Technical Report

UCAM-CL-TR-6
ISSN 1476-2986

Number 6

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

System programming in a high level language

Andrew D. Birrell

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 1977 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Acknowledgements

The work described in this dissertation has been supported by the Science Research Council. It would not have been possible without the support and encouragement of Brenda, and of my supervisor Roger Needham, and of my head of department, Professor M.V.Wilkes. In particular, Brenda showed particular diligence in typing this document, and perseverence in encouraging me to write it.

The paper reproduced as Appendix Y was first published as an invited paper at the 1977 conference on Algol68 at the University of Strathclyde [40].

Contents

Section A: Introduction

1. The Problem

1.1 Aims and Requirements

This thesis is concerned with the construction of a high level language system suitable for the implementation of a general purpose operating system for a computer. There are three aspects to this task: firstly, a suitable high level language must be chosen or designed; secondly, a suitable implementation of this language must be manufactured; thirdly, the operating system itself must be written. These three aspects inevitably overlap in time - experience in implementing the language may cause one to review decisions taken in the design of the language, and experience in constructing the operating system will bring to light inadequacies, inconveniences and inelegancies in both the implementation and the language.

Most previous work in this field has been concerned with the first of these aspects, and has adopted the approach of designing special-purpose languages, categorized as 'System Programming Languages' (SPL's) or 'Machine Oriented Languages' (MOL's). Various such languages have been developed, some of which are discussed below. Few such languages have achieved the elegance or generality of ordinary general-purpose languages such as Pascal or Algol68. Little or no investigation has previously been made into the second of these aspects, the implementation of the language. The implementation, as distinct from the language, can have a very considerable effect on the practicability of using the resulting language system for manufacturing an operating system. Certainly, there are languages which, however brilliant the implementation, would inevitably be disastrous for writing an operating system; but the implementation, however suitable the language, makes the difference between the language system being an aid or an impediment to the system programmer. It is with aspects of the implementation that this thesis is mainly concerned.

It should be emphasised that we are considering the real construction of an operating system on physical hardware without sophisticated external support. The 'language system' must not amount to a simulation package, nor include facilities which would normally be considered part of the operating system (such as in Simula or Concurrent Pascal), unless those facilities are themselves implemented using a high level language (they could then be considered to be part of the operating system). It is a principle of the design that the language system should not contain imbedded design decisions which would be more properly considered as part of the design of the operating system - it

should be a tool, not an integral part of the operating system. Also, since we are providing a general purpose operating system, user programs can be written in any language - we are not assuming a single-language operating system.

When embarking on the production of a suitable language system, we must at an early stage make a fundamental decision: whether to base the work on an existing high level language which can be modified as necessary, or whether to construct a new language from scratch. If we adopt the latter alternative, we will inevitably design a special-purpose language and follow in the footsteps of BLISS et al. In fact, I have adopted the former alternative. This leads to new ground: we must decide to what extent modifications to the language are needed, and this should enable us to decide which factors in the design of a suitable language arise peculiarly from writing an operating system, and which arise from the normal requirements of a non-numerical program. Also, when following this path we can investigate the extent to which suitable implementation techniques can aid us in achieving our aims with minimal special purpose modifications to the language.

The language system produced was intended to be used (and is used) for an operating system for the Cambridge CAP computer. This computer, and its operating system, have many unusual features, but the techniques and facilities developed for the language system are not especially CAP-oriented - they have applicability to many other machines and environments.

1.2 Use of Normal High Level Language Facilities

The advantages of using a high-level language instead of machine-code are numerous and well known; they amount to saying that, when writing in a high-level language, it is easier to write programs, more difficult to write faulty ones, and when you have written a faulty program you discover the fault sooner. We are not much concerned here with faulty programs (although much of the distinction between a 'good' and a 'bad' implementation of a language lies in what it will do with a faulty program), but it must always be borne in mind when considering any topic in language design or implementation that a programmer must be told as early as possible of any mistake. As far as possible, all error checking should be performed at compile time, and we must always try to have sufficient redundancy in constructs for a simple mistake to be detectable. Error detection is one of the major gains in using a high level language, but first we must be able to express our problem conveniently.

