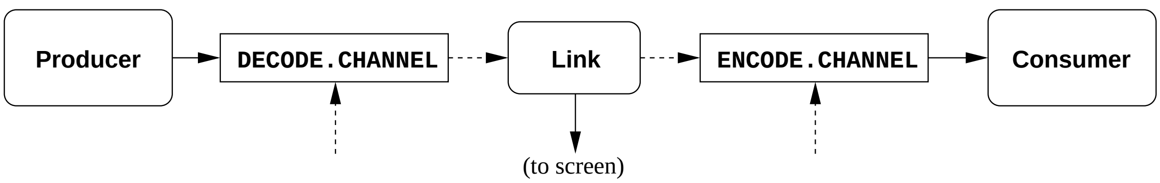


Figure 7.9: Example process network using compiler-generated protocol converters

[8]Assuming the absence of any compiler bugs.

Figure 7.9 shows a process network that uses 'DECODE.CHANNEL' and 'ENCODE.CHANNEL' locally, with a 'Link' process between them that just displays the communicated data (for debugging), before passing it on. The code for the overall network is simple, particularly the use of 'DECODE.CHANNEL' and 'ENCODE.CHANNEL' for users of any generic networking infrastructures provided. For example:

```
PROTOCOL LINK.P IS INT; INT:      -- protocol between decode/encode

PROTOCOL APP.PROTO IS
  CASE
    empty
    simple; INT
    sequential; REAL64; INT::[]BYTE
    complex; MOBILE [][]INT
:

...  producer, consumer and link processes

PROC link.example (CHAN BYTE screen!)
  CHAN APP.PROTO from.producer, to.consumer:
  CHAN LINK.P to.link, from.link:
  [2]CHAN BOOL kill:
  PAR
    producer (from.producer!)
    DECODE.CHANNEL (from.producer?, kill[0]?, to.link!)
    link (to.link?, screen!, from.link!)
    ENCODE.CHANNEL (from.link?, kill[1]?, to.consumer!)
    consumer (to.consumer?)
:
```

In order to program the 'Link' process, a further ability to manipulate dynamic mobile arrays is required. That is, the ability to (temporarily) transform outputs from 'DECODE.CHANNEL' into dynamic mobile arrays, and for the other side, the ability to turn a dynamic mobile array into an address/size pair for input by 'ENCODE.CHANNEL'. The following section describes the functionality provided to do this, followed by the implementation of the example 'link' process.

7.4.2 Detaching and Attaching Dynamic Mobile Arrays

In order to meet the requirements of 'ENCODE.CHANNEL', the 'link' process must generate a dynamic mobile array containing a copy of the originally communicated data. To make this possible, two additional built-in PROCs have been created, that provide a mechanism for interchanging between address/size pairs and dynamic mobile arrays:

```
PROC DETACH.DYNMOB (MOBILE []BYTE data, RESULT INT addr, size)
PROC ATTACH.DYNMOB (VAL INT addr, size, RESULT MOBILE []BYTE data)
```

The use of these built-in PROCs is intended for the network-infrastructure provider only. The (trivial) implementations of these make no attempt to check the validity of the address, leaving the potential for error quite high. Furthermore, the possibility of alias creation exists, which if uncontrolled could result in catastrophic run-time errors. In an attempt to limit potential damage, these built-in PROCs have subtle effects on the undefinedness checker (section 4.6).

For 'DETACH.DYNMOB', the dynamic mobile BYTE array detached is afterwards considered undefined — in addition to actually being made undefined at run-time (by having its size-field set to zero). In a similar manner, the VAL INT address parameter of 'ATTACH.DYNMOB' (or any base variable in an expression), is afterwards considered undefined. This causes the compiler to generate warnings if, for example, the same address is attached twice.

Implementing The 'Link' Process

The 'link' process from figure 7.9, using 'ATTACH.DYNMOB' and 'DETACH.DYNMOB', can be implemented with:

```
PROC link (CHAN LINK.P in?, CHAN BYTE screen!, CHAN LINK.P out!)
  WHILE TRUE
    INT addr, size:
    in ?? addr; size
      MOBILE []BYTE d.tmp, d.new:
      SEQ
        -- temporarily attach as dynamic mobile, copy, and detach
        ATTACH.DYNMOB (addr, size, d.tmp)
        d.new := CLONE d.tmp
        DETACH.DYNMOB (d.tmp, addr, size)

        -- report
        showdata (d.new, screen!)

        -- detach and send to encode process
        DETACH.DYNMOB (d.new, addr, size)
        out ! addr; size
:
```

The new dynamic mobile created by this process (using CLONE) is possibly freed by the 'ENCODE.CHANNEL' process. For the example 'APP.PROTO', this will happen for all cases *except* the 'MOBILE [][]INT' communicated by the 'complex' tag. In this case, the dynamic mobile is not freed, but instead is transferred to the application.

In all cases, the programmer should assume as little as possible about the pointer/size pairs produced by 'DECODE.CHANNEL', except that they are valid pointers to their respective data (BYTE) sizes. The critical requirement is that any use of the pointer and size is extended from the output by 'DECODE.CHANNEL' — i.e. the (outputting) user process must not be allowed to resume until the decoded pointer and size has been fully used.

7.4.3 Implementation of Protocol Conversion

Protocol conversion is implemented by in-line expanding the 'DECODE.CHANNEL' or 'ENCODE.CHANNEL' call into a suitable occam process. The generated process is primarily a loop with a PRI ALT over the 'term' and 'in' channels. A communication on the 'term' channel simply arranges for the process to stop looping. The code generated inside the body of the ALT guard for 'in' is substantially more complex.

The following shows the common code generated for both 'DECODE.CHANNEL' and 'ENCODE.CHANNEL'. This includes a number of temporaries, used during the decoding and encoding. More temporaries may be added to this later, for protocol handling that requires them.

```
BOOL $ed.anon:
INT $ed.addr:
INT $ed.count:                -- other temporaries below here
SEQ
  $ed.anon := TRUE
  WHILE $ed.anon
    PRI ALT
      term ??                 -- no input
        SEQ
          $ed.anon := FALSE
          SECOND.HALF term ?? -- release terminating process
      in ??                   -- no input
        SEQ
          ...  protocol specific conversion code
```

Because of the somewhat specialist nature of decode and encode, a special *action-node* has been added to the compiler — 'X.INPUT.OUTPUT'. This performs parts of both 'INPUT' and 'OUTPUT' action-nodes, but is different enough to warrant its own "node tag" within the compiler.

Like other action-nodes, 'X.INPUT.OUTPUT' has a left-hand-side (set to the input channel 'in'), a right-hand-side (set to a list starting with the output channel 'out'), and an action-type (set to an expression that represents the size of the communication).

The list on the right-hand-side, starting with the 'out' channel, contains the various '$ed.' temporaries, as well as other temporaries required by that particular 'X.INPUT.OUTPUT' node. Additionally, action-nodes contain a set of flags that control some aspects of code-generation. This was added primarily to support the extended rendezvous (section 3.7), and the optimisation of skipping XABLE for ALTing (extended) inputs. For 'X.INPUT.OUTPUT', these flags indicate whether the action is for 'DECODE.CHANNEL' or 'ENCODE.CHANNEL', and whether the operation involves a mobile or dynamic-mobile protocol.

A sizeable amount of code within the tree transformation part of the occam compiler is responsible for generating sequences of 'X.INPUT.OUTPUT' (and other code) from protocol trees. The following sections describe the transformation and code-generation for the various protocol types, starting with the operation of 'DECODE.CHANNEL' — that is simpler than 'ENCODE.CHANNEL'.

The handling of variant (tagged) and mobile protocols are discussed individually, since these require particular care in the implementation.

Decoding Non-Mobile Simple Protocols

Simple protocols, such as 'INT', 'REAL64', user-defined types and fixed-size array types are communicated using only one communication. For 'DECODE.CHANNEL', this is handled by a single 'X.INPUT.OUTPUT' node. Figure 7.10 shows the tree structure generated from transforming an instance of 'DECODE.CHANNEL', where the 'in' channel is of type 'CHAN INT'.

New code within the back end of the compiler takes this tree structure (starting at the 'X.INPUT.OUTPUT' node), and generates code that performs the actual decode operation. For a simple 'CHAN INT' decode, where the output channel is two sequentially communicated INTs, this is:

```
--{{{  wait for outputting process
-- LD ADDRESSOF in
-- XABLE                  -- XABLE skipped
--}}}
```

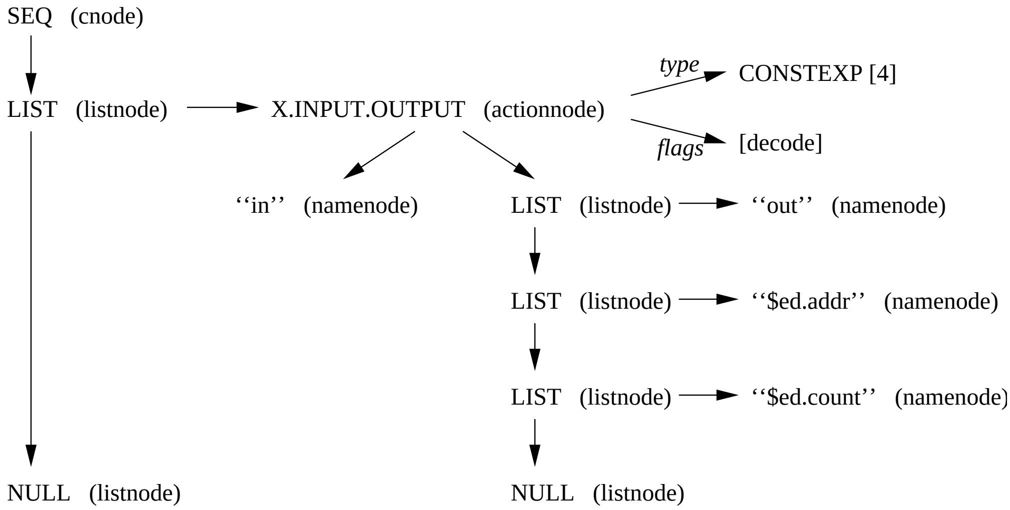


Figure 7.10: Tree structure generated for decoding a simple `INT` channel

```
--{{{  communicate address
LD ADDRESSOF out          -- load output channel
LD ADDRESSOF in           -- load input channel
LDNL 0                    -- load outputting process
LDNL -4                   -- load outputting process (data) pointer
OUTWORD                   -- output data address
--}}}

--{{{  communicate count
LD ADDRESSOF out          -- load output channel
LDC 4                     -- load BYTESIN (INT)
OUTWORD                   -- output size
--}}}

--{{{  resume outputting process
LD ADDRESSOF in
XEND
--}}}
```

The code generated for a simple '`INT`' channel decode does not use either of the '`$ed.addr`' or '`$ed.count`' temporaries. More complex code generation may require these temporaries however.

In this example, the '`XABLE`' call at the beginning of the decode is skipped — since it is the first communication after the `ALT`. The implementation of this is handled in the same way as it is for extended inputs (section 3.7.2), by setting the 'skip-xable' flag in the action-node.

Decoding Sequential and Counted-Array Protocols

The handling of sequential protocol decoding is reasonably trivial — each component of the protocol is decoded sequentially, added into the list shown of the left-hand side of figure 7.10. Ordinary (and extended) sequential protocol inputs undergo a similar transformation, but are converted into sequences of communications, rather than sequences of '`X_INPUT_OUTPUT`'.

Counted-array protocols, such as '`INT::[]REAL64`' are slightly more complex. These are communicated as two separate items: the '`INT`' number of elements, followed by the '`REAL64`' array. Figure 7.11 shows the tree structure generated by the compiler for this type of protocol. A fresh temporary is used to store the count, whose type is the count part of the protocol ('`INT`' in this example).

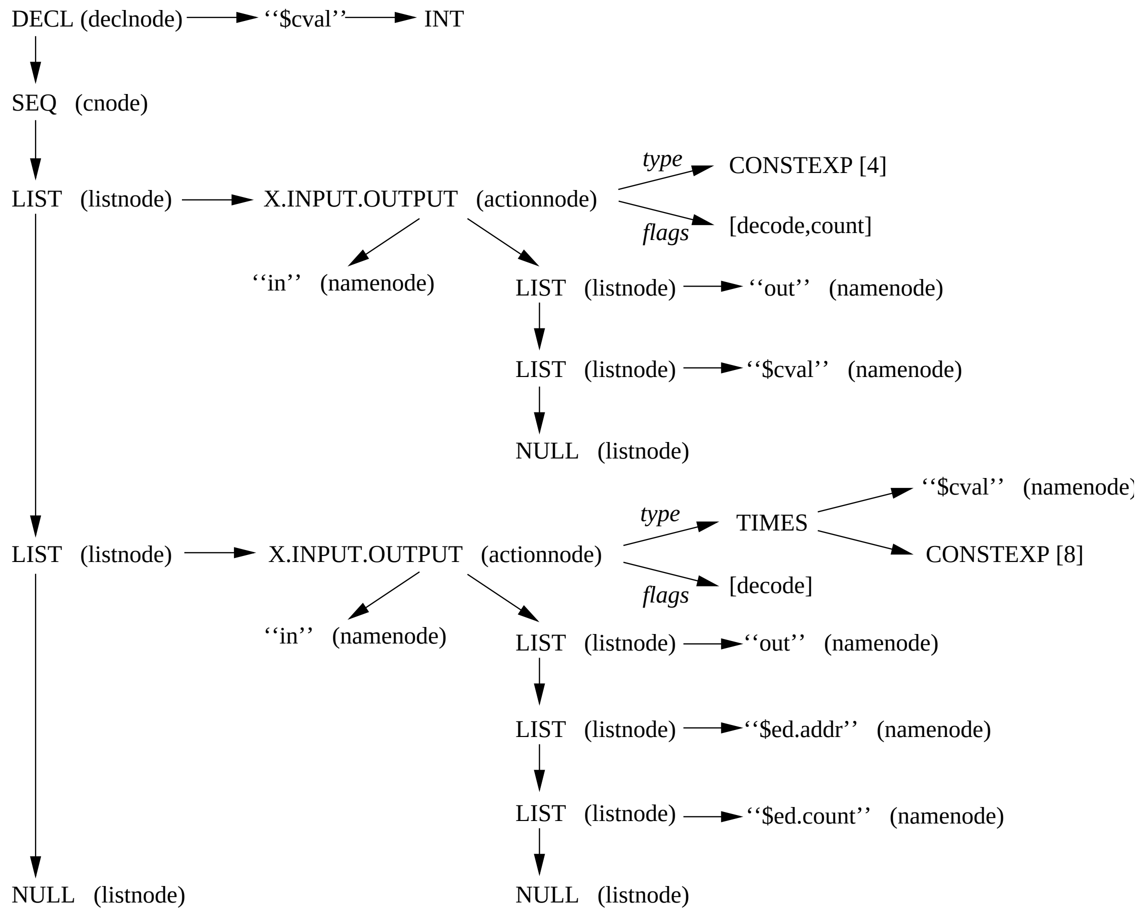


Figure 7.11: Tree structure generated for decoding an `INT::[]REAL64` counted-array channel

To distinguish the handling of the count part (that needs to store the value into '`$cval`'), the 'counted' flag is set in the action-node. The code-generator, instead of only generating code to communicate the address and size of the count, performs an extended input (using '`XIN`') into '`$cval`'. The code generated is similar to that shown above, with the following inserted just below any extended-enable:

```
LD ADDRESSOF $cval          -- address of temporary
LD ADDRESSOF in             -- load input channel
LDC 4                       -- load BYTESIN (INT)
XIN                         -- extended input
```

The code generation for the data-part of the counted-array protocol follows that of simple protocol generation, except that the count is an expression involving the temporary '`cval`'. This expression is typically trivial however, involving at most one '`TIMES`' (unchecked multiply) operation. Currently, there is no support in occam for handling multi-dimensional counted-arrays. Given the layout of multi-dimensional arrays, only the highest dimension is a candidate for counted-arrays; all other dimensions must be known constant sizes, for example: '`INT::[][16]REAL64`'. This is no more complicated than basic counted-array handling — just that the base-type is now '`[16]REAL64`', but is still a known constant size.

Encoding Simple, Sequential and Counted-Array Protocols

The encoding of channels is somewhat more complicated than decoding, in terms of code generated. The tree structures generated by '`ENCODE.CHANNEL`' for simple protocol encoding are essentially the same as that shown in figure 7.10, except that they carry the 'encode' flag, rather than 'decode'.

As discussed earlier, '`ENCODE.CHANNEL`' takes as input pairs of addresses and sizes, and those pairs must represent a dynamic mobile `BYTE` array. This fits particularly well for one percieved use of this mechanism — connecting channels over networks — since incoming data is likely to be recieved into a dynamic mobile array. For back-to-back connection, this requirement actually complicates things — as demonstrated by the earlier '`link`' process (from figure 7.9).