Much system programming is amenable to writing in a normal high level language with no particular trouble. Such operations as converting a textual file title into a disc address are merely

mathematical mappings (albeit somewhat complicated ones), and can be implemented using techniques remarkably similar to those used in any non-numerical computing. A programmer implementing such algorithms is involved in the task of taking a single, complicated operation and breaking it into several simpler ones. When writing in machine code, he is forced to resolve the problem into particular bit or word manipulations allowed by his particular hardware and operating environment; when writing in a high level language, such drastic measures are not forced upon him, since he has available to him operations which are less basic. A high level language makes available to the programmer sets of abstract objects, which he can consider not in terms of his hardware and operating system, but purely in terms of the objects, their relationships to each other, and the operations he can perform upon them. For almost all of almost every system program, it is sufficient to express the algorithm in such abstract terms - it is extremely rare for the programmer to require to manipulate non-abstract (hardware or system oriented) objects or operations. Expressing algorithms in such abstract terms clearly has many advantages. If the abstract objects and operations are suitable, it will be much easier to convert an algorithm into them than into the machine objects, purely because they involve 'higher level' concepts more closely related to the original algorithm. For example, indexing an array is more closely related to table look-up than is adding a fixed point integer, multiplied by the amount of store-per-element, to the base address of a sequence of elements; indeed, the programmer can use the abstract facility without knowing how it is implemented today. Thus, a suitable set of abstractions will be ones which would be encountered while converting the abstract algorithm into machine code - using the abstractions saves a step in the conversion. If the algorithm can be expressed purely in terms of the abstractions, then it has no dependency on the hardware or operating environment. This has several beneficial consequences: firstly, it implies that the programmer has not made a mistake in mapping his objects onto the hardware (all such mistakes are centralised in the compiler!); secondly, the hardware can change without affecting his program (for a system program, the gain here is that the program can be developed on a different computer, or before the hardware of the target computer has stabilised); thirdly, the operating system interfaces can change (this is quite likely to happen during the development of a system, and having consequently to rewrite all the programs, rather than just recompile them, would be unfortunate). Also, abstract objects and operations are likely to have (and, indeed, are usually designed to have) consistency constraints which can be checked at compile time; this can produce very powerful error checking, and is a feature of most good high level languages.

Even programs which are not concerned exclusively with abstract objects or operations (for example, a program handling file directories or disc addresses) are likely to be concerned mainly with abstract operations (such as sorting the entries in the directory or yielding the disc addresses from a table look-

up). There are very few programs concerned with objects or operations which are exclusively hardware or system oriented. Consequently, most of most of our programs can be written in a general purpose language, and only occasionally will we require facilities peculiar to system programs. It is with the provision of such general purpose facilities in a suitable manner that section 'B' is concerned.

The language chosen as basis for the project was Algol68 [1]. The particular language chosen is not vitally important; most general purpose high level languages such as Algol 60, PL/1, or PASCAL would be equally suitable. Any language having a reasonably clean definition, powerful data structuring facilities, and good structures for ordering flow of control would be acceptable. Some of the languages embodying more recent ideas in language design, such as Alphard [2] or CLU [3] might suggest themselves as a better base language, but this would be a comment on the languages in general, not on their particular suitability for our present purposes. Since we are considering the minimal modifications necessary to provide a language system suitable for writing an operating system, and since work on a portable Algol68 compiler was already under way in Cambridge, this compiler was a suitable starting point. The Cambridge Algol68 compiler has developed in parallel with the CAP operating system, and since I was closely involved in both projects I have been able to take appropriate steps in the design and implementation of the compiler.

1.3 SPL Requirements

There are many features and facilities which are often described as being characteristic of SPL's, and before embarking on the remainder of this thesis it will be worthwhile to outline these. I do not necessarily agree that these actually are requirements, and this will be apparent as the description of the system actually produced develops. An assessment of which facilities are required has been given by Goos [4], although I disagree with him on several points.

It is clearly a necessary requirement that it should be possible to write every machine operation within the language. This should preferably take the form of in-line code, rather than separately compiled (or assembled) routines.

There must be facilities for the user to control storage allocation and management, but the system must provide suitable defaults. Similar remarks apply to the I/O models, and indeed to all code that is traditionally considered as 'runtime system'. In general, the system should provide a model which the programmer is free not to use.

It must be possible to write the whole of the operating system within the language. This necessarily includes writing the runtime system of the language, since otherwise we are merely indulging in a buck-passing exercise.

It is often recommended that facilities for packed structures be provided (as in PASCAL, MARY et al), but I am not convinced. One of the aims of this request is to describe hardware-defined objects, and I believe this to be a fundamentally misguided approach to that problem [C2.2]. The other aim is efficiency, which is laudable, but not fundamental since, in principle, the compiler is free to decide whether or not to use packed representations.

Some facility for parallel processing or synchronization is often requested. I believe it will become apparent later that this is not appropriate for the style of language system envisaged here. In the same vein, some mechanism for the handling of interrupts and other asynchronous events is certainly required, but I believe the library facilities for these described below [C1.1] to be sufficient.

Since an operating system is necessarily a large and complicated software package, some form of separate (and preferably modular) compilation will be required; this is discussed extensively below. With sufficient support in this direction, a language system can become a very powerful implementation tool.

1.4 The Approach Adopted

As noted above, the decision was made at an early stage to base the present work on an existing language, namely Algol68, rather than design a completely new one. It was expected that the bulk of the work would be in designing language changes to make Algol68 suitable as a system programming language, particularly for an architecture as unusual as the CAP. In practice, very few language changes have been made, although a complete operating system has now been written. Instead, the emphasis in the work has been in the careful design of the implementation to facilitate its use in writing the operating system, and it is with aspects of the implementation that the bulk of this thesis is concerned. Detailed consideration is given to separate compilation and library mechanisms, to the portability of the compiler, to the storage management techniques used for object programs, and especially to the provision of a runtime system for object programs.

Although the language unaltered is sufficient for almost all aspects of the operating system, there are situations where changes are needed. Some such changes which have been made are

described, but also some changes which seem desirable but have not been made - some of these are in the form of problems, to which the solution is not yet apparent.

The work being described has at most stages in its development been required to provide a usable system, and where techniques and solutions are described, they have generally been the subject of much practical exercise. I have tried throughout to avoid the danger of lapsing into a present or past tense when the future tense would be more truthful.

2. Background

2.1 The Algol68C Project

The Algol68C compiling system grew out of a small experimental system written in 1970 by S.R.Bourne and M.J.T.Guy in the University of Cambridge. They implemented a language called 'Z70', which bore some resemblance to the recently designed language Algol68. Z70 was an expression language (like Algol68); that is, almost any construct of the language delivered a value. For example, one could write:

```
(a:=b)+c
```

or even

```
a+(INT i=readint; i*i)
```

As such, its syntax and semantics were similar to Algol68, although Z70 was a much smaller language. The major difference, was that Z70 had only two data types ('integer' and 'function'), whereas Algol68 has an unlimited number, through the use of modes for describing structured values, for checking argument types in procedure calls, and for 'united' modes (used for values whose type varies dynamically at runtime).

Z70 was originally implemented on the Titan computer (a prototype ICL Atlas 2), using a syntax-directed compiler ('PSYCO', based on that produced by Irons [5]). PSYCO is driven by a set of production rules specifying the syntax of the language, and containing semantic rules causing output of compiled code at appropriate moments; output was in the form of 'IIT', a Titan assembly language. This original PSYCO system had some delightful properties, such as the message:

```
'.....' is not a 'PROGRAM'
```

Program debugging could be achieved by a binary chop technique!

This system was rapidly improved: some of the functions of PSYCO were taken over by a program written in Z70, the language was extended to include more of the features of Algol68, and PSYCO itself was modified, mainly to produce intelligible diagnostics and error recovery. By the middle of 1972, the system was recognisable as a deviant of Algol68 (ie. an 'extended subset', though this description could also be applied to FORTRAN). By this stage, the various components of the system were as follows. PSYCO was used as a syntax-directed parser producing a parse tree as intermediate output in polish prefix form; a Z70 program read this in, reconstructed the parse tree, and generated IIT. Finally, the IIT was concatenated with the

library (written in IIT) and assembled to produce an executable core image.

By the middle of 1972, the Z70 language still differed from Algol68 in several ways. Major omissions were: some of the intricacies of 'balancing'; multi-dimensional arrays; assigning, or yielding as value, of non-trivial objects (such as structures or arrays); user-defined operators; the Algol68 input-output system and most of the Algol68 library (called the 'standard prelude' by Algol68 devotees). There were several minor extensions, and a primitive separate compilation mechanism (which will be described later). 'Z70' was now renamed 'Algol68C', and released for use on Titan.

A decision was taken in 1972 to produce a new, portable, Algol68C compiler; in addition to being portable, the compiler was required to give good diagnostics, have good error recovery, and produce locally good code (though there was no intention to attempt any global optimisation). Its prospective uses were thought to be: student jobs, general algorithmic work and systems implementation (such as compiler writing and operating systems). The over-riding requirement was portability.