For simple protocols, the two `INT`s carried on the '`in`' channel are input into the temporaries '`$ed.addr`' and '`$ed.count`'. The first input is performed using a standard `INT`-size input, into '`$ed.addr`':

```
LD ADDRESSOF $ed.addr       -- destination address
LD ADDRESSOF in             -- load input channel
LDC 4                       -- INT communication
IN                          -- channel input
```

To maintain the extended communication, the second input is done in three steps: extended enable, extended input and, after the real output, extended end. If the size is a known constant then the extended-input into '`$ed.count`' can be skipped entirely, as is the case for all simple protocols. For example, the code generated for handling the (constant) count input of a simple '`INT`' protocol is:

```
LD ADDRESSOF in             -- load input channel
XABLE                       -- wait for outputting process

--{{{  skipped extended input for count
-- LD ADDRESSOF $ed.count   -- destination address
-- LD ADDRESSOF in          -- load input channel
-- LDC 4                    -- BYTESIN (INT)
-- XIN                      -- extended input
--}}}
```

The next step is to perform the actual communication back into a user application, whilst the outputting process (communicating the two '`INT`'s) remains blocked. This is simply an ordinary output in most cases, with the only exception being mobile types — covered separately in the following sections. For a simple '`INT`' channel, where the size is a known constant (4), the following code is generated:

```
LD $ed.addr                 -- load (dynamic mobile) address
LD ADDRESSOF out            -- load output channel
LDC 4                       -- BYTESIN (INT)
OUT
```

If the size of the communication is not a known constant, then the value stored in '`$ed.count`' is used. Either this is set by an extended-input, as has been skipped above, or by evaluating the count expression on the '`X_INPUT_OUTPUT`' node, which is available and correct in any case. Although not done by default this can be used as a run-time check to ensure that the size communicated (of the dynamic mobile '`BYTE`' array) is the size expected (for user-output). As it happens, the compiler will always evaluate the (possibly constant) size expression (the type of the action-node), and simply does not input the 'size' value sent, or inputs and discards it. This makes coding a network receiver process slightly simpler, since it can communicate the size of a detached dynamic mobile, rather than the size of the actual communication. However, the application using this infrastructure *must* be aware of the sizes involved to a certain extent, to ensure that all data involved has been received by the encoding side.

Once the data has been output into the user application, the dynamic mobile from which it came can be freed, and the blocked outputting process resumed. This is done using the following code:

```
LD $ed.addr             -- load (dynamic mobile) address
MRELEASE                -- free memory
LD ADDRESSOF in         -- load input address
XEND                    -- resume outputting process
```

Counted-array protocol encoding is performed in a similar manner to the decode — the two items are handled separately, and the inputted count *data* is stored in a temporary, '`$cval`', as shown in figure 7.11. For counted-array encoding, a small optimisation is possible for the count part: the size '`INT`' can be input using a standard '`IN`' instruction, reading into '`$ed.count`'. This is safe since it is known that another communication follows immediately, right back to the original outputting process, connected to an instance of '`DECODE.CHANNEL`'.

Sequential protocols are simply generated as their components in sequence, in much the same way as for '`DECODE.CHANNEL`'.

Variant Protocol Handling

The handling of variant protocols is substantially less trivial than the previous protocol types described, but no more complex than its handling for the extended-input in general (described in section 3.7.2). For this type of protocol, the outputting process sends the appropriate 'tag' followed by the sequential protocol data (that may be nothing — i.e. empty tags). There are two forms of input allowed on a variant protocol channel — one that expects a specific tag and one that allows a selection based on the tag. For example, the variant protocol:

```
PROTOCOL USER
  CASE
    empty
    some.int; INT
:
```

can be input by either expecting a specific tag, "`in ? CASE empty`", or by selection, allowing handling of any or all tags:

```
in ? CASE
  empty
    ...  empty handling
  INT v:
  some.int; v
    ...  int data handling
```

The implementations for both forms of variant input are essentially the same — input the tag value, then perform an action (possibly more input) based on its value. Output is somewhat simpler — output the tag, followed by items for the sequential protocol (if not empty).

The implementations of `DECODE.CHANNEL` and `ENCODE.CHANNEL` for variant protocols are largely similar in structure. For decode, the tag is inputted (using the extended rendezvous) and its value outputted (as a pointer/size pair), before performing a '`CASE`' selection on the tag, to decode any internal sequential protocol. Encoding follows a similar pattern, first performing an extended input on the tag, outputting that to the application (in the extended process), and then executing a `CASE` selection to encode any following sequential protocol. As discussed with the extended rendezvous (section 3.7), if the protocol contains no empty tags, the initial tag input need not be extended — which allows for a slight simplification (and marginal overhead reduction) in the handling of these.

For the above tagged protocol, '`USER`', the code generated for a call to '`DECODE.CHANNEL (in?, term?, out!)`' is implemented by the following (pseudo occam `ALT` body):

```
...  ALT handling for 'term' channel
BYTE $ttmp:                -- tag temporary
in ?? $ttmp
  out ! ADDRESSOF $ttmp; SIZEOF $ttmp
  CASE $ttmp
    empty
      SKIP                 -- nothing to do
    some.int
      in ?? ($ed.addr, $ed.count)
        out ! $ed.addr; $ed.count
```

The parse-tree created by the translation of variable protocols is similar to the above occam fragment, but uses '`X_INPUT_OUTPUT`' nodes where extended-inputs are shown. The handling of any nested sequential protocols is essentially the same as previously described — simply the sequential decoding of its components.

Encoding variant protocols is almost exactly the reverse — as would be expected. The first communication (a dynamic mobile byte array) will be the tag value. In a similar way to the encoding of the counted-array 'count', the tag is input into a local temporary '`$ttmp`' and communicated to the user application, whilst in the extended input. A '`CASE`' selection is then made on the tag value, and the encoding of nested sequential protocols is handled as before.

Mobile Protocol Handling

The handling of (static) mobile protocols is slightly obscure, due to the way in which mobile communication is performed (using the '`MIN`', '`XMIN`' and '`MOUT`' instructions).

Mobile outputs are performed using the '`MOUT`' instruction, that takes a *pointer* to the workspace slot where the actual data pointer (into mobilespace) is allocated. As with other communication instructions, it is the second party to the synchronisation that performs the actual communication. For mobiles, using '`XIN`' (where the outputting process had previously blocked in the channel), the operation is a simple exchanging of the *data* pointers — accessed through the pointers passed in the communication.

The general implementation of '`DECODE.CHANNEL`' — using the extended rendezvous — ensures that the channel is ready when the protocol conversion code runs. Instead of swapping pointers, '`DECODE.CHANNEL`' simply outputs the pointer (accessed by de-referencing the pointer left in the outputting process's workspace), and the count (that is always a fixed size within mobilespace). For example, the (core) code generated for decoding a simple "`MOBILE [128]BYTE`" channel is:

```
--{{{  wait for outputting process
LD ADDRESSOF in            -- load input channel
XABLE                      -- wait
--}}}

--{{{  communicate address
LD ADDRESSOF out           -- load output channel
LD in                      -- load outputting process (from input channel)
LDNL -4                    -- load outputting workspace pointer (MOUT)
LDNL 0                     -- load data pointer (in mobilespace)
OUTWORD                    -- output data address
--}}}

--{{{  communicate count
LD ADDRESSOF out           -- load output channel
LDC 128                    -- load BYTESIN ([128]BYTE)
OUTWORD                    -- output size
--}}}

--{{{  resume outputting process
LD ADDRESSOF in
XEND
--}}}
```

Handling of the corresponding encode is slightly trickier. The encode process will receive a pointer and address to a *dynamic* mobile, containing data destined for mobilespace. Currently, there is no easy way for an outputting process to wait for its inputting counterpart, essentially an outputting `ALT` — forbidden in occam due to its high implementation cost.

Instead, a local temporary mobile is allocated for the '`ENCODE.CHANNEL`' process. Once the pointer has been inputted by the encode process, it is immediately copied into the local mobile temporary. The size is then inputted, using the extended input, and its contents verified. The encode process then simply performs the '`MOUT`' instruction, passing a pointer to the temporary mobile, then reschedules the blocked outputting process (that is communicating the size).

The following shows the pseudo-occam that implements the mobile-specific part of '`ENCODE.CHANNEL`' for the above '`MOBILE [128]BYTE`' protocol:

```
MOBILE [128]BYTE $mtmp:
SEQ
  in ? $ed.addr                -- read address
  $mtmp := [$ed.addr FOR 128]  -- copy data
  in ?? $ed.count
    SEQ
      ASSERT ($ed.count = 128)
      out ! $mtmp              -- mobile output
```

Currently, there is no support for handling nested mobiles, as there is no compiler support for nested *static* mobiles at present. However, supporting '`DECODE.CHANNEL`' and '`ENCODE.CHANNEL`' for these is relatively simple — multiple communications that communicate the various parts of the mobile structure.

Dynamic Mobile Protocol Handling

Currently, the only supported dynamic mobiles (for encode and decode) are arrays — of non-mobile types. Support for communication of (dynamic) mobile channel-types is discussed in section 8.4.2, and is not currently implemented.

The handling of dynamic mobile array protocols is, more or less, a combination of similar handling for static mobiles and counted-array protocols, but with some simplifications. When outputting a dynamic mobile array, that might be input for 'DECODE.CHANNEL', a process uses either the 'MOUT64' or 'MOUTN' instruction. Both of these take a pointer into the process's workspace, that points at the slots allocated for a dynamic mobile array — first the pointer, followed by the dimension sizes in ascending workspace slots (as discussed in section 4.2.4). The 'MOUT64' instruction communicates the pointer and the (single) dimension. 'MOUTN' communicates the pointer and an arbitrary number of dimensions, given as an extra parameter to the instruction.

The decoding of a dynamic mobile array is handled in two parts. First, the dimension counts are communicated, followed by the actual data of the array (that is always allocated as a single block, regarless of the number of dimensions). Once outputted, the dynamic mobile array that was communicated is released. Unlike static mobiles, the outputting process for a dynamic mobile expects to completely lose the reference. For static mobiles, the current reference is lost, but a new one gained — when decoding a static mobile, the reference is left unchanged, as described in the previous section.

The code generated for 'DECODE.CHANNEL' is similar to that generated for static mobile decoding — performing an 'XABLE' to wait for the outputting process, followed by various outputs. The first pair of outputs communicates the dimensions of the dynamic mobile. For decoding a "MOBILE [][]REAL64" protocol, this first output pair would be:

```
--{{{  communicate dimension sizes address
LD ADDRESSOF out           -- load output channel
LD in                      -- load outputting process (from input channel)
LDNL -4                    -- load outputting workspace pointer (MOUT64/MOUTN)
LDNLP 1                    -- load address of dimension starts
OUTWORD                    -- output data address
--}}}

--{{{  communicate dimension sizes count
LD ADDRESSOF out           -- load output channel
LDC 8                      -- load count (2 dimensions * 4 bytes)
OUTWORD                    -- output count
--}}}
```

Before the data itself can be communicated, its size must be calculated. This is implemented as a sequential run of loads and multiplies, starting with the base-type size. Although a loop (for calculating the size) might generate less code, it only becomes more efficient (after native-code translation) for pathologically large numbers of dimensions. A local temporary is used to store the base (workspace) address of the dimension sizes.

For the earlier "MOBILE [][]REAL64" protocol, the code to communicate the data itself is:

```
--{{{  communicate address
LD ADDRESSOF out
LD in
LDNL -4
LDNL 0
OUTWORD
--}}}

--{{{  communicate size
LDC 8                       -- BYTESIN (REAL64)
LD in                       -- load address of outputting process
LDNL -4                     -- outputting process's pointer field
LDNLP 1                     -- address of 1st dim
STL 0                       -- store temp
LDL 0                       -- load temp
LDNL 0                      -- 1st dimension
MUL                         -- multiply
LDL 0
LDNL 1                      -- 2nd dimension
MUL                         -- multiply
LD ADDRESSOF out
REV
OUTWORD                     -- output count
--}}}
```

Following this, the communicated dynamic mobile is released, accessed through the '`in`' channel. All that remains then is to resume the blocked outputter, using a call to '`XEND`'.

Encoding is a somewhat simpler process. The process generated by '`ENCODE.CHANNEL`' for dynamic mobile array protocols first receives a dynamic mobile address and size containing the dimensions, followed by a pointer and size of the actual data, as another dynamic mobile. The encoding requires a dynamic mobile, although this does not consume any more space than its workspace slots. The dimensions, when inputted, are copied into the corresponding dimensions in the temporary. When the data address is received, it is simply put into the dynamic mobile temporary — and explicitly not released locally (unlike other data received by any '`ENCODE.CHANNEL`' — that is copied). An ordinary dynamic mobile output (either '`MOUT64`' or '`MOUTN`') is then used to communicate the temporary out of '`ENCODE.CHANNEL`' into the application.

When the size (of the data) is received, it is optionally checked against the product of the dimension counts and the base-type size. The blocked outputting process (communicating the size) is then resumed, using '`XEND`' as before.

In a perfectly correct infrastructure, '`DECODE.CHANNEL`'s and '`ENCODE.CHANNEL`'s should match exactly (with logically similar protocols either side). Any errors incurred between these — such as incorrectly sized communications — will generally lead to catastrophic run-time error. Thus it is the responsibility of the application, or infrastructure provider, to ensure the integrity of communication between '`DECODE.CHANNEL`' and '`ENCODE.CHANNEL`'.

7.5 A Pre-Processor for occam

A familiar feature in many programming languages is that of a pre-processor. For C and C++ in particular, extensive use is made of the pre-processor (‘cpp’ within the GNU ‘gcc’ suite), often without the programmer realising it (since ‘gcc’ invokes ‘cpp’ as part of the compilation sequence).

One of the primary uses of a pre-processor is to support conditional compilation — the inclusion or removal of code depending on compile-time system or application requirements. The ‘cpp’ pre-processor goes much further than this, providing complex ‘macro’ support, with often undesirable effects (you can take the address of a C function, but not that of a pre-processor macro).

Pre-processors are also frequently used to handle system-dependent aspects of compilation. For KRoC, this might represent the availability of certain language or run-time system features, for example, mobiles and blocking system-calls. Although it would be possible for the compiler/application to provide this information through ‘VAL’ constants, these are sometimes not sufficient — mainly when code must be completely excluded from compilation, or when constants are generated based on the value of another constant (occam currently has no support for a conditional operator, ‘?’ and ‘:’ in C/Java).

The pre-processor support added to occam here has been implemented within the compiler itself, rather than a separate utility. This allows indentation to be considered, and enforced, that is certainly desirable. The occam compiler, unlike many C compilers, has no mechanism to specify the filename or line-number undergoing compilation. These types of directive are generated by the ‘cpp’ pre-processor, such that compiler reports errors in the file before it was pre-processed. For occam, any external pre-processor would have to leave the positioning of lines unchanged — not technically a problem (as previous occam pre-processors have done this), but such approaches are not particularly graceful.

The new occam pre-processor supports the definition of named constants, condititional compilation and various other features. Macro support has not been implemented, both because it is largely unncessary (the ‘INLINE’ keyword on PROC and FUNCTION definitions suffices), and because macros, when found in other languages (such as C), are frequently abused.

Sections 7.5.1 and 7.5.2 cover the named constant support in the pre-processor, which will typically be required for conditional compilation, covered in section 7.5.3 along with indentation. Section 7.5.4 presents the additional facility for user-generated errors (as might be required occasionally).

7.5.1 Named Constants

Constants in the occam pre-processor are defined (and re-defined) using a familiar syntax:

```
#DEFINE USE.BLOCKING.SYSCALLS
#DEFINE HOSTNAME "pondweed"
#DEFINE PORT 1234
```

Three value-types for constants are supported: nothing, a string, or an integer. Constants defined without values are typically only useful for conditional compilation (section 7.5.3). Those defined with values may be used in ordinary occam too — literally substituted for the value they represent. To refer to a pre-processor define, the name is prefixed with ‘##’ — making it clear that this is *not* an ordinary occam name. For example:

```
socket.create.connect.tcp (sock, ##HOSTNAME, ##PORT)
```

Once defined, constants remain in scope for the rest of the current compilation, even if they are defined inside a PROC or FUNCTION definition. The removal of a constant is done in a similar manner, through the use of '#UNDEFINE':

```
#UNDEFINE USE.BLOCKING.SYSCALLS
#UNDEFINE PORT
```

7.5.2 Built-In Defines

When the compiler starts up, it defines various constants, supplying information about what features are supported by the compiler. This allows code to behave differently at compile-time, should it be required. In some cases it is merely useful — for example, having an application report what version of the compiler it was built with.