In view of previous experience of having part of the Z70 compiler written in Z70, and because the authors had found Algol68C a very convenient and easily debuggable language, and, mainly because of the portability inherent in such a system, it was decided to write the compiler itself in Algol68, using the subset available through the existing compiler. The system envisaged was as follows.

To implement Algol68 in its entirety, using a multi-pass compiler with intermediate outputs, requires a minimum of four passes to produce assembly code [6]. By a minor restriction on the use of indicants [7], this number is reduced to three; by keeping a parse tree as an internal data structure, the intermediate outputs are eliminated; to achieve portability, a specially designed intermediate language is used as the 'assembly code'. The compiler thus consists of three 'phases'. The first, called the 'parser', reads the source program, performs mode-independent parsing and links applied occurrences of identifiers to the corresponding definitions. Phase two performs 'mode equivalencing' (determining which textual representations of modes correspond to the same data type), 'balancing' (arranging that constructs whose flow of control has several exits (eg. conditionals) yield the same data type at each exit), the linking of applied occurrences of operators to the appropriate operator definitions, (the language allows user-defined operators), and the mode-independent parsing and error checking. Phase two was also intended to perform such storage allocations as could be determined at compile time, but to date this has been deferred until phase three. Phase three is straightforward in concept - it scans the parse tree (as modified by phase two) and outputs the intermediate code. In practice, phase three is tortuous, and

is not yet working entirely satisfactorily, but this will be discussed later. All error checking, except for those checks which inherently must be delayed until runtime (such as some scope checking and array bound checking), should be performed by the compiler, and the intermediate code output should not require any checking, except that some degree of checking is desirable for system debugging.

The intermediate code to be used, called 'ZCODE', will be discussed later. For each machine a 'translator' was to be written to read in the ZCODE, produce executable code, and handle linking to the library; a machine dependent library must also be written for each machine.

By keeping the source text of the compiler as machine independent as possible, portability would be achieved by conventional bootstrapping techniques [B3.2].

In September 1972, S.R.Bourne started work on the parser for this system; M.J.T.Guy was now mainly engaged on other projects, though he still provided much valuable advice, and was invaluable for his insight into design problems. In January 1973, I started work on phase two and the code for the Algol68C separate compilation mechanism. By June 1973, phases one and two were working well enough to accept the source program of themselves. At this time, S.R.Bourne departed for three months sabbatical leave. A group consisting of myself, I.Walker (another research student) and A.Andrews (a research assistant) started work on phase three (based on the Z70 code generator) and a Zcode translator for Titan. In late September 1973, the compiler successfully compiled itself on Titan; one week later, the Titan computer was finally closed down.

It then took three months to produce a working compiler on the IBM370/165 in Cambridge (although a system which worked by interpreting ZCODE was produced by I.Walker in November) - the system was working shortly before the end of December 1973.

During 1973, the intermediate code, Zcode, had been developed by the group, with some feedback from I.Wand of York University, who intended to transfer the system to their ICL4130. The detailed design of Zcode is defined elsewhere [8], and is likely to be the subject of other publications [9].

Further development of the compiler was done in 1974 by S.R.Bourne, M.J.T.Guy and myself. This included rewriting the parser error recovery, removing machine dependencies, and rationalising the mechanisms used to link the libraries (the consequences of the last two will be discussed in detail later). At the end of 1974, S.R.Bourne left the project, and the University of Cambridge Computing Service assumed responsibility for the IBM370 compiler. During 1975 I developed a Zcode translator and a library system to use the compiler for the experimental CAP computer, and bootstrapped the compiler onto

that machine.

A reference manual [7], defining and describing the language accepted by the system, was written by S.R.Bourne and myself, with help from I.Walker. I have written an implementors' guide [10], which defines the interfaces and auxiliary inputs of the compiler, and provides the information needed by someone wishing to transfer the system onto a machine.

2.2 The CAP Project

The Cambridge CAP computer is part of an on-going research project investigating protection techniques and their applications to operating systems. The project has had several phases. Firstly, a highly protected architecture, based on the concept of 'capabilities' [11], was designed [12]. Secondly, this architecture was realised as a processor, the CAP, built in the Computer Laboratory [13]. Thirdly, an operating system has been designed and built to run on the CAP [14,15,16], written almost exclusively in Algol68C.

The architecture, which is described in more detail elsewhere [17,18], supports a hierarchical process structure. Within a process, execution moves among protection domains known as 'protected procedures'. Each protected procedure consists of a number (one to three) of capability segments (ie. store segments known to contain only capabilities, which can be manipulated only by special purpose instructions). The capabilities in these segments specify the totality of facilities and privileges available to the protected procedure. No operation, be it executing an instruction, reading from or writing to store, sending a message on a software message channel or using a physical peripheral, can be performed without quoting the appropriate capability from one of these capability segments. The quotation of a capability is always implicit, by use of a 'general address'. It is a design principle of the CAP that there is no form of 'privileged mode' or 'supervisor state'; protected procedures which one would regard as privileged are privileged only by virtue of the set of capabilities to which they have access. It should be immediately apparent, then, that the programming environment for writing the innermost parts of the operating system differs from the most ordinary object program not structurally, but only in the set of capabilities available, and so programming tools providing the full freedom of one situation will also be suitable for the other.

Apart from the ordinary instructions which one would expect of any general purpose computer, the CAP has instructions concerned with capabilities and the protection architecture. There are:

MOVECAP to move a capability

REFINE to yield a more restrictive capability

SEGINF to enquire about a store capability's size and access

INDINF to ask about the contents of a capability

ENTER to transfer control to another protection domain, which is specified by a given capability. On return from that domain, control resumes immediately after the ENTER instruction

RETURN to make control revert to the calling regime

When a protected procedure is entered, control is transferred to a fixed address in that procedure. If the procedure wishes to provide multiple entry points or some more complicated entry arrangement, it must simulate this by software.

Initially, while the hardware was being built, the CAP was simulated interpretively using a mini-computer. During this period, cross-compilation of BCPL and Algol68C were made available from a local IBM370. A small single-user operating system (OS6, written in BCPL [19]) was implemented, and various experimental programs were written. When the hardware became available, OS6 was transferred to it, and the cross-compilation facilities suitably updated. In due course, compilation of Algol68C and BCPL were made available under OS6, compiling programs to run under OS6. Facilities were also made available to compile Algol68C programs (under OS6 or on the 370) to run on the stand-alone CAP. This allowed construction of the operating system proper.

The details of the operating system which has been constructed are described elsewhere and need not concern us here, but an outline of the structure may aid the reader. The system consists of two levels of process hierarchy; the top level has only one protected procedure, which is known as the 'co-ordinator' and is responsible for process scheduling, dispatching and synchronisation. The remainder of the system runs as sub-processes of the co-ordinator. These processes typically have one protected procedure unique to the process, and several protected procedures shared between the processes, providing services for the main protected procedure. There are several identical copies of some of these processes (such as the teletype handlers), while others are unique (such as those which implement the virtual memory system). A number of the processes are known as 'user processes'; in these the main protected procedure will, when initiated (for example, by a user at a teletype), identify a user then enter a nominated protected procedure, typically an interactive command program which allows the user to execute his programs as protected procedures entered from the command

program.

At an early stage in considering the design of the operating system, the decision was taken that the system should not be a single program in the language sense. That is, the total system should be a set of separate programs, communicating together and co-operating. This decision was taken as part of the operating system design, irrespective of any similar decision that might be (and was [C3.1]) taken by designers of language systems. In terms of the operating system design, it has several advantages. A principle of the architecture is that each protected procedure should know nothing about the internal working of another procedure - this is most readily fulfilled by treating each procedure as a separate program. The operating system, given this decision, could readily be written in whatever language seemed most appropriate for a particular protected procedure. This decision also forced the use of interfaces specified in a language independent manner, since the calling and called procedures do not in general know in which language each other is written. In practice, all but one protected procedure of the operating system are written in Algol68 (the one is written in assembler, on efficiency grounds, so that it does not use any workspace - a feat no other compiler was capable of achieving); however, one must remember that in any case the command program, entered as a protected procedure from the operating system, is not part of the system and may be an arbitrary program written in an arbitrary language.

We should note that the CAP project has thus exhibited many of the requirements typical of SPL applications. The compiler itself has been required to run on different machines and various (and varying) operating systems. Object programs have run in environments ranging from the empty machine, through protected procedures in system or user processes, to the command program and ordinary user object programs. Some exercise enormous privileges (though only when necessary for their task!), others have none; some run in a virtual memory environment, others are confined to resident real store; some perform explicit I/O orders, others communicate with I/O processes, still others with in-process I/O protected procedures, while some are unable to use any I/O facilities at all. Some form of separate compilation mechanism is necessary to share program code between differing protected procedures. Interfacing with protected procedures presumed to be written in other languages can present great difficulties in the handling of data objects [C2.2]. Note that neither the co-ordinator, nor many of the system processes can be allowed the luxury of pausing for a garbage-collection - in a language as complicated as Algol68, this requires the most careful arrangements for storage management.