Table 7.3 shows the built-in defines that are added by the compiler. In addition to these are two special built-in defines: "FILE" and "LINE". These hold the current file-name (string) and line-number (integer) respectively, and are automatically updated by the compiler. These typically find use when reporting (serious) internal error conditions, for example:

```
IF
  gone.bad
    SEQ
      do.error ("suitable message", ##FILE, ##LINE)
      STOP
  TRUE
    SKIP
```

where the parameter list for 'PROC do.error' is "VAL []BYTE msg, filename, VAL INT linenum". The programmer may re-define these arbitrarily, but such practice is not recommended — the file-name is re-defined by the compiler whenever a new file is opened (for '#INCLUDE' directives), or whenever a previously opened file is resumed (following '#INCLUDE' and '#USE' directives). The line-number is re-defined at every line, so re-defining or un-defining it makes little sense — because the compiler will (re-)define it on next input line.

The special compiler option "-zpp" instructs the compiler to print out a list of defines, after it has parsed the top-level file. Even if the input fails to parse, this list will still be displayed (left at whatever state it was in when the parser aborted). The output generated is in '#DEFINE' form, such that it could be fed back in as input if required. For instance:

```
#DEFINE FILE "mct50.occ"
#DEFINE LINE 32
#DEFINE TARGET.VENDOR "pc"
#DEFINE TARGET.OS "linux-gnu"
#DEFINE TARGET.CPU "i686"
#DEFINE TARGET.CANONICAL "i686-pc-linux-gnu"
#DEFINE VERSION "OFA 1.3.3K"
#DEFINE INITIAL.DECL
#DEFINE MOBILES
#DEFINE USER.DEFINED.OPERATORS
#DEFINE OCCAM2.5
#DEFINE PROCESS.PRIORITY 32
```

Name	Type	Description
`PROCESS.PRIORITY`	integer	If defined, the number of process priority levels supported (section 7.2).
`OCCAM2.5`	*nothing*	If defined, indicates that support for user-defined types and other occam 2.1 features is available.
`USER.DEFINED.OPERATORS`	*nothing*	Defined if user-defined operators are supported.
`INITIAL.DECL`	*nothing*	Defined if `INITIAL` style declarations are supported.
`MOBILES`	*nothing*	Defined if (all) mobiles are supported (Chapter 4).
`BLOCKING.SYSCALLS`	*nothing*	Defined if blocking system-calls are supported (section 6.3).
`VERSION`	string	Compiler version string.
`NEED.QUAD.ALIGNMENT`	*nothing*	Defined if the target architecture requires quad-word (64-bit) alignment of data.
`TARGET.CANONICAL`	string	The canonical compiler target name — i.e. the host type that '`occ21`' was compiled for.
`TARGET.CPU`	string	The target CPU — i.e. the CPU type that the compiler runs on.
`TARGET.OS`	string	The target operating system — i.e. the operating-system which the compiler was built to run on.
`TARGET.VENDOR`	string	The target vendor — i.e. the hardware that the compiler runs on, typically "`pc`".

Table 7.3: Compiler generated pre-processor defines

7.5.3 Conditional Compilation and Indentation

Perhaps the most common use of a pre-processor is to support the inclusion and exclusion of code based on constant conditions. This pre-processor follows the existing '`cpp`' style, using '`#IF`', '`#ELIF`', '`#ELSE`' and '`#ENDIF`'. The constant expressions used for the '`#IF`' and '`#ELIF`' (else-if) directives are formed in a similar way to occam expressions, but specifically for use with the pre-processor *only* — such expressions may not refer to occam variables.

The formal syntax for pre-processor expressions is given in Appendix A.3, with the main reduction being '*pp.ifexp*':

```
pp.ifexp  =  TRUE
          |  FALSE
          |  DEFINED ( pp.name )
          |  pp.name pp.cmpop pp.name
          |  pp.name pp.cmpop pp.value
          |  pp.boolmop pp.ifexp
          |  pp.ifexp pp.booldop pp.ifexp
          |  ( pp.ifexp )
```

Parsing of these expressions is done in the occam style, such that left/right reduction ambiguities are errors. All cases for '*pp.ifexp*' reduce to a constant boolean value, that may be the explicit '`TRUE`' or '`FALSE`' constants. A special '`DEFINED`' function is provided that tests whether a particular name has been defined in the pre-processor.

The expression may also be a comparison between the value of a '*pp.name*', and either the value of another '*pp.name*', or a literal '*pp.value*'. In both cases, type-compabaility is insisted upon, and it is an error if any '*pp.name*' has an empty value.

More complex expressions can be formed using brackets, monadic and dyadic operators. The single '*pp.boolmop*' is 'NOT'. The two available '*pp.booldop*'s are 'AND' and 'OR'. The evaluation of expressions over 'AND' and 'OR' is lazy where possible, making possible expressions such as:

```
#IF (DEFINED (PROCESS.PRIORITY)) AND (PROCESS.PRIORITY < 16)
```

and:

```
#ELIF (NOT DEFINED (MAGIC)) OR (MAGIC <> 99)
```

Indentation becomes an issue when if-like constructs are available in the pre-processor. For example:

```
#IF DEFINED (PROCESS.PRIORITY)
  VAL INT num.levels IS ##PROCESS.PRIORITY:
#ELSE
  VAL INT num.levels IS 1:
#ENDIF

...  process using "num.levels"
```

After the pre-processor has done the necessary, the occam parser proper is left with a declaration that is indented. This will cause an indentation error when parsing the following process. Rather than having the pre-processor automatically adjust the indentation level within '#IF' constructs, a special directive is used to instruct the pre-processor to *relax* the indentation within the block. This approach is taken since it allows the use of flattened '#IF' constructs, as might be produced by automated software tools (that have no interest in indentation). The above could be written correctly as:

```
#IF DEFINED (PROCESS.PRIORITY)
  #RELAX
  VAL INT num.levels IS ##PROCESS.PRIORITY:
#ELSE
VAL INT num.levels IS 1:
#ENDIF

...  process using "num.levels"
```

The '#RELAX' directive may only occur once per block, and lasts until the following (non-indented) '#ELIF', '#ELSE' or '#ENDIF' directive. Use of '#RELAX' at the very start of the block is not compulsary, but is certainly recommended for clarity.

The above example shows the use of conditional compilation at an outermost level. However, directives can appear directly inside code, wherever required. For example:

```
WHILE TRUE
  #IF DEFINED (USE.PRIALT)
  PRI ALT
  #ELSE
  ALT
  #ENDIF
    INT x:
    in ? x
      ...  guarded process
    ...  other guards and processes
```

The use of indentation and '#RELAX' is unlikely to be beneficial here, given the use of the '#IF' directive (to select between different occam constructs).

7.5.4 User-Generated Errors

In many cases, it may be desirable for warnings or errors to be generated from within user code, if conditions for compilation are not right. For example:

```
#IF NOT DEFINED(PROCESS.PRIORITY)
  #ERROR This version is for priority-enabled code only
#ELIF PROCESS.PRIORITY < 32
  #WARNING Limited priority support
#ENDIF
```

These follow in a similar manner to 'cpp's '#warning' and '#error' directives, taking arbitrary text as an argument. The occam pre-processor will only perform substitutions within the text for defined names prefixed with the '##' operator. For example:

```
#IF DEFINED (PROCESS.PRIORITY) AND (PROCESS.PRIORITY < 32)
  #WARNING Limited priority support: ##PROCESS.PRIORITY levels
#ENDIF
```

The '#ERROR' directive, when actively encountered, aborts compilation with the given message. The '#WARNING' directive simply emits a compiler warning. In both cases, the error comes out alone (brief), without showing the surrounding program context.

Chapter 8

Conclusions and Further Work

This thesis has successfully demonstrated that high-performance dynamic concurrency is practical, with good properties concerning safety, flexibility and performance.

Safety is a critical requirement of concurrent programs, that the occam compiler enforces with strict static aliasing and parallel-usage checks. With the exception of the n-replicated `PAR` (section 5.2), the work presented in this thesis is also safe — i.e. it does not introduce any opportunity for aliasing or race-hazards errors.

From the viewpoint of standard occam [Inm95], the work within this thesis significantly improves the flexibility, and hence applicability, of occam. Of particular significance are the mechanisms that provide support for parallel process recursion, n-replication and 'forking' (covered in Chapter 5), and those that provide mobile data, channels and processes (Chapter 4). Much of this has been made possible by the addition of dynamic memory allocation to occam, also in this thesis (section 3.8).

In terms of real-world flexibility, the addition of support for blocking system-calls (section 6.3) is a significant gain. As are the mechanisms that provide priority (section 7.2), debugging (section 7.3) and support for concurrent interaction with C (using a familiar Inmos/CCSP interface).

The following section 8.1 examines parallels between occam and Object-Orientated (OO) languages. Following from this, section 8.2 describes some commonly found features in OO languages that might be desirable in a future occam (if at all).

The implementation of the work presented in this thesis is largely complete. The few incomplete parts (examined in section 8.3) represent a fairly significant amount of work, mostly within the occam compiler.

Using this work as a basis, the potential for future development is huge — both at the language level, and in building new applications that utilise this work. Section 8.4 examines several possible future (language) additions that are of particular interest, although there are and will be many more.

8.1 occam and Object Orientation

Although Object-Orientation (OO) provides a natural way of modelling problems, the majority of its commonly used implementations suffer from serious problems. OO in languages such as C++ [Str97a, Str97b] and Java [JGS96] work by encapsulating problem-domain *objects* in *classes*. A class is a type, that may be related to other classes through inheritance, forming a rich (and potentially complex) type system. An in-depth discussion on inheritance and polymorphism, with particular relation to type-systems, can be found in [CW85].

C++ and Java classes combine state (per-instance variables, known as *attributes* or *fields*) and functions (*methods*) to operate on that (local) state. The largely sequential flow-of-control in these languages results in object interactions that are based on *method calls*. Unless specifically engineered, objects are *dead* — brought to life by the active flow-of-control for a particular method call. Locke [Loc01] provides some interesting insights into the variety of problems that object-orientation (as embodied in languages such as C++ and Java) suffers from.

8.1.1 A Ring of Processes

A classic example is a ring of processes (not necessarily identical), with each component taking a value, performing some computation, then passing a new value onto the next process in the ring. The occam/CSP implementation is trivial — a ring of parallel processes connected using channels. The only real danger is that of deadlock, but this will only occur if the ring 'fills up', leaving each process blocked trying to communicate with the next process, that is blocked, and so on.

An implementation in C is also trivial, and would usually be a '`for(;;)`' (infinite) loop combined with two variables (value and state), with a '`switch()`' statement to select between 'process' implementations. Each time round the loop the 'state' value is simply incremented modulo the number of states. Other implementations are certainly possible, but this represents one of the simpler solutions.

For OO languages such as C++ and Java, the normal (object-oriented) approach would be to define classes for each different 'process', each containing a suitable compute method. The average Java student would almost certainly start by writing something similar to:

```
//  "ComputeProcess" is an abstract type
//  that only requires the implementation
//  of the compute method.

class Process1 extends ComputeProcess
{
    public int compute (int val)
    {
        ...  body of compute

        return val;
    }
}
```

Given another class that simply contains the starting '`main()`' method, the way forward is not always clear. Assuming an array of '`ComputeProcess`' objects is created and suitably initialised (to instances of the desired processes), one correct solution is to use a loop within the '`main()`' method that simply invokes '`compute`' on each object in turn. However, the concept of "a ring of processes" might tempt the student into a more 'object-based' approach, such that each process executes the '`compute`' method of the next one, and so on. Besides the problem of getting references to the 'next' process into each object, the resulting cyclic method calling results in significant and unnecessary memory consumption — each object is forced to remain 'active' whilst the next one computes.

Fortunately, the inefficiencies of this second implementation are nearly always obvious, and alternative implementations are sought. This problem arises simply because of an attempt to treat life-less *objects* as active *processes*. This is not a deficiency of object-orientation itself however, that principally provides only the type-system (including inheritance and polymorphism).

Fortunately, help is at hand for Java in the form of 'CSP for Java' (JCSP) [WSHB99, Wel99]. This provides a number of classes which provides CSP-style support for concurrency including, but not limited to, parallel-processes (with a barrier synchronisation), channels and alternation. Recent additions include a networking infrastructure for building distributed systems (again based on the CSP model) [WAF02].

JCSP adds the required element of concurrency to Java needed to implement the process-ring correctly, using *active objects* — that are JCSP processes. Instead of method-calling, interaction *between* objects is done using communication on channels, that accurately captures the desired functionality (sending a value to the next process in the ring). Although JCSP provides a powerful tool for building concurrent systems in Java, the programmer is still required to use that functionality correctly — and the problems associated with aliasing remain.

8.1.2 Process Types

The (mobile) process types introduced in section 4.5 are similar to OO 'objects', but lack the object-oriented type system.

An occam process type describes an *interface*, that particular mobile processes (equivalent to classes) implement. Thus, a mobile process has no (direct) type of its own, and does not form any part of a type hierarchy (which classes do provide, through inheritance). This gives mobile processes independence (encapsulation) — unless explicitly programmed, mobile processes do not depend on each other in any way. This is not necessarily the case for objects in C++ or Java — whose correct operation may depend on sub-types (and their implementations) introduced through inheritance.

8.2 Desirable OO Features for occam

Despite this author's dislike for common object-orientated languages[1], many OO implementations contain features which might be desirable for implementation in a future occam.

8.2.1 Objects

The concept of *objects* already exists in occam to a certain degree. An OO object (at the design stage) can easily be implemented as an occam `PROC` — with channels providing the interface for interacting with that object.

The mobile process-types extension (section 4.5) adds to this by allowing processes to be assigned to variables of the relevant *process-type* (i.e. the type which describes a public interface which the occam process implements). Processes can then be communicated around networks, similar to passing object-references in C++ and Java. Unlike C++ and Java however, communication of these does not allow aliases to be created — from the mobile semantics. Furthermore, the occam implementation restricts communication of mobile processes, such that they may not be *moved* (by communication or assignment) whilst active.

What is permitted is the active usage of mobile *channel-types* (section 4.3). These can be used to form the interface to a mobile process, such that the mobile process takes the *server-end*, servicing client requests from the possibly shared *client-end*. The client-end of the interface is then simply communicated and assigned as needed. Using a shared client-end permits controlled access to the shared server resource by many parallel clients.

[1] Largely brought about by problems encountered whilst developing a sizable C++ application — one memorable problem was how to allow an object to '`delete`' itself from one of its own methods.

8.2.2 Inheritance

Inheritance has been provided for, in part, with variant occam protocols (section 3.2). Inheritance in this way is not the same as that found in C++ and Java however. The practical purpose of inheritance in these languages is to allow *specialisation* — taking an existing object definition (class) and extending/enhancing it in some way.

occam processes can easily achieve the same functionality — if a `PROC` desires the underlying functionality of another `PROC`, it simply runs that other `PROC` in parallel with some interface code. Figure 8.1 shows an example of this, that might be implemented with the following occam code:

```
PROC my.process (CHAN INT in?, out!)
  ...  body of "my.process"
:

PROC my.process (CHAN INT in?, out!)
  CHAN INT i.in, i.out:           -- internal channels
  PAR
    ...  interface process
    my.process (i.in?, i.out!)
:
```

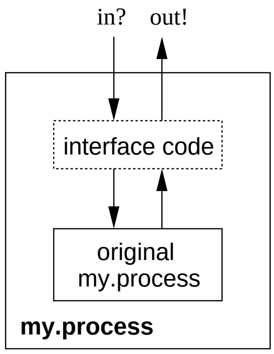


Figure 8.1: Implementing inheritance in occam by re-definition

The extended-rendezvous (section 3.7) can also be used to retain the original semantics of communication, making it impossible for the environment or the original '`my.process`' to detect the presence of the added interface code. To avoid introducing a possibility for deadlock, care must be taken in such interface implementations. Use of a separate interface process for each channel would prevent any *additional* opportunity for deadlock.

8.2.3 Polymorphism

Polymorphism is a feature found in a number of languages, at a number of different levels. C++ and Java support *type-polymorphism* (on object types) through the use of inheritance. They also support *function-polymorphism*, realised in function/method overloading. Support for `PROC` and `FUNCTION` overloading in occam is certainly possible — the envisaged technique would be to decorate the exported names of these with parameter and return-type information, known as *name-mangling* in C++'s implementation.

However, whether or not this is desirable is highly debatable. Particularly because there are no relationships imposed on the overloaded methods. They are utterly separate functions, but all with the same name. If used correctly, function/method overloading can be a very useful feature. However, the possibility for confusion created through misuse is definitely a good reason to *not* provide such support.

8.3 Future Work — Tidying Up

This section discusses some of the items presented in this thesis that remain incomplete in the implementation. These include arbitrary process `FORK`ing, support for fully nested mobilespaces, and the implementation of mobile processes; discussed in the following sections.

8.3.1 Arbitrary Process `FORK`ing

The initial idea for the `FORK` was to allow *any* process to be dynamically spawned, not just a `PROC` instance. As it turns out however, allowing only `PROC`s to be `FORK`ed solves a potential problem — how to deal with variable accesses within the `FORK`ed process. The semantics of `FORK` parameter passing is that of communication, restricting the `PROC` to `VAL` or `MOBILE` parameters only.

If arbitrary process `FORK`ing is to be supported, a mechanism for handling this must be provided. If an approach similar to that of '`INLINE PROC`'s is tried, the `FORK`ed process will be left with a series of (`VAL`) abbreviations before it (from the parameters). However, this does not solve the problem — `VAL` abbreviations use references for items larger than one word. '`INITIAL`' declarations [Moo99] do provide a nice solution, however, that works for `MOBILE`s as well as ordinary types. As a simple example, consider:

```
PROC send.message (SHARED CHAN BYTE out!)
  VAL []BYTE message IS "hello, world!*n":
  CLAIM out!
    SEQ i = 0 FOR SIZE message
      out ! message[i]
:

PROC main (SHARED CHAN BYTE scr!)
  FORKING
    SEQ
      FORK send.message (CLONE scr!)
      ...  continue processing in parallel
:
```

Ideally, it should be possible to implement this example within a single `PROC`. Using the method suggested above, this example could be implemented by:

```
PROC main (SHARED CHAN BYTE scr!)
  FORKING
    SEQ

      INITIAL SHARED CHAN BYTE out! IS CLONE scr!:
      FORK
        VAL []BYTE message IS "hello, world!*n":
        CLAIM out!
          SEQ i = 0 FOR SIZE
            out ! message[i]

      ...  continue processing in parallel
:
```

Whilst this example is, in theory (pending implementation), perfectly valid, the `INITIAL` abbreviation of '`CLONE scr!`' is unnecessary. Since that channel is explicitly `SHARED` (and `MOBILE` — it is an anonymous channel-type end), normal parallel usage rules do not apply. Instead, the compiler only checks parallel-usage within `CLAIM` blocks — where parallel usage is illegal. Furthermore, the compiler checks to ensure that any accesses to the shared channel are `CLAIM`ed.

8.3.2 Full Nested Mobilespace Support

Currently, support for nested `MOBILE` types is fairly limited. The two particular cases that are currently supported are: dynamic mobile arrays of mobile channel-ends; and dynamic mobile arrays within static mobiles.

Overall, there are four general cases of mobile nesting: static-in-static, dynamic-in-static, static-in-dynamic and dynamic-in-dynamic. The most trivial of these to implement is dynamic-in-static nesting. Dynamic mobiles have a fixed-size *local* memory requirement (typically in a process workspace), but this can easily be allocated within a static mobile. A common use for this type of nesting (and hence why it has been implemented already), is for packet handling within protocol stacks — a dynamic mobile array is used to hold the data, contained within a static mobile record that holds additional control information.

The most complex of these will likely be static-in-dynamic nesting, since dynamically allocated mobiles cannot be returned to the free-lists if they have interacted with the global mobilespace. Static-in-static nesting is related to this, but should be much simpler to implement — references will always remain within mobilespace. Dynamic-in-dynamic mobile nesting should also be relatively straightforward to implement. One part of this has already been implemented — dynamic mobile arrays of mobile channel-ends:

```
CHAN TYPE FOO
  MOBILE RECORD
    CHAN INT req?:
    CHAN INT repl!:
:

MOBILE []FOO? server.ends:
MOBILE []FOO! client.ends:
SEQ
  ...
```

There is currently no support, however, for nested dynamic mobile arrays, such as:

```
MOBILE []MOBILE []BYTE strings:
```

Fortunately, the work required to support all nested mobiles has already been done in part. In particular, the front-end of the compiler correctly processes most nested mobiles (where tried), and it is in the back-end (code-generator) where the implementation is needed most.

8.3.3 Implementation of Mobile Processes

As noted in its description in section 4.5, the implementation of mobile processes is incomplete. Some initial work has begun however — the compiler will correctly interpret '`PROC TYPE`' declarations and allow declarations of those types.

The envisaged implementation is one where the memory requirement of a particular mobile process is that of its persistent state, plus some additional house-keeping information. As they are described in section 4.5, the additional house-keeping information for mobile processes can be small — in theory, just the type-hash of the mobile process.

This is possible because the type-conversion (described on page 103) explicitly requires the name of the mobile process involved. To support conversion between process types without explicit mobile process naming requires that the state of a mobile process includes information about how to convert between interface types. This could be provided with a simple array of type-hash values (of `PROC TYPE`s) alongside information relating to that particular implementation (most importantly, where the code is located, and what the memory requirements of that particular implementation are).

For the programmer, this is a much more pleasant way of presenting type-conversion — and can be supported using a simple type-cast. For example:

```
PROC TYPE FOREVER IS (CHAN INT in?, out!):
PROC TYPE TERMINATING IS (CHAN INT in?, out!, CHAN BOOL term?):

FOREVER p:
TERMINATING q:
SEQ
  p := MOBILE mproc ()          -- create an instance of "mproc"
  q := TERMINATING p            -- convert to TERMINATING interface
```

The only disadvantages of providing support for this are a slightly larger memory requirement and an $O(n)$ cost for switching between interfaces, where n is the number of interfaces implemented by a particular '`MOBILE PROC`'.

8.4 Future Work — Moving On

The remainder of this chapter discusses some potential ideas for future language and compiler additions. Not included are items in currently on-going research, such as RMoX [BJV03] — that represents a potentially vast amount of future research.

8.4.1 Fault-Tolerance for Concurrent Systems

Fault-tolerance has always been a slightly tricky issue for occam programs. If a process stops responding to its environment (which will happen if the process terminates or generates a run-time error), any attempts at interaction with it will result in deadlock — a property that will generally spread throughout the process network until no runnable processes remain.

A related problem is that of graceful termination and resetting, described in depth in [Wel89]. A general solution suggested there is one of 'poison' signalling, where terminating processes communicate 'poison' to their neighbours, that send it on to their neighbours, and so on. However, care must be taken to avoid deadlock within the 'poison' signalling, particularly if the network is not obviously deadlock-free (a ring of processes, for example). One way to do this is by using the IO-PAR paradigm [WJW93] for 'poison' signalling.

Although this mechanism allows a component to detect termination of its neighbours, it does not provide any mechanism to indicate *failure* in a neighbour — where that process simply `STOP`s (a primitive process that starts, but never terminates).

The failure handling mechanism in C++ and Java uses *exceptions*. Locally, a process tries an action and handles failure by catching any exceptions thrown. In C++, all exceptions are programmer-generated, at some level. Array-bounds checking and other run-time checks simply do not exist, resulting in undefined behaviour when the program errors. In Java, exceptions are generally programmed explicitly. However, the run-time system also generates exceptions, common ones being '`null`' pointer exceptions and array-bounds exceptions.

Fortunately, CSP already provides the semantic model required, 'interrupts'. In CSP, the process $(P \bigtriangleup Q)$ behaves like P until the first event in Q occurs, then the process continues as Q. If Q is a process that synchronises on a failure-event (exception), then it becomes a failure-handling process — and the desired effect is achieved. One possible syntax for exception-handling in occam, borrowing from the familiar C++/Java syntax, might be:

```
TRY
  ...  process that may cause a run-time error
CATCH
  STOPERROR
    ...  process STOP'ed
  ELSE
    ...  process caused another error
```

Such a mechanism would enable a process to handle run-time failure locally, which it could then indicate to its environment through normal communication, or internally reset to a known state.

There are, however, many unresolved issues with the implementation of any such mechanism. In particular remain the problems of detecting the failure of neighbouring components, and the removal of 'errored' processes from the run-time system (that includes processes started in parallel within the '`TRY`' block).

8.4.2 Higher-Order Channel-Type Communication

Section 7.4 describes the compiler built-ins '`DECODE.CHANNEL`' and '`ENCODE.CHANNEL`', that allow communication to be 'detached' from occam, primarily to support communication over TCP/IP networks and other distributed infrastructures (where those infrastructures are programmed in occam, rather than being provided as part of the run-time system).

The only type of communication not supported by these are that of mobile channel-type ends. Because channel-ends are not data, they cannot be communicated externally — the correct operation of channel communication requires a shared memory space (in the KRoC implementation, based on the Transputer hardware implementation).

Semantically, the communication of a channel-end, or channel-ends, over a network is the *stretching* of those channels. Besides handling the local setup (and linking) with the networking infrastructure, steps must be taken to ensure that repeatedly communicated ends do not result in 'knots'. Fortunately, previous research has already addressed many of these issues, in the Icarus language [MM98b].

8.4.3 Scalar Types For occam

The addition of scalar types to occam has often been considered, and the implementation should be non-complex. Sometime similar to scalar types exists in variant communication, whose 'cases' could be considered elements of some scalar type.

The main motivation for scalar types is that they provide a mechanism for typing related constants. For example:

```
SCALAR TYPE COLOUR IS RED, GREEN, BLUE:

PROC print.text (VAL []BYTE str, VAL COLOUR col)
  ...  body of "print.text"
:
```

The usual way of representing this is to use a `BYTE` or `INT` and define a series of constants that represent the various possibilities. Thus:

```
VAL BYTE RED IS 0:
VAL BYTE GREEN IS 1:
VAL BYTE BLUE IS 2:

PROC print.text (VAL []BYTE str, VAL BYTE col)
   ...  body of "print.text"
:
```

However, in this version, there is no guarantee that the value of '`col`' passed to '`print.text`' is valid. In the scalar-typed version, there is this guarantee — nothing other than the constants '`RED`', '`GREEN`' or '`BLUE`' may be used as '`COLOUR`' values.

8.5 Concluding Remarks

This thesis has presented methods and algorithms that provide support for highly dynamic (and secure) parallel programming, and enhanced interaction between concurrent software systems and their external (operating-system) environments. Furthermore, this thesis has demonstrated that dynamic parallel programming can be both secure and fast — by minimising overheads, better performance can be obtained — with many operations measured in tens of nano-seconds.

Performance has particular relevance for process scheduling, since the scheduler incurs an overhead proportional to the parallel granularity of a system (putting aside cache effects). Alternative support for concurrent programming, as provided by POSIX or Java threads, for example, incurs massive overheads by comparison, even when efforts are made to optimise. Much of this is due to the lack of basic, and securely controlled, provision for concurrency in the language itself.

When carefully managed, concurrency can be a powerful ally, resulting in software systems that are compositional, scalable, and whose designs are easily understood.

Bibliography

[Ame99] American National Standards Institute. Programming languages – C, 1999, 1999. ISO/IEC 9899:1999.

[APR+96] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts. hcc: A Handel-C Compiler. Technical report, Oxford University Computing Laboratory, UK, August 1996.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, The Netherlands, 1984. ISBN: 0-444-87508-5.

[Bar92] Geoff Barrett. occam 3 Reference Manual. Technical report, Inmos Limited, March 1992. Available at: `http://wotug.ukc.ac.uk/parallel/occam/documentation/`.

[Bar00a] Fred Barnes. *Socket, File and Process Libraries for* `occam`. Computing Laboratory, University of Kent at Canterbury, June 2000. Available at: `http://www.cs.ukc.ac.uk/people/rpg/frmb2/documents/`.

[Bar00b] 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.

[Bar01] 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.

[Bar02] F.R.M. Barnes. User Defined Channels in `occam`. Technical report, Computing Laboratory, University of Kent at Canterbury, UK, April 2002.

[Bar03] F.R.M. Barnes. occwserv: an `occam` web-server. In *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, IOS Press, Amsterdam, The Netherlands, September 2003.

[BDv99a] M. Boosten, R.W. Dobinson, and P.D.V. van der Stok. Fine-Grain Parallel Processing on Commodity Platforms. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering*, pages 263–276. WoTUG, IOS Press, the Netherlands, April 1999. ISBN: 90-5199-480-X.

[BDv99b] M. Boosten, R.W. Dobinson, and P.D.V. van der Stok. MESH: MEssaging and ScHeduling for Fine-Grain Parallel Processing on Commodity Platforms. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'1999)*. CSREA press, June 1999. ISBN: 1-892512-15-7.

[Bet00] Richard Beton. libcsp – A Binding Mechanism for CSP Communication and Synchronisation in Multithreaded C Programs. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 239–250, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[BJV03] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, IOS Press, Amsterdam, The Netherlands, September 2003.

[But97] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, July 1997. ISBN: 0-201-63392-2.

[BW01a] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 243–264, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[BW01b] F.R.M. Barnes and P.H. Welch. Mobile Data Types for Communicating Processes. In *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, volume 1, pages 20–26. CSREA press, June 2001. ISBN: 1-892512-66-1.

[BW02a] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part I. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 331–361, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[BW02b] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part II. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 363–380, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[BW03] F.R.M. Barnes and P.H. Welch. Prioritised dynamic communicating and mobile processes. *IEE Proceedings – Software*, 150(2), April 2003.

[Bc01] Ralf Bchle. Linux/MIPS HOWTO, April 2001. Available at: `http://oss.sgi.com/mips/mips-howto.html`.

[CP99] B.M. Cook and R.M.A. Peel. Occam on field programmable gate arrays – steps towards the para-PC. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 211–228. WoTUG, IOS Press, the Netherlands, April 1999. ISBN: 90-5199-480-X.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[DHWN94] M. Debbage, M. Hill, S. Wykes, and Dennis Nicole. Southampton's portable occam compiler (SPOC). In R. Miles and A. Chalmers, editors, *Proceedings of WoTUG*

17: Progress in Transputer and Occam Research, volume 38 of *Concurrent Systems Engineering*. IOS Press, The Netherlands, April 1994. ISBN: 90-5199-163-0.

[ea82] J. Hennessy et al. The MIPS machine. In *Proceedings of COMPCON Spring '82*, pages 2–7, 1982.

[FA01] Jeffrey S. Foster and Alex Aiken. Checking programmer-specified non-aliasing. Technical report, University of California, Berkeley, Computer Science Division (EECS), October 2001. Report No. UCB/CSD-01-1160.

[For00] Formal Systems (Europe) Ltd., 3, Alfred Street, Oxford. OX1 4EH, UK. *FDR2 User Manual*, May 2000.

[GE90] J. Grosch and H. Emmelmann. A Tool Box for Compiler Construction. *LNCS*, 477:106–116, 1990.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GRS93] M.H. Goldsmith, A.W. Roscoe, and B.G.O. Scott. Denotational Semantics for occam2, Part 1. In *Transputer Communications*, volume 1 (2), pages 65–91. Wiley and Sons Ltd., UK, November 1993.

[GRS94] M.H. Goldsmith, A.W. Roscoe, and B.G.O. Scott. Denotational Semantics for occam2, Part 2. In *Transputer Communications*, volume 2 (1), pages 25–67. Wiley and Sons Ltd., UK, March 1994.

[Han95] Per Brinch Hansen. Efficient Parallel Recursion. *ACM SIGPLAN Notices*, 30(12):9–16, December 1995. Reprinted in: *The Origin of Concurrent Programming*, edited by Per Brinch Hansen, pp. 525-534, Springer, ISBN 0-387-95401-5. 2002.

[Han96] Per Brinch Hansen. Efficient Parallel Recursion. In Per Brinch Hansen, editor, *The Search for Simplicity: Essays in Parallel Programming*, pages 509–518. IEEE Computer Society, Los Alamitos, California, 1996. chapter 25.

[HMSS87] M. Homewood, D. May, D. Shepherd, and R. Shepherd. The IMS T800 Transputer. *IEEE Micro*, pages 10–26, October 1987.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN: 0-13-153271-5.

[HPJW92] P. Hudak, S.L. Peyton-Jones, and P. Walder. Report on the Programming Language Haskell: A Non-Strict Purely Functional Language, version 1.2. *ACM SIGPLAN notices*, 27(5), May 1992.

[Inm84a] Inmos Limited. *occam Programming Manual*. Prentice Hall, 1984.

[Inm84b] Inmos Limited. *occam2 Reference Manual*. Prentice Hall, 1984. ISBN: 0-13-629312-3.

[Inm88] Inmos Limited. *Transputer Reference Manual*. Prentice Hall, March 1988. ISBN: 0-13-929001-X.

[Inm93] Inmos Limited. *The T9000 Transputer Hardware Reference Manual*. SGS-Thompson Microelectronics, 1993.

[Inm95] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: `http://www.wotug.org/occam/`.

[Int96] 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).

[Int99] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999. Available at: `http://developer.intel.com/design/PentiumIII/manuals/`.

[JGS96] B. Joy, J. Gosling, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996. ISBN: 0-20-163451-1.

[JL97] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York, 1996, reprint 1997.

[JMG+02] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, California, USA, June 2002. Available at: `http://www.usenix.org/events/usenix02/jim.html`.

[JWB02] C.L. Jacobsen, P.H. Welch, and F.R.M. Barnes. Building graph-like process structures using forked dynamic process creation and shared mobile channel types. Private communication, December 2002.

[Kha02] Ashfaq A. Khan. *Practical Linux Programming: Device Drivers, Embedded Systems and the Internet*. Charles River Media, February 2002. ISBN: 1-58450-096-4.

[Law02] A.E. Lawrence. Acceptances, Behaviours and Infinite Activity in CSPP. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 17–38, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[LL73] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[Loc01] T.S. Locke. Towards a Viable Alternative to OO – extending the occam/CSP programming model. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 329–349, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[MM98a] David May and Henk L. Muller. Using Channels for Multimedia Communication. Technical report, University of Bristol, Department of Computer Science, February 1998.

[MM98b] Henk L. Muller and David May. A simple protocol to communicate channels over channels. In *EURO-PAR '98 Parallel Processing, LNCS 1470*, pages 591–600, Southampton, UK, September 1998. Springer Verlag.

[MM01] David May and Henk Muller. Copying, Moving and Borrowing semantics. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 15–26, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[Moo99] 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.

[Moo00a] J. Moores. Native JCSP – the CSP for Java Library with a Low-Overhead CSP Kernel. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 263–274, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[Moo00b] James Moores. *The Design and Implementation of occam/CSP Support for a Range of Languages and Platforms*. PhD thesis, The University of Kent at Canterbury, Canterbury, Kent. CT2 7NF, December 2000.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes – parts I and II. *Journal of Information and Computation*, 100:1–77, 1992. Available as technical report: ECS-LFCS-89-85/86, University of Edinburgh, UK.

[MTW93] 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.

[NDHW94] D.A. Nicole, M. Debbage, M. Hill, and S. Wykes. Southampton's Portable Occam Compiler (SPOC). In A.G. Chalmers and R. Miles, editors, *Proceedings of WoTUG-17: Progress in Transputer and Occam Research*, pages 40–55, 1994. ISBN: 90-5199-163-0.

[PM88] K.P. Park and K.W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[Poo92] Michael D. Poole. Fixed Maximal Depth Recursion in occam. In *OUG Newsletter*, number 16 in Concurrent Systems Engineering. IOS Press, Netherlands, January 1992.

[Poo96] M.D. Poole. Occam for all – two approaches to retargetting the INMOS compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 167–178. World occam and Transputer User Group, IOS Press, Netherlands, March 1996. ISBN: 90-5199-261-0.

[Poo98] M.D. Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In P.H. Welch and A.W.P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 187–198, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[Pro95] oFA Project. occam For All: Case for support, February 1995. Available online at: `http://wotug.ukc.ac.uk/parallel/occam/projects/occam-for-all/`.

[PSBR94] E. Ploeg, J. P. E. Sunter, A. W. P. Bakkers, and H. W. Roebbers. Dedicated multi-priority scheduling. In Roger Miles and Alan Chalmers, editors, *Proceedings of WoTUG-17: Progress in Transputer and Occam Research*, volume 38 of *Transputer and Occam Engineering*, pages 18–31. IOS Press, The Netherlands, April 1994. ISBN: 90-5199-163-0.

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8), August 1992. Originally appeared at Supercomputing '91. Available from `http://citeseer.nj.nec.com/pugh92omega.html`.

[PV02] K.S. Pedersen and B. Vinter. Java PastSet: A Structured Distributed Shared Memory System. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 97–108, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[PW95] William Pugh and David Wonnacott. Going Beyond Integer Programming with the Omega Test to Eliminate False Data Dependences. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):204–211, 1995. Available at `http://citeseer.nj.nec.com/pugh92going.html`.

[RC01] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly, `www.oreilly.com`, second edition, July 2001. ISBN: 0-59600-008-1.

[Reg02] Regents of the University of California at Berkeley. OpenBSD, November 2002. Available at:`http://openbsd.org/`.

[Ros97] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

[RR64] B. Randal and L.J. Russell. ALGOL 60 implementation, 1964.

[RSS95] K. Rothwell, G. Shaw, and A. Smith. Porting the INMOS occam compiler to the SPARC architecture. In unknown, editor, *Proceedings of WoTUG 17: Progress in Transputer and Occam Research*, volume 44 of *Transputer and Occam Engineering*. IOS Press, The Netherlands, April 1995. ISBN: 90-5199-222-X.

[SARW98] T. Sheen, A.R. Allen, A. Ripke, and S. Woo. oc-X: an optimising multiprocessor occam system for the PowerPC. In P.H. Welch and A.W.P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 167–186, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[SBW03] M. Schweigler, F.R.M. Barnes, and P.H. Welch. Flexible, Transparent and Dynamic occam Networking with KRoC.net. In *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, IOS Press, Amsterdam, The Netherlands, September 2003.

[SC96] Paul Singleton and Barry M. Cook. Southampton's portable occam compiler (SPOC): an occam-for-all perspective. Technical Report TR-96.03, Department of Computer Science, Keele University, January 1996. ISSN: 1353-7776, available at: `ftp://ftp.cs.keele.ac.uk/pub/techreports/1996/tr96-03.ps`.

[SGS94] SGS-Thompson Microelectronics Limited. *T9000 occam 2 Toolset Handbook*. SGS-Thompson Microelectronics Limited, 1994. Document number: 72 TDS 457 00.

[SLG⁺00] J. Squyres, A. Lumsdaine, W. George, J. Hagedorn, and J. Devaney. The interoperable message passing interface (IMPI) extensions to LAM/MPI. In *Proceedings of the MPI Developer's Conference*, Ithica, NY, 2000.

[Sta98] Richard M. Stallman. *Using and Porting GNU CC: Version 2.8*. Free Software Foundation, March 1998.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, June 1992. ISBN: 0-201-56317-7.

[Ste98] W. Richard Stevens. *UNIX Network Programming, Vol. 2: Interprocess Communication*. Prentice Hall, second edition, September 1998. ISBN: 0-13081-081-9.

[Str97a] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1997. ISBN: 0-20-153992-6.

[Str97b] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997. ISBN: .

[Sun90] V.S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[SWB90] J.P.E. Sunter, K.C.J. Wijbrans, and A.W.P. Bakkers. Cooperative Priority Scheduling in Occam. In H.S.M. Zedan, editor, *Proceedings of the 13th occam User Group Technical Meeting: Real-Time Systems with Transputers*, Transputer and Occam Engineering, pages 175–185. IOS Press, The Netherlands, September 1990. ISBN: 90-5199-041-3.

[Swe99] Dominic Sweetman. *See MIPS Run*. Morgan Kaufmann, San Francisco, CA, USA, 1999. ISBN: 1-55860-410-3.

[SY85] R. Strom and S. Yemini. The NIL distributed systems programming language: a status report. *j-SIGPLAN*, 20(5):36–44, May 1985.

[Tho95] Simon Thompson. *Miranda: The Craft of Functional Programming*. Addison Wesley, July 1995. ISBN: 0-201-42279-4. See also: `http://www.cs.ukc.ac.uk/pubs/1995/353`.

[Tho99] S.J. Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1999. ISBN: 0-201-34275-8.

[Tur85] D.A. Turner. Miranda – Non-strict functional programming with polymorphic types. In Jean-Pierre Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (Nancy, France)*, pages 1–16, Berlin, Germany, 1985. LNCS 201, Springer Verlag.

[US 83] US Department of Defense. *The Ada Programming Language Reference Manual*. US Government Printing Office, 1983. ANSI/MIL-STD-1815A-1983.

[Vel98] Kevin Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, The University of Kent at Canterbury, Canterbury, Kent. CT2 7NF, December 1998.

[Vin99] Brian Vinter. *PastSet: a Structured Distributed Shared Memory System*. PhD thesis, Troms University, 1999.

[VW99] K. Vella and P.H. Welch. CSP/occam on Shared Memory Multiprocessor Workstations. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 87–119. WoTUG, IOS Press, the Netherlands, April 1999. ISBN: 90-5199-480-X.

[WA99] P.H. Welch and P.D. Austin. The JCSP (CSP for Java) Home Page, 1999. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`.

[WAF02] P.H. Welch, J.R. Aldous, and J. Foster. CSP networking for java (JCSP.net). In P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra, and A.G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002. ISBN: 3-540-43593-X. See also: `http://www.cs.ukc.ac.uk/pubs/2002/1382`.

[WB00] D.C. Wood and F.R.M. Barnes. Post-Mortem Debugging in KRoC. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 179–192, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[WB01] P.H. Welch and F.R.M. Barnes. extending channel communication, December 2001. Private communication.

[WB02] P.H. Welch and F.R.M. Barnes. On making (`pri`) `alt` highly efficient. Private communication, November 2002.

[Wel85] P.H. Welch. Five Essays on Occam. *Occam User Group Newsletter*, 2, January 1985. Also Internal Report, Training Department, GEC Avionics Ltd., Airport Works, Rochester, KENT ME2 1XX. See also: `http://www.cs.ukc.ac.uk/pubs/1985/237`.

[Wel89] P.H. Welch. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*, pages 310–317, Enschede, Netherlands, April 1989. Occam User Group, IOS Press, Netherlands. ISBN 90 5199 007 3.

[Wel90] P.H. Welch. Multi-Priority Scheduling for Transputer-Based Real-Time Control. In H.S.M. Zedan, editor, *Proceedings of the 13th occam User Group Technical Meeting: Real-Time Systems with Transputers*, Transputer and Occam Engineering, pages 198–214. IOS Press, The Netherlands, September 1990. ISBN: 90-5199-041-3.

[Wel99] P.H. Welch. CSP for Java (JCSP), 1999. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`.

[Wel00] P.H. Welch. Process Oriented Design for Java – Concurrency for All. In *PDPTA 2000*, volume 1, pages 51–57. CSREA Press, June 2000. ISBN: 1-892512-52-1.

[Wel03] P.H. Welch. `ALT`ing on a `barrier` synchronisation. Private communication, March 2003.

[WJW93] P.H. Welch, G.R.R. Justo, and C.J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993*

World Transputer Congress, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press, Netherlands. ISBN 90-5199-140-1. See also: `http://www.cs.ukc.ac.uk/pubs/1993/279`.

[WL98] Adam K.L. Wong and Francis C.M. Lau. MALT: A Multiway Alternation Construct for occam. In P.H. Welch and A.W.P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 199–210, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[WM99] D.C. Wood and J. Moores. User-Defined Data Types and Operators in occam. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 121–146. WoTUG, IOS Press, the Netherlands, April 1999. ISBN: 90-5199-480-X.

[WMBW00] P.H. Welch, J. Moores, F.R.M. Barnes, and D.C. Wood. The KRoC Home Page, 2000. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`.

[Woo97] David C. Wood. KRoC — An Implementors' Guide. Internal Documentation, available by request to: `D.C.Wood@ukc.ac.uk`, 1997.

[Woo98] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent at Canterbury, August 1998.

[Woo00] D.C. Wood. An Experiment with Recursion in occam. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 193–204, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[WP97] Peter H. Welch and Michael D. Poole. occam for Multi-Processor DEC Alphas. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 152–174, Amsterdam, The Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press. ISBN: 90-5199-336-6.

[WSHB99] P.H. Welch, G.S. Stiles, G.H. Hilderink, and A.P. Bakkers. CSP for Java: Multithreading for All. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[WW96] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166. World occam and Transputer User Group, IOS Press, Netherlands, March 1996. ISBN: 90-5199-261-0.

[WW97] Peter H. Welch and David C. Wood. Higher Levels of Process Synchronisation. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 104–129, Amsterdam, The Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press. ISBN: 90-5199-336-6.

Part III

Appendices

The appendices provide supplemental information that are not part of the main thesis. They are provided primarily as a reference for future development, but also serve as useful references in their own right.

Appendix A presents the new ordered grammar for occam, incorporating all the various new language features. Appendix B provides a reference for the newly added ETC instructions (the extended Transputer code generated by the compiler).

Appendix A

Ordered Syntax

This appendix provides the BNF-style syntax for the various additions made to the occam language. The style and names used are consistent with the definitions given in earlier language definitions [Inm84b, Inm95, Bar92]. In the interests of completeness, a whole BNF grammar for the occam language currently supported is given, with additions and modifications from identified appropriately.

The following section, A.1, describes the basic constructs used for parsing — names, strings and numbers. These are used both by the core language grammar, in section A.2, and by the pre-processor in section A.3. The pre-processor is covered in detail in section 7.5.

Modifications and extensions from the occam 2.1 language are indicated using the key given in table A.1, below.

†	indicates extensions made to the language in [WM99]. A fuller description of the implementation of these can be found in [Moo00b].
‡	indicates extensions to the language presented in this thesis, with a reference to the relevant section.
$\Re$	indicates constructs no longer fully supported.

Table A.1: Key to ordered syntax additions

A.1 Names, Strings and Numbers

occam source code is essentially comprised of names, string and numbers. This section re-presents the definitions from §G.4 in the occam 2.1 reference manual [Inm95].

An occam '*name*' must begin with an alphabetic character and contain only letters, numbers or dots. A '*string*' is text enclosed between double-quotes, that may contain various escape sequences. Single characters are specified inside single-quotes, and may be an escaped character. Numbers fall into various categories, but are ultimately composed of either '*digits*' or '*hex.digits*'. More formally, with the help of regular expressions:

digits	=	`[0-9]`+
hex.digits	=	`[0-9 A-F]`+
name	=	`[a-z A-Z][a-z A-Z 0-9 .]`∗

A.2 Core Language Grammar

abbreviation = *name* `IS` *variable* `:`
| *specifier* *name* `IS` *variable* `:`
| `VAL` *name* `IS` *expression* `:`
| `VAL` *specifier* *name* `IS` *expression* `:`
| *name* `IS` *channel* `:`
| *specifier* *name* `IS` *channel* `:`
| *name* `IS` `[` $\{_1,$ *channel*$\}$ `]` `:`
| *specifier* *name* `IS` `[` $\{_1,$ *channel*$\}$ `]` `:`
| *name* `IS` *timer* `:`
| *specifier* *name* `IS` *timer* `:`
| *name* `IS` *port* `:`
| *specifier* *name* `IS` *port* `:`
| `RESULT` *specifier* *name* `IS` *variable* `:` ‡ (3.3)

actual = *variable*
| *channel*
| *timer*
| *port*
| *expression*

allocation = `PLACE` *name* `AT` *expression* `:`
| `PLACE` *name* *expression* `:` ‡ (3.5.1)

alternation = `ALT`
  {*alternative*}
| `ALT` *replicator*
  alternative
| `PRI ALT`
  {*alternative*}
| `PRI ALT` *replicator*
  alternative

alternative = *guarded.alternative*
| *alternation*
| *channel* `?` `CASE`
  {*variant*}
| *boolean* `&` *channel* `?` `CASE`
  {*variant*}
| *extended.input* ‡ (3.7)
| *boolean* `&` *extended.input* ‡ (3.7)
| *specification*
alternative

array.constructor = `[` *replicator* `|` *expression* `]` ‡ (3.4)

assignment = *variable.list* `:=` *expression.list*

base	=	*expression*	
boolean	=	*expression*	
byte	=	`'`*character*`'`	
case.expression	=	*expression*	
case.input	=	*channel* `? CASE`	
		{*variant*}	
channel	=	*name*	
	|	*name channel.dir*	‡ (3.1)
	|	*channel* `[` *expression* `]`	
	|	*channel* `[` *expression* `]` *channel.dir*	‡ (3.1)
	|	`[` *channel* `FROM` *base* `FOR` *count* `]`	
	|	`[` *channel* `FROM` *base* `]`	
	|	`[` *channel* `FOR` *count* `]`	
channel.dir	=	`?`	‡ (3.1)
	|	`!`	
channel.mobendtype	=	*name channel.dir*	‡ (4.3)
	|	`SHARED` *name channel.dir*	
channel.type	=	`CHAN OF` *protocol*	
	|	`CHAN` *protocol*	‡ (3.5.1)
	|	`[` *expression* `]` *channel.type*	
choice	=	*guarded.choice*	
	|	*conditional*	
	|	*specification*	
		choice	
conditional	=	`IF`	
		{*choice*}	
	|	`IF` *replicator*	
		choice	
conversion	=	*data.type operand*	
	|	*data.type* `ROUND` *operand*	
	|	*data.type* `TRUNC` *operand*	
count	=	*expression*	
data.type	=	`BOOL`	
	|	`BYTE`	
	|	`INT`	
	|	`INT16`	
	|	`INT32`	
	|	`INT64`	

	|	REAL32	
	|	REAL64	
	|	*name*	
	|	[*expression*] *data.type*	
declaration	=	*data.type* $\{_1,$ *name*$\}$:	
	|	*channel.type* $\{_1,$ *name*$\}$:	
	|	*timer.type* $\{_1,$ *name*$\}$:	
	|	*port.type* $\{_1,$ *name*$\}$:	
	|	INITIAL $\{_1,$ *data.type*$\}$ *name* IS $\{_1,$ *expression*$\}$:	†
	|	MOBILE *data.type* $\{_1,$ *name*$\}$:	‡ (4.2)
	|	MOBILE [] *data.type* $\{_1,$ *name*$\}$:	‡ (4.2)
	|	*channel.mobendtype* $\{_1,$ *name*$\}$:	‡ (4.3)
	|	SHARED *channel.type* $\{_1,$ *name*$\}$:	‡ (4.4)
	|	PLACED *channel.type* *name* AT *expression* :	‡ (6.4)
	|	PLACED *data.type* *name* AT *expression* :	
definition	=	DATA TYPE *name* IS *data.type* :	
	|	DATA TYPE *name* *structured.type* :	
	|	CHAN TYPE *name* *structured.chantype* :	‡ (4.3)
	|	[REC|RECURSIVE] CHAN TYPE *name* *structured.chantype* :	‡ (4.4)
	|	PROTOCOL *name* IS *simple.protocol*	
	|	PROTOCOL *name* IS *sequential.protocol*	

	|	PROTOCOL *name*
		  CASE
		    $\{tagged.protocol\}$
		:	

	|	PROTOCOL *name* EXTENDS $\{_1,$ *name*$\}$
		  CASE
		    $\{tagged.protocol\}$
		:
			‡ (3.2.2)

	|	PROC *name* ($\{_0,$ *formal*$\}$) *process* :	
	|	[REC|RECURSIVE] PROC *name* ($\{_0,$ *formal*$\}$) *process* :	‡ (5.1)
	|	$\{_1,$ *data.type*$\}$ *function.header* *value.process* :	
	|	$\{_1,$ *data.type*$\}$ *function.header* IS *expression.list* :	
	|	*specifier* *name* RETYPES *variable* :	

```
                  |  VAL specifier name RETYPES expression :
                  |  specifier name RETYPES channel :
                  |  specifier name RETYPES port :
                  |  specifier name RESHAPES variable :
                  |  VAL specifier name RESHAPES expression :
                  |  specifier name RESHAPES channel :
                  |  specifier name RESHAPES port :

delayed.input     =  timer ? AFTER expression

dyadic.operator   =  + | - | * | / | \ | REM | PLUS | MINUS | TIMES
                  |  /\ | \/ | >< | BITAND | BITOR | AND | OR
                  |  = | <> | < | > | >= | <= | AFTER
                  |  @@ | $$ | % | %% | && | <% | %> | <& | &>              †
                  |  <] | [> | <@ | @> | @ | ++ | !! | == | ^              †

exponent          =  +digits
                  |  -digits

expression        =  operand
                  |  monadic.operator operand
                  |  operand dyadic.operator operand
                  |  MOSTPOS data.type
                  |  MOSTNEG data.type
                  |  SIZE data.type
                  |  conversion

expression.list   =  {1, expression}
                  |  name ( {0, expression} )
                  |  ( value.process
                     )

extended.input    =  channel ?? {1; input.item}                        ‡ (3.7)
                       process
                       process
                  |  channel ?? CASE tagged.list                       ‡ (3.7)
                       process
                       process
                  |  channel ?? CASE                                   ‡ (3.7)
                       {extended.variant}

extended.variant  =  tagged.list                                       ‡ (3.7)
                       process
                       process
                  |  specification
                     extended.variant

field.name        =  name
```

<table>
<tr><td>forking</td><td>=</td><td><code>FORKING</code></td><td>‡ (5.3)</td></tr>
<tr><td></td><td></td><td> process</td><td></td></tr>
<tr><td>formal</td><td>=</td><td>specifier $\{_1, \mathit{name}\}$</td><td></td></tr>
<tr><td></td><td>|</td><td><code>VAL</code> specifier $\{_1, \mathit{name}\}$</td><td></td></tr>
<tr><td></td><td>|</td><td><code>RESULT</code> specifier $\{_1, \mathit{name}\}$</td><td>‡ (3.3)</td></tr>
<tr><td>function.header</td><td>=</td><td><code>FUNCTION</code> name <code>(</code> $\{_0, \mathit{formal}\}$ <code>)</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>FUNCTION "</code>monadic.operator<code>" (</code> formal <code>)</code></td><td>†</td></tr>
<tr><td></td><td>|</td><td><code>FUNCTION "</code>dyadic.operator<code>" (</code> formal <code>,</code> formal <code>)</code></td><td>†</td></tr>
<tr><td>guard</td><td>=</td><td>input</td><td></td></tr>
<tr><td></td><td>|</td><td><code>SKIP</code></td><td>‡ (3.9.1)</td></tr>
<tr><td></td><td>|</td><td>boolean <code>&</code> input</td><td></td></tr>
<tr><td></td><td>|</td><td>boolean <code>&</code> <code>SKIP</code></td><td></td></tr>
<tr><td>guarded.alternative</td><td>=</td><td>guard</td><td></td></tr>
<tr><td></td><td></td><td> process</td><td></td></tr>
<tr><td>guarded.choice</td><td>=</td><td>boolean</td><td></td></tr>
<tr><td></td><td></td><td> process</td><td></td></tr>
<tr><td>input</td><td>=</td><td>channel <code>?</code> $\{_1;$ input.item$\}$</td><td></td></tr>
<tr><td></td><td>|</td><td>channel <code>? CASE</code> tagged.list</td><td></td></tr>
<tr><td></td><td>|</td><td>timer.input</td><td></td></tr>
<tr><td></td><td>|</td><td>delayed.input</td><td></td></tr>
<tr><td></td><td>|</td><td>port <code>?</code> variable</td><td></td></tr>
<tr><td>input.item</td><td>=</td><td>variable</td><td></td></tr>
<tr><td></td><td>|</td><td>variable <code>::</code> variable</td><td></td></tr>
<tr><td>integer</td><td>=</td><td>digits</td><td></td></tr>
<tr><td></td><td>|</td><td><code>#</code>hex.digits</td><td></td></tr>
<tr><td>literal</td><td>=</td><td>integer</td><td></td></tr>
<tr><td></td><td>|</td><td>byte</td><td></td></tr>
<tr><td></td><td>|</td><td>real</td><td></td></tr>
<tr><td></td><td>|</td><td>integer <code>(</code> data.type <code>)</code></td><td></td></tr>
<tr><td></td><td>|</td><td>byte <code>(</code> data.type <code>)</code></td><td></td></tr>
<tr><td></td><td>|</td><td>real <code>(</code> data.type <code>)</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>TRUE</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>FALSE</code></td><td></td></tr>
<tr><td>loop</td><td>=</td><td><code>WHILE</code> boolean</td><td></td></tr>
<tr><td></td><td></td><td> process</td><td></td></tr>
<tr><td>monadic.operator</td><td>=</td><td><code>-</code> | <code>MINUS</code> | <code>~</code> | <code>BITNOT</code> | <code>NOT</code> | <code>SIZE</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>@@</code> | <code>$$</code> | <code>%</code> | <code>%%</code> | <code>&&</code> | <code><%</code> | <code>%></code> | <code><&</code> | <code>&></code></td><td>†</td></tr>
<tr><td></td><td>|</td><td><code><]</code> | <code>[></code> | <code><@</code> | <code>@></code> | <code>@</code> | <code>++</code> | <code>!!</code> | <code>==</code> | <code>^</code></td><td>†</td></tr>
</table>

operand	=	*variable*	
	|	*literal*	
	|	*table*	
	|	*array.constructor*	‡ (3.4)
	|	`(` *expression* `)`	
	|	`(` *value.process* `)`	
	|	*name* `(` $\{_0$, *expression*$\}$ `)`	
	|	*operand* `[` *expression* $\{_]$, $\}$	
	|	`BYTESIN (` *operand* `)`	
	|	`BYTESIN (` *data.type* `)`	
	|	`OFFSETOF (` *name* `,` *field.name* `)`	
option	=	$\{_1$, *case.expression*$\}$ *process*	
	|	`ELSE` *process*	
	|	*specification* *option*	
output	=	*channel* `!` $\{_1$; *output.item*$\}$	
	|	*channel* `!` *tag*	
	|	*channel* `!` *tag* `;` $\{_1$; *output.item*$\}$	
	|	*port* `!` *expression*	
output.item	=	*expression*	
	|	*expression* `::` *expression*	
parallel	=	`PAR` $\{$*process*$\}$	
	|	`PAR` *replicator* *process*	‡ (5.2)
	|	`PRI PAR` $\{$*process*$\}$	
	|	`PRI PAR` *replicator* *process*	
	|	*placed.par*	$\Re$
placed.par	=	`PLACED PAR` $\{$*placed.par*$\}$	$\Re$
	|	`PLACED PAR` *replicator* *placed.par*	$\Re$
	|	`PROCESSOR` *expression* *process*	$\Re$
port	=	*name*	
	|	*port* `[` *expression* `]`	
	|	`[` *port* `FROM` *base* `FOR` *count* `]`	
	|	`[` *port* `FROM` *base* `]`	

	|	`[` *port* `FOR` *count* `]`	
port.type	=	`PORT OF` *data.type*	
	|	`PORT` *data.type*	‡ (3.5.1)
	|	`[` *expression* `]` *port.type*	
proc.instance	=	*name* `(` $\{_0$, *actual*$\}$ `)`	
process	=	*assignment*	
	|	*input*	
	|	*output*	
	|	`SKIP`	
	|	`STOP`	
	|	*sequence*	
	|	*conditional*	
	|	*selection*	
	|	*loop*	
	|	*parallel*	
	|	*forking*	‡ (3.7)
	|	*alternation*	
	|	*case.input*	
	|	*extended.input*	‡ (3.7)
	|	*proc.instance*	
	|	`FORK` *proc.instance*	‡ (5.3)
	|	*specification* *process*	
	|	*allocation* *process*	
protocol	=	*name*	
	|	*simple.protocol*	
real	=	*digits* `.` *digits*	
	|	*digits* `.` *digits* `E` *exponent*	
replicator	=	*name* `=` *base* `FOR` *count*	
	|	*name* `=` *base* `FOR` *count* `STEP` *stride*	‡ (3.6)
selection	=	`CASE` *selector* $\{$*option*$\}$	
selector	=	*expression*	
sequence	=	`SEQ` $\{$*process*$\}$	
	|	`SEQ` *replicator* *process*	
sequential.protocol	=	$\{_1$; *simple.protocol*$\}$	

simple.protocol	=	*data.type*	
	|	`ANY`	
	|	*data.type* `::[]` *data.type*	
specification	=	*declaration*	
	|	*abbreviation*	
	|	*definition*	
specifier	=	*data.type*	
	|	*channel.type*	
	|	*timer.type*	
	|	*port.type*	
	|	`[]` *specifier*	
	|	`[` *expression* `]` *specifier*	
stride	=	*expression*	
string	=	`"` *characters* `"`	
structured.type	=	`RECORD`	
		$\{$*data.type*$\{_1,$ *field.name*$\}$:$\}$	
	|	`PACKED RECORD`	
		$\{$*data.type*$\{_1,$ *field.name*$\}$:$\}$	
	|	`MOBILE RECORD`	‡ (4.2)
		$\{$*data.type*$\{_1,$ *field.name*$\}$:$\}$	
	|	`PACKED MOBILE RECORD`	‡ (4.2)
		$\{$*data.type*$\{_1,$ *field.name*$\}$:$\}$	
structured.chantype	=	`MOBILE RECORD`	‡ (4.3)
		$\{$*channel.type*$\{_1,$ *field.name*$\}$*channel.dir*:$\}$	
table	=	*string*	
	|	*string* `(` *name* `)`	
	|	`[` $\{_1,$ *expression*$\}$ `]`	
	|	*table* `[` *expression* `]`	
	|	`[` *table* `FROM` *base* `FOR` *count* `]`	
	|	`[` *table* `FROM` *base* `]`	
	|	`[` *table* `FOR` *count* `]`	
tag	=	*name*	
tagged.list	=	*tag*	
	|	*tag* `;` $\{_1;$ *input.item*$\}$	
tagged.protocol	=	*tag*	
	|	*tag* `;` *sequential.protocol*	
	|	`FROM` *name*	‡ (3.2.1)
timer.input	=	*timer* `?` *variable*	

timer	=	*name*
	|	*timer* [*expression*]
	|	[*timer* $\{_{FROM},\}$base FOR *count*]
	|	[*timer* $\{_{FROM},\}$base]
	|	[*timer* FOR *count*]
timer.type	=	TIMER
	|	[*expression*] *timer.type*

value.process	=	VALOF
		  process
		  RESULT *expression.list*

	|	*expression* *value.process*
variable	=	*name*
	|	*variable* [*expression*]
	|	[*variable* FROM *base* FOR *count*]
	|	[*variable* FROM *base*]
	|	[*variable* FOR *count*]
variable.list	=	$\{_1,$ *variable*$\}$
variant	=	*tagged.list* *process*
	|	*specification* *variant*

A.3 Pre-Processor Grammar

pp.booldop = `AND` | `OR`

pp.boolmop = `NOT`

pp.cmpop = `=` | `<>` | `<` | `<=` | `>` | `>=`

pp.define = `#DEFINE` *pp.name* {*pp.value*}
| `#DEFINE` *pp.name*

pp.directive = *pp.define* | *pp.undefine* | *pp.user* | *pp.if*

pp.if = `#IF` *pp.ifexp*
{`#RELAX`}
...
pp.iftail

pp.ifexp = `TRUE`
| `FALSE`
| `DEFINED` (*pp.name*)
| *pp.name* *pp.cmpop* *pp.name*
| *pp.name* *pp.cmpop* *pp.value*
| *pp.boolmop* *pp.ifexp*
| *pp.ifexp* *pp.booldop* *pp.ifexp*
| (*pp.ifexp*)

pp.iftail = `#ELIF` *pp.ifexp*
{`#RELAX`}
...
pp.iftail
| `#ELSE`
{`#RELAX`}
...
`#ENDIF`
| `#ENDIF`

pp.undefine = `#UNDEFINE` *pp.name*

pp.user = `#WARNING` *characters*
| `#ERROR` *characters*

pp.value = *digits*
| " *characters* "

Appendix B

Extended Transputer Code Additions

This appendix describes the various additions made to Michael Poole's Extended Transputer Code (ETC) [Poo98] in order to support the work described in this thesis.

The additions fall into two main categories – new *virtual-Transputer* instructions and new ETC-*specials*. ETC specials are used to describe non-code information (such as entry-points and `PROC` descriptor lines) and also code-generating operations which are likely to be target-architecture dependent (such as `FUNCTION` result passing).

The new virtual-Transputer instructions are listed in section B.1, with more detailed information on each instruction, grouped by function, in sections B.1.1 through B.1.5. Section B.2 describes the ETC-special additions, along with details of "magic" comments generated by the compiler to aid optimisation and cater for special target-dependant operations.

B.1 New Instructions

The first group of newly added instructions are to support dynamic-memory, listed in table B.1. Section B.1.1 provides more information on each of these instructions. Abstracting dynamic memory support in this way makes it relatively trivial to implement in other KRoC targets.

Name	**Description**
`MALLOC`	allocate a block of memory
`MRELEASE`	free a block of memory
`MNEW`	allocate memory from free-list
`MFREE`	free memory to free-list
`MRELEASEP`	free current process workspace

Table B.1: Virtual-Transputer instructions to support dynamic memory

The second group of added instructions are those relating to `MOBILE` communication. Table B.2 shows a list of these instructions, with more detail for each one in section B.1.2. `MOBILE`s are described in Chapter 4.

The third group of added instructions are used to implement the extended rendezvous (section 3.7). Table B.3 shows a list of these instructions, with more detail for each one in section B.1.3.

Name	Description
MIN	mobile input
MOUT	mobile output
MIN64	dynamic mobile array input
MOUT64	dynamic mobile array output
MINN	multi-dimensional dynamic mobile array input
MOUTN	multi-dimensional dynamic mobile array output

Table B.2: Virtual-Transputer instructions to support `MOBILE` communications

Name	Description
XABLE	extended input enable
XIN	extended input
XMIN	extended mobile input
XMIN64	extended dynamic mobile array input
XMINN	extended multi-dimensional dynamic mobile array input
XEND	extended input end

Table B.3: Virtual-Transputer instructions to support the extended rendezvous

The fourth group of instructions are used to handle external channel communications, for use with user-defined channels (section 6.4). They exist only for the sake of optimisation, since it is not always known (at compile time) whether a channel is external or not. Table B.4 lists these instructions, with more detail for each one given in section B.1.4.

Name	Description
EXTVRFY	verify external channel
EXTIN	external input
EXTOUT	external output
EXTENBC	enable external channel
EXTDISC	disable external channel
EXTMIN	external mobile input
EXTMOUT	external mobile output
EXTMIN64	external dynamic mobile array input
EXTMOUT64	external dynamic mobile array output
EXTMINN	external multi-dimensional dynamic mobile array input
EXTMOUTN	external multi-dimensional dynamic mobile array output

Table B.4: Virtual-Transputer instructions to support external communication

The final group instructions covers all the rest. This includes instructions to handle the various modified `ALT` enabling and disabling sequences (section 3.9), as well as compiler-generated external (user-defined) channel interfacing (section 6.4), priority handling (section 7.2) and debugging (section 7.3). Table B.5 shows a list of these instructions, with more detail for each one given in section B.1.5.

Each instruction is presented in a consistent manner, showing the name, op-code and any Transputer registers used. For example, an instruction which takes a single argument from the stack (and leaves the rest unchanged) would appear as:

Name	Description
NDISS	new disable skip-guard
NDISC	new disable channel-guard
NDIST	new disable timeout-guard
ENBS3	enable skip-guard with jump
ENBC3	enable channel-guard with jump
ENBT3	enable timer-guard with jump
TRAP	debug trap
GETPRI	retrieve process priority
SETPRI	change process priority

Table B.5: Virtual-Transputer instructions to support miscellaneous extensions

`Name Opcode`		(instruction type)
regs in	*areg*	single input
regs out	*areg'*	*breg*
	breg'	*creg*
	creg'	*undefined*

B.1.1 Dynamic Allocation

These instructions are used to access a dynamic memory capability. In KRoC/Linux, dynamic memory is managed using a Brinch-Hansen style memory allocator [Han95, Han96], with memory pools of increasing (approximately) half-powers of 2 – i.e. $\ldots, 32, 48, 64, 96, 128, 192, 256, \ldots$. The current implementation supports allocation of sizes in the range of 4 bytes to 1.5 giga-bytes (pools 0 to 57 inclusive). Allocations of non-zero sizes less than 4 are rounded up to 4 bytes.

MALLOC – Dynamic Memory Allocation

`MALLOC #E2`		(secondary instruction)
regs in	*areg*	desired size in bytes
regs out	*areg'*	pointer to allocated block
	breg'	*breg*
	creg'	*creg*

The `MALLOC` instruction provides a very general method for dynamic memory allocation, comparable to the standard C `malloc()` function. The number of bytes desired are placed in *Areg*, `MALLOC` is called and a pointer to the allocated block is left in *Areg'*. If the number of bytes requested is greater than the maximum supported size, a run-time error occurs.

The allocated block should be freed using the `MRELEASE` instruction. Allocations of zero are supported, but the pointer returned from such should never be dereferenced or passed to `MRELEASE`.

MRELEASE – Dynamic Memory Release

`MRELEASE #E3`		(secondary instruction)
regs in	*areg*	pointer to allocated block
regs out	*areg'*	*breg*
	breg'	*creg*
	creg'	*undefined*

`MRELEASE` is used to free dynamic memory allocated by '`MALLOC`'. Once released, the memory should never again be accessed. Explicitly setting pointers to '`NULL`' after releasing is recommended when debugging memory allocation errors. The run-time kernel may check the given pointer for validity and generate a run-time error for attempts to release `NULL` (or other invalid) pointers.

MNEW – Dynamic Allocation from Pool

`MNEW #E0`		(secondary instruction)
regs in	*areg*	pool number
regs out	*areg'*	pointer to allocated block
	breg'	*breg*
	creg'	*creg*

This instruction is used to allocate a block of memory from a given pool. As described at the start of this section, the memory pools are arranged in approximately half-powers of two. The following table shows the pool indexes and corresponding block sizes, for the first 20 pools:

Pool	**Bytes**	**Pool**	**Bytes**	**Pool**	**Bytes**	**Pool**	**Bytes**
0	4	5	24	10	128	15	768
1	6	6	32	11	192	16	1024
2	8	7	48	12	256	17	1536
3	12	8	64	13	384	18	2048
4	16	9	96	14	512	19	3072

MFREE – Dynamic Release to Pool

`MFREE #E1`		(secondary instruction)
regs in	*areg*	pool number
	breg	pointer to allocated block
regs out	*areg'*	*creg*
	breg'	*undefined*
	creg'	*undefined*

`MFREE` is used to free dynamic memory allocated by '`MNEW`'. Memory should be returned to the same pool from which it was allocated. Memory allocated using '`MALLOC`' should never be freed using '`MFREE`' — '`MRELEASE`' should be used instead.

MRELEASEP – Dynamic Process Release

`MRELEASEP #11`		(descheduling secondary instruction)
regs in	*areg*	workspace offset
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction is a combination of '`MRELEASE`' and '`STOPP`', and is used to allow a process whose workspace has been dynamically allocated to terminate and free its memory. For such processes, freeing the memory using '`MRELEASE`' then calling '`STOPP`' could potentially result in undefined behaviour (from race-hazards) — if the memory is re-used by another process between the '`MRELEASE`' and '`STOPP`' instructions: '`STOPP`' stores its return address in the process workspace. The '`MRELEASEP`' instruction does not store the return address — unnecessary since the process is finishing!

B.1.2 Mobile Communication

The instructions within this section are used to support mobile communication, for all types of mobile. '`MIN`' and '`MOUT`' are typically used for static mobiles (i.e. those allocated in *mobilespace*). The other mobile communication instructions, `MIN64`, `MOUT64`, `MINN` and `MOUTN` are used for dynamic mobiles.

MIN – Mobile Input

`MIN #E4`		(descheduling secondary instruction)
regs in	*areg*	channel address
	breg	destination address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction is used to implement the input-half of mobile communication. Communication, using this instruction, is done by pointer-swapping. Once reference is lost and another gained. The address placed in '*Breg*' is a pointer (to workspace) where the mobile variable (locally) resides. The value in this (workspace) location is swapped with the corresponding outputting process.

MOUT – Mobile Output

`MOUT #E5`		(descheduling secondary instruction)
regs in	*areg*	channel address
	breg	source address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction provides the output-half of mobile communication, using pointer-swapping. The address placed in '*Breg*' is a pointer (to workspace) where the mobile variable (locally) resides. The value in this (workspace) location is swapped with the corresponding inputting process. In a similar manner to other output instructions, `ALT`ing inputs cause the outputting process to block and the `ALT`er to be rescheduled.

MIN64 – Dynamic Mobile Array Input

`MIN64 #E6`		(descheduling secondary instruction)
regs in	*areg*	channel address
	breg	destination address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction implements the input-side of dynamic mobile communication, specifically for single-dimensional dynamic mobile arrays. The address passed points to the two words (64-bits) that will be over-written by the data sent by '`MOUT64`'.

MOUT64 – Dynamic Mobile Array Output

`MOUT64 #E7`		(descheduling secondary instruction)
regs in	*areg*	channel address
	breg	source address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction implements the output-side of dynamic mobile communication, specifically for single-dimension dynamic mobile arrays. The address passed points to the two words (64-bits) that hold the actual data address and the first dimension size.

The '`MIN64`'/'`MOUT64`' instructions implement a copy, not a pointer-swap (as '`MIN`' and '`MOUT`' do).

MINN – Multi-Dim Dynamic Mobile Array Input

`MINN #62`		(descheduling secondary instruction)
regs in	*areg*	word count
	breg	channel address
	creg	destination address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

The '`MINN`' instruction implements the input-side of dynamic mobile communication, for dynamic mobile arrays with more than one dimension. The value placed in '*areg*' is the number of dimensions plus one (for the pointer). The address given in '*creg*' points to the workspace location where the pointer and dimension counts will be stored.

MOUTN – Multi-Dim Dynamic Mobile Array Output

`MOUTN #64`		(descheduling secondary instruction)
regs in	*areg*	word count
	breg	channel address
	creg	source address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction implements the output-side of dynamic mobile communication, for dynamic mobile arrays with more than one dimension. The value placed in '*areg*' is the number of dimensions plus one (for the pointer). The address given in '*creg*' points to the workspace location where the pointer and dimension counts (for the dynamic mobile array) reside.

Like '`MIN64`' and '`MOUT64`', these instructions use a copy, rather than a swap.

B.1.3 Extended Inputs

The extended rendezvous (section 3.7) uses three new instructions for basic functionality: '`XABLE`', '`XIN`' and '`XEND`'; plus another three to support extended-inputs on `MOBILE` protocols: '`XMIN`', '`XMIN64`' and '`XMINN`'.

XABLE – Extended Input Enable

`XABLE #E8`		(descheduling secondary instruction)
regs in	*areg*	channel address
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

The extended enable instruction is used at the start of an extended input sequence to *wait* for the corresponding outputting process. If a process is already blocked in the channel this instruction returns immediately.

XIN – Extended Input

`XIN #E9`		(secondary instruction)
regs in	*areg*	count
	breg	channel address
	creg	destination address
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction performs a standard input (copy) from the process waiting in the channel pointed to by '*breg*'. The outputting process is assumed to be waiting (from an earlier '`XABLE`'). After the copy, the outputting process is *not* rescheduled.

XMIN – Extended Mobile Input

`XMIN #EA`		(secondary instruction)
regs in	*areg*	channel address
	breg	destination address (in mobilespace)
regs out	*areg'*	*creg*
	breg'	*undefined*
	creg'	*undefined*

This instruction performs a standard mobile input (pointer swap) from the process waiting in the channel pointed to by '*areg*'. The outputting process is assumed to be waiting (from an earlier '`XABLE`'). After the pointer-swap, the outputting process is not rescheduled.

XMIN64 – Extended Dynamic Mobile Array Input

`XMIN64 #EB`		(secondary instruction)
regs in	*areg*	channel address
	breg	destination address (in workspace)
regs out	*areg'*	*creg*
	breg'	*undefined*
	creg'	*undefined*

The '`XMIN64`' instruction performs a dynamic mobile array input (pointer and dimension copy), for single-dimensional dynamic arrays, from the process waiting in the channel pointed to by '`breg`'. The outputting process is assumed to be waiting (from an earlier '`XABLE`'). After the pointer and dimension copy, the outputting process is not rescheduled.

XMINN – Extended Multi-Dim Dynamic Mobile Array Input

`XMINN #65`		(secondary instruction)
regs in	*areg*	word count
	breg	channel address
	creg	destination address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction performs a multi-dimensional dynamic mobile array input (pointer and dimensions copy). The outputting process is assumed to be already blocked in the channel pointed to by '*breg*' (from an earlier '`XABLE`'). After the copy, the outputting process is not rescheduled. The value in '*areg*' gives the number of words to copy, which should be one more than the number of dimensions.

XEND – Extended Input End

`XEND #EC`		(secondary instruction)
regs in	*areg*	channel address
regs out	*areg'*	*breg*
	breg'	*creg*
	creg'	*undefined*

This instruction is executed at the end of an extended input to reschedule the blocked outputting process. After the blocked process has been added to the run-queue, the channel pointed to by '*areg*' is cleared (to NotProcess.p).

B.1.4 External Communication

The external communication instructions are used for supporting user-defined channels, described in section 6.4. In total there are 11 instructions — 8 for communication (arranged in pairs), 2 for `ALT` enable/disable and one extra for providing 'verification' of external channels.

The compiler only generates these instructions when it *knows* that a channel is `PLACED` (the mechanism used for accessing user-defined channels). External channels are flagged by having their low-address bit set to 1, that the translation of other communication instructions must check for (when user-defined channels are enabled). Code that does not use user-defined channels need not generate such checks (the intention is that user-defined channels are kept as local as possible).

EXTVRFY – External Channel Verify

`EXTVRFY #14`		(secondary instruction)
regs in	*areg*	hashcode
	breg	channel address
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction is used to verify an external channel at the point of its `PLACE`ment in occam (that may be a '`PLACED`' channel declaration). The channel address is passed '*breg*' with the protocol-hash in '*areg*'. The underlying user-defined channel implementation ensures that both sides of the channel have the same hash-code. If the hash-codes are different, a run-time error occurs.

EXTIN – External Channel Input

`EXTIN #60`		(secondary instruction)
regs in	*areg*	count
	breg	channel address
	creg	destination address
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction performs input from an external channel, whose address is given in '*breg*'. The size of the communication is given in '*areg*', with the data placed at the address given in '*creg*'.

EXTOUT – External Channel Output

`EXTOUT #61`		(secondary instruction)
regs in	*areg*	count
	breg	channel address
	creg	source address
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction performs output to an external channel, whose address is given in '*breg*'. The data to be sent is pointed to by '*creg*', with the size (in bytes) given in '*areg*'.

EXTENBC – External Channel Alternative Enable

`EXTENBC #66`		(secondary instruction)
regs in	*areg*	guard (pre-condition)
	breg	channel address
regs out	*areg'*	guard (pre-condition)
	breg'	*creg*
	creg'	*undefined*

This instruction performs the `ALT` enabling sequence for an external channel, whose address is given in '*breg*'. Like the '`ENBC`' instruction, if the guard given in '*areg*' is zero (false), this instruction does nothing, leaving zero in '*areg*'. Otherwise the external channel is enabled and the non-zero guard is returned in '*areg*'.

EXTNDISC – External Channel New Alternative Disable

`EXTNDISC #67`		(secondary instruction)
regs in	*areg*	process address
	breg	guard (pre-condition)
	creg	channel address
regs out	*areg'*	external channel ready
	breg'	*undefined*
	creg'	*undefined*

This instruction performs the `ALT` disabling sequence for an external channel, whose address is given in '*creg*'. Like the '`DISC`' instruction, if the guard given in '*breg*' is zero (false), this instruction does nothing, leaving zero in '*areg*' (channel not ready). Otherwise the external channel is disabled. The value returned in '*areg*' indicates the ready-ness of the external channel — zero if the channel was not ready.

Unlike the '`DISC`' instruction, however, '`EXTNDISC`' always sets the `ALT`ing process's 'Temp' slot (to the address given in '*areg*'), if the channel is ready. The '`DISC`' instruction only sets the 'Temp' slot if the channel was ready *and* the 'Temp' slot was previously NotProcess.p.

EXTMIN – External Channel Mobile Input

`EXTMIN #68`		(secondary instruction)
regs in	*areg*	channel address
	breg	destination address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction performs mobile input from an external channel, whose address is given in '*areg*'. The address given in '*breg*' holds the workspace address of the mobile's shadow-slot. The shadow-slot contains the pointer (into mobilespace) of where the data should be stored.

The size of the communication is not given explicitly in this instruction. If the user-defined code called requires the size of the underlying `MOBILE` type, the '`-msf`' compiler command-line flag should be used. This causes the compiler to allocate an extra word adjacent (and above) the workspace shadow-slot that is initialised to the type size when the mobile variable comes into scope (when its shadow-slot is initialised).

EXTMOUT – External Channel Mobile Output

`EXTMOUT #69`		(secondary instruction)
regs in	*areg*	channel address
	breg	source address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction performs mobile output to an external channel, whose address is given in '*areg*'. The address given in '*breg*' holds the workspace address of the mobile's shadow-slot. This contains the pointer (into mobilespace) of the data to be output.

The size of the communication is not given explicitly in this instruction. If the user-defined code called requires the size of the underlying `MOBILE` type, the '`-msf`' compiler command-line flag should be used. This causes the compiler to allocate an extra word adjacent (and above) the workspace shadow-slot that is initialised to the type size when the mobile variable comes into scope (when its shadow-slot is initialised).

EXTMIN64 – External Channel Dynamic Mobile Input

`EXTMIN64 #6A`		(secondary instruction)
regs in	*areg*	channel address
	breg	destination address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

The '`EXTMIN64`' instruction performs a single-dimensional dynamic mobile array input from an external channel. The external channel address is given in '*areg*'. The address in '*breg*' points to the workspace slots where the dynamic pointer and dimension count are allocated. When this instruction is called, the dynamic mobile pointer and size are *undefined*. The user-defined code called must set these fields appropriately. Dynamic memory blocks can be allocated using the '`dmem_alloc()`' function in the run-time kernel, or through the new '`MALLOC`' instruction (section B.1.1).

The base-type size is not given explicitly in this instruction. If the user-defined code called requires the size of the array elements, the '`-msf`' compiler command-line flag should be used. This causes the compiler to allocate an extra word beyond the dimension count, initialised to the base-type size when the mobile variable enters scope.

EXTMOUT64 – External Channel Dynamic Mobile Output

`EXTMOUT64 #6B`		(secondary instruction)
regs in	*areg*	channel address
	breg	source address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

The '`EXTMOUT64`' instruction performs a single-dimensional dynamic mobile array output to an external channel, whose address is given in '*areg*'. The address in '*breg*' points to the workspace slots where the dynamic pointer and dimension count are allocated. When this instruction returns, the pointer and count fields of the mobile array output are assumed to be *undefined*. Dynamic memory blocks can be released using the '`dmem_release()`' function in the run-time kernel, or through the new '`MRELEASE`' instruction (section B.1.1).

The base-type size is not given explicitly in this instruction. If the user-defined code called requires the size of the array elements, the '`-msf`' compiler command-line flag should be used. This causes the compiler to allocate an extra word beyond the dimension count, initialised to the base-type size when the mobile variable enters scope.

EXTMINN – External Channel Multi-Dim Dynamic Mobile Input

`EXTMINN #6E`		(secondary instruction)
regs in	*areg*	word count
	breg	channel address
	creg	destination address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction performs multi-dimensional dynamic mobile array input, from an external channel whose address is given in '*breg*'. The address given in '*creg*' points to the workspace location where the dynamic pointer and dimension sizes will be stored. '*areg*' should be set to the number of dimensions plus one (for the pointer).

When this instruction is called, the dynamic mobile pointer and dimension sizes (in workspace) are *undefined*. The user-defined code called must set these fields appropriately. Dynamic memory blocks can be allocated using the '`dmem_alloc()`' function in the run-time kernel, or through the new '`MALLOC`' instruction (section B.1.1).

The base-type size is not given explicitly in this instruction. If the user-defined code called requires the size of the base-type, the '`-msf`' compiler command-line flag should be used. This causes the compiler to allocate an extra word beyond the dimension sizes, initialised to the base-type size when the mobile variable enters scope.

EXTMOUTN – External Channel Multi-Dim Dynamic Mobile Output

`EXTMOUTN #6F`		(secondary instruction)
regs in	*areg*	word count
	breg	channel address
	creg	source address (in workspace)
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

This instruction performs multi-dimensional dynamic mobile array output, from an external channel whose address is given in '*breg*'. The address given in '*creg*' points to the workspace location where the dynamic mobile pointer and dimension sizes are stored. '*areg*' should be set to the number of dimensions plus one (for the pointer).

When this instruction returns, the compiler assumes that the pointer and dimension sizes are *undefined*. Dynamic memory blocks can be released using the '`dmem_release()`' function in the run-time kernel, or through the new '`MRELEASE`' instruction (section B.1.1).

The base-type size is not given explicitly in this instruction. If the user-defined code called requires the size of the base-type, the '`-msf`' compiler command-line flag should be used. This causes the compiler to allocate an extra word beyond the dimension sizes, initialised to the base-type size when the mobile variable enters scope.

B.1.5 Miscellany

The remaining instructions provide support for the reversed `ALT` disabling sequence (section 3.9.2), the enhanced `ALT` enabling sequence (section 3.9.3), process priority (section 7.2), and a debugging '`TRAP`'.

NDISC – New Channel Alternative Disable

`NDISC #ED`		(secondary instruction)
regs in	*areg*	process address
	breg	guard (pre-condition)
	creg	channel address
regs out	*areg'*	channel ready
	breg'	*undefined*
	creg'	*undefined*

This instruction performs an alternative disable on the channel pointed to by '*creg*', with the evaluated pre-condition in '*breg*' and the guarded process address in '*areg*'. If the pre-condition is zero (false), this instruction does nothing and returns with zero in '*areg*' (channel not ready). Otherwise the channel is disabled, if it was not already. If the channel is found to be ready, the process-address is placed in the invoking process's 'Temp' slot and the instruction returns with '*areg*' non-zero (true). Otherwise the 'Temp' slot is left unchanged and the instruction returns with zero (false) in '*areg*'.

NDIST – New Timeout Alternative Disable

`NDIST #EE`		(secondary instruction)
regs in	*areg*	process address
	breg	guard (pre-condition)
	creg	timeout
regs out	*areg'*	timed out
	breg'	*undefined*
	creg'	*undefined*

This instruction performs an alternative disable on the timeout in '*creg*', with the evaluated pre-condition in '*breg*' and the guarded process address in '*areg*'. If the pre-condition is zero (false), this instruction does nothing and returns with zero in '*areg*' (not timed out). Otherwise, the 'TLink' field of the `ALT`ing (invoking) process is checked. If 'TimeNotSet.p, no timeout occurred and zero (false) is returned in '*areg*'.

If the 'TLink' field is 'TimeSet.p', then the timeout expired (on the timer-queue). The the timeout passed in '*creg*' is *before* the time in the `ALT`ing process's 'Time' field, the address given in '*areg*' is placed in the `ALT`ing process's 'Temp' field, and the instruction returns with non-zero (true) in '*areg*'. Otherwise, this timeout is after the one which expired, and zero (false) is returned in '*areg*'.

If the 'TLink' field is anything other than these constants, then the `ALT`ing process is still on the timer-queue ('TPtr'). The timer-queue is scanned and the `ALT`ing process removed, before setting the process's '`TLink`' field to 'TimeNotSet.p' and returning with zero (false) in '*areg*'.

NDISS – New Skip Alternative Disable

`NDISS #EF`		(secondary instruction)
regs in	*areg*	process address
	breg	guard (pre-condition)
regs out	*areg'*	guard (pre-condition)
	breg'	*undefined*
	creg'	*undefined*

This instruction performs an alternative disable on a `SKIP` guard, with the evaluated pre-condition in '*breg*' and the guarded process address in '*areg*'. If the pre-condition is zero (false), this instruction does nothing and returns with zero left in '*areg*'. Otherwise the address passed in '*areg*' is placed in the `ALT`ing process's 'Temp' field and the instruction returns with non-zero (true) in '*areg*'.

ENBC3 – Enhanced Channel Alternative Enable

`ENBC3 #70`		(secondary instruction)
regs in	*areg*	jump address
	breg	guard (pre-condition)
	creg	channel address
regs out	*areg'*	guard (pre-condition)
	breg'	*undefined*
	creg'	*undefined*

The '`ENBC3`' instruction performs an 'enhanced' alternative enable on the channel pointed to by '*creg*', with the evaluated pre-condition in '*breg*' and the 'jump' address in '*areg*'. If the pre-condition is zero (false), this instruction does nothing and returns with zero in '*areg*'.

If the pre-condition is non-zero (true), and the channel is ready, execution continues at the address given in '*areg*'. The address given is typically that of a label placed directly before the corresponding disabling instruction, causing the guarded to be immediately selected. If the channel is not ready, it is enabled. In both cases, the (non-zero) pre-condition is returned in '*areg*'.

ENBT3 – Enhanced Timeout Alternative Enable

`ENBT3 #99`		(secondary instruction)
regs in	*areg*	jump address
	breg	guard (pre-condition)
	creg	timeout
regs out	*areg'*	guard (pre-condition)
	breg'	*undefined*
	creg'	*undefined*

The '`ENBT3`' instruction performs an 'enhanced' alternative enable on the timeout given in '*creg*', with the evaluated pre-condition in '*breg*' and the 'jump' address in '*areg*'. If the pre-condition is zero (false), this instruction does nothing and returns with zero in '*areg*'.

If the pre-condition is non-zero (true), then the `ALT`ing process's 'TLink' field is checked. If 'TimeNotSet.p' (no timeout set), the 'TLink' field is set to 'TimeSet.p' and the 'Time' field is set to the timeout given in '*creg*'. Otherwise, if the 'TLink' field was already set to 'TimeSet.p', the timeout in the 'Time' field is set to the timeout given in '*creg*', but only if the time in '*creg*' was not after the time already set.

The time stored in the `ALT`ing process's 'Time' field is then checked to see if the timeout has expired (by comparing it with the *current* time). If so, the `ALT`ing process's 'State' field is set to 'Ready.p', its 'Time' field is set to the current time, and the instruction returns to the given jump-address (from '*areg*'), leaving the (non-zero) guard/pre-condition in '*areg*'. The address given is typically that of a label placed directly before the corresponding disabling instruction, causing the guarded process to be immediately selected. If the timeout has not expired, the instruction returns normally, with the original guard/pre-condition in '*areg*'.

ENBS3 – Enhanced Skip Alternative Enable

`ENBS3 #AD`		(secondary instruction)
regs in	*areg*	jump address
	breg	guard (pre-condition)
regs out	*areg'*	guard (pre-condition)
	breg'	*creg*
	creg'	*undefined*

The '`ENBS3`' instruction performs an 'enhanced' alternative enable for a `SKIP` guard, with the evaluated pre-condition in '*breg*' and the 'jump' address in '*areg*'. If the pre-condition is zero (false), this instruction does nothing and returns with zero in '*areg*'.

If the pre-condition is non-zero (true), then the instruction returns to the given jump-address (from '*areg*'), leaving the (non-zero) guard/pre-condition in '*areg*'. The address given is typically that of a label placed directly before the corresponding disabling instruction, causing the guarded process to be immediately selected.

TRAP – Debug Trap

`TRAP #FC`		(secondary instruction)
regs in	*areg*	trapval-A
	breg	trapval-B
	creg	trapval-C
regs out	*areg'*	*areg*
	breg'	*breg*
	creg'	*creg*

This instruction provides a convenient way of debugging inline Transputer assembly. The contents of '*areg*', '*breg*' and '*creg*' are left un-changed, and need not all be defined. The current run-time kernel displays a dump of the Transputer stack and other scheduler variables when this instruction is encountered.

GETPRI – Get Process Priority

`GETPRI #`		(secondary instruction)
regs in	*no inputs*	
regs out	*areg'*	current priority
	breg'	*areg*
	creg'	*breg*

The '`GETPRI`' instruction retrieves the current process priority and places it in '*areg*', pushing the rest of the stack down. The CCSP run-time kernel keeps this information in a global '`PPriority`' variable, which this instruction simply reads.

SETPRI – Set Process Priority

`SETPRI #`		(secondary instruction)
regs in	*areg*	new priority
regs out	*areg'*	*undefined*
	breg'	*undefined*
	creg'	*undefined*

The '`SETPRI`' instruction changes the current process priority to the priority-level specified in '*areg*'. If the priority given is out of range, it is silently adjusted to the minimum or maximum priority-level appropriately.

The the change in priority results in another process (of higher priority) becoming runnable, the invoking process is placed on the appropriate priority run-queue and the higher priority process scheduled.

B.2 New ETC Specials

The additions to the ETC specials are to support *mobilespace* and the new `LOOPEND` instructions. For mobilespace, specials are used to describe the layout of (non-dynamic) `MOBILE`s in blocks of mobilespace. The new `LOOPEND` instructions, for handling `STEP` in replicators (section 3.6), are coded as ETC specials since the exiting `LOOPEND` instruction is an ETC special — in the interests of sensible label handling[1]. For optimising replicators that do not use their replicator *name* (described in section 3.6.2), and for handling other target-dependant features (such as IO access for `PORT`s), special comments are inserted by the compiler. These are described in section B.2.3.

The additions that support (primarily) semaphore operations for shared mobile channel-ends are covered in section B.2.4, along with any other added ETC specials in this group (including the new rescheduling instruction).

B.2.1 Mobilespace Initialisation Specials

Two ETC specials are used to encode information about the mobilespace required by a `PROC`. The first gives the mobilespace size, in words, and is introduced by "`6D F4`", followed by "`LDC` *n*". This will be zero if a `PROC` does not require mobilespace.

The other provides the mobilespace map, used to generate code that initialises mobilespace at run-time. The map is introduced by "`6D F5`", followed by "`LDC` *mspoff*", "`LDC` *count*", and the map itself, '*count*' pairs of "`LDC` *sl-offset*, `LDC` *dt-offset*".

'`mspoffs`' gives the workspace offset of the hidden mobilespace parameter ("`$MSP`"), which points to the start of a `PROC`s mobilespace. The initialisation should only be performed if the first word of mobilespace is '`MINT`' (`0x80000000`).

Each '*sl-offset, dt-offset*' pair (providing *word* offsets), is processed according to the following rules: If (*sl-offset*<0), then '`MINT`' is stored in the mobilespace at '*dt-offset*'. Else, if (*dt-offset*>0) (and (*sl-offset*$\geqslant 0$)), then the address of the mobilespace at '*dt-offset*' is stored in the mobilespace at '*sl-offset*'. Else, if ($dt - offset < 0$), then 'NotProcess.p' is stored in the mobilespace at '*sl-offset*'.

[1] All instructions involving labels are generated as ETC specials, allowing code-generation optimisations for targets which can handle addresses directly in jump instructions.

Otherwise (when (*dt-offset*$= 0$) and (*sl-offset*$\geqslant 0$)), '`MINT`' is stored in the mobilespace at '*sl-offset*' and zero is stored at '*sl-offset* $+1$'.

B.2.2 New `LOOPEND` Specials

The existing loop-end special (named '`LOOPEND`') is introduced by "`6D F1`", followed by three "`LDLP`" instructions. The first gives the workspace offset (relative to the loop-end) where the loop-count and replicator value are stored. The other two '`LDLP`' instructions provide the numeric labels of the loop-start and loop-end.

The operation of the loop-end instruction is to *decrement* the loop-count, jumping to the loop-end label if zero. Otherwise, the replicator value is *incremented* and a jump is made back to the loop-start.

Two new loop-end specials provide support for '`STEP`' in replicators (section 3.6). The first of these, '`LOOPEND3`' provides support for arbitrary strides. Instead of incrementing the replicator value by one, the replicator value is incremented by the word stored above it, which the compiler initialises (to the '`STEP`' value) before the loop starts. This is introduced using "`6D F2`", followed by the same three instructions (ws-offset, start-label and end-label).

The second new loop-end special, '`LOOPENDR`' is similar to the basic '`LOOPEND`' special, but decrements the replicator value instead of incrementing it. This is used to support the specific case where the loop stride is -1. The special is introduced using "`6D F3`", followed by the ws-offset, start-label and end-label instructions.

B.2.3 Magic Compiler Comments

In addition to code and data, ETC can include special 'compiler comments'. Traditionally, these were used to provide some additional information in the compiler output (Transputer assembler), mostly for human consumption. The Sparc version of KRoC [WW96] (that generates textual compiler output), uses compiler comments to introduce various non-code information, file-names and line-numbers for example. With ETC [Poo98] compiler output, as used by KRoC/Linux, much of the non-code information is provided through ETC specials, and 'compiler comment' is one of these. Compiler comments can also be explicitly generated by the programmer, using the '`#COMMENT`' directive.

The current occam compiler in KRoC/Linux generates three 'special' compiler comments, that affect the code-generation of the following instruction.

The first of these, "`.MAGIC IOSPACE`", indicates that the following instruction (either a load or a store) should be turned into an IO-port read or write — for the i386 architecture at least. From occam, this is accessed in one of two ways: using a '`PLACED PORT`' type; or through a '`PLACED`' array/variable declaration immediately followed by the '`#PRAGMA IOSPACE`' directive.

The second, "`.MAGIC PREENABLE`", is used in conjunction with the enhanced `ALT` enabling instructions to provide `ALT` pre-enabling (described in section 3.9.3). Instead of translating the following instruction directly (that will be one of '`ENBC3`', '`ENBT3`' or '`ENBS3`'), the translator generates a 'test' for the `ALT` condition, and immediately jumps if the guard is ready (to the address given in '*areg*'). If the pre-condition is zero (false), or the guard is not ready, nothing is done — i.e. no enabling takes place.

If timeout guards are present, the compiler places code before the pre-enabling sequence that loads the current time into the `ALT`ing process's 'Time' field. This is subsequently used when pre-enabling '`ENBT3`', that compares the timeout given (in '*creg*') with the current time, and jumps if the timeout has expired.

The final magic compiler comment is "`.MAGIC UNUSED LOOPVAR`". This is inserted by the compiler immediately before a loop-end ETC special to indicate that the replicator variable is un-used. This allows for a slight optimisation in the code generated for loops that do not use their replicator, that can make a significant difference in very tight loops.

B.2.4 Semaphores, Rescheduling and Others

The remaining ETC specials are introduced using "`6F F0`". This is already used to provide support for a number of special instructions, for example, '`BOOLINVERT`' and '`NOTPROCESS`'.

Five new special instructions have been added here: '`SEMINIT`', '`SEMCLAIM`' and '`SEMRELEASE`' for semaphore operations; '`RESCHEDULE`' for performing a reschedule; and '`CHECKNOTNULL`' that checks for a valid (non-null) pointer.

The three semaphore operations are used by '`SHARED`' mobile channel-ends, covered in section 4.3. The '`SEMINIT`' instruction initialises the semaphore structure pointed to by '*areg*', currently two words in length. The '`SEMCLAIM`' instruction claims the semaphore pointed to by '*areg*'. If the semaphore is already held, the current process is placed on the semaphore queue and descheduled. After a claim, a semaphore is released using '`SEMRELEASE`', with the pointer to the semaphore in '*areg*'. If releasing the semaphore causes a higher-priority process (section 7.2) to become runnable, the current process should be descheduled, and the higher-priority process rescheduled.

The new '`RESCHEDULE`' instruction is used to perform a simple reschedule — placing the current process on the back of the run-queue, and running a process taken off the front of the run-queue. The usual method of generating a reschedule is with the instruction sequence "`LDLP` 0, `RUNP`, `STOPP`". This, however, breaks when process priority is enabled — '`RUNP`' is expecting an already descheduled process as its argument, but the current process is active. In particular, the current process's 'Priority' field (that `RUNP` uses to determine which run-queue the process should be placed on), is undefined — it is only set by '`STOPP`'.

The alternative reschedule code uses '`STARTP`', and inherits the current process's priority. This results in more code being generated, however, and the resulting translation into native code is unlikely to be efficient (considering the simplicity of the basic reschedule operation). Rather than matching reschedule instruction sequences explicitly, the compiler now generates the special '`RESCHEDULE`' instruction. The translator can then produce a highly optimised (inlined) reschedule, particularly when the current run-queue is held in processor registers.

The final new instruction in this group is '`CHECKNOTNULL`'. This performs a run-time check to ensure that the value in '*areg*' is non-null ($\neq$ NotProcess.p). The compiler uses this to check the validity of dynamic mobile channel-end references, before attempting to access any of the components within, and only when the state of the mobile channel-end being accessed is unknown.