The present CAP operating system is now running; further development is still under way, and systems under different experimental architectures may be developed from it. I believe those involved in writing the operating system have been pleased

(and pleasantly surprised!) with their experience of using the Algol68C system [15].

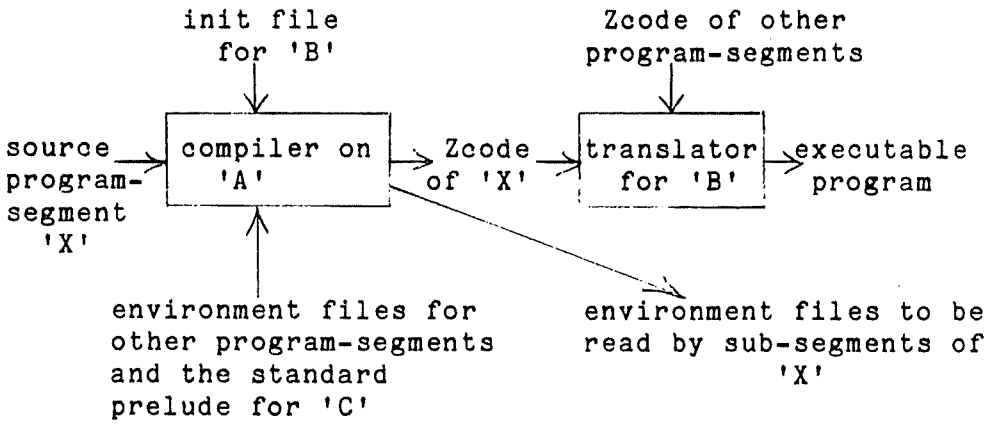
Section B: High Level Language Implementation

1. Today's Algol68C System

Subsequent chapters of this section describe those features of the present Algol68C system which I consider in some way original (either because they have not, to my knowledge, been done before, or because they have not been viewed in some particular way before), and which I have designed or to whose design I have materially contributed. The teamwork involved in the Algol68C project was such that each of us discussed ideas with the other, and it is impossible to say absolutely "this is mine"; any project of which that could be said seems doomed to failure through lack of co-operation.

In order that the individual topics to be discussed may be viewed in relation to the complete system, I will give here an overview of the Algol68C system. The central core of the system is the compiler. This is a large Algol68C program (about 18000 lines), and represents the major programming effort. It accepts the source form of a 'program segment' (the item of separate compilation), and produces Zcode. Zcode is in fact a family of intermediate codes, differing in such things as the number and properties of the registers and the size and layout of stored objects; these differences are specified by the 'initialisation file' read in at the start of each compilation. Linking between separately compiled program-segments is determined at compile time by reading or writing 'environment files', and is expressed in the Zcode by program labels and access to common store. The library and runtime system is linked to in precisely the same manner, as if it were an Algol68C program-segment. The translator is a separate, machine dependent program which reads in Zcode and manufactures executable code.

The system, compiling on some machine 'A' for some machine 'B' (which may be the same as 'A') with operating system 'C', may be pictured as follows.



This document does not describe the detailed technicalities of any of the algorithms or interfaces: these are described elsewhere [10,20].

2. Separate Compilation Mechanisms

2.1 Requirements

Separate compilation mechanisms can be found in many programming languages. Even the earliest language having any claim to be at all 'high level', FORTRAN, is usually implemented with a very versatile independent compilation mechanism, even in unsophisticated implementations, albeit with no error checking.

The most general description of separate compilation is as follows: take a piece of source text, 'A'; process it in some way (eg. by a compiler!) into a partially compiled form, 'X'. Some time later, some source text 'B' is presented to the compiler, such that A+B forms the source of a complete program. 'B' is processed by the compiler, and the result is in some way combined with 'X' to form complete executable code of the program A+B. There are several reasons for wanting to undertake such an operation.

If, because of some bug or the whim of the programmer, we subsequently want to alter the 'B' part of the program, only 'B' need be recompiled, then recombined with 'X'. This is an argument of efficiency, and can to some extent be blamed on compiler writers: if we wrote faster, cheaper compilers then the programmer would be less tempted to adopt such strategies. (This argument has been put forward by Hoare [21].) If computers were infinitely large (or infinitely fast) ...

Separate compilation allows the source program to be segmented. This is an argument of data management: with 50,000 line programs (and such programs, unfortunately, exist), maintenance and debugging of a single source file is a major problem. Also, if several people are working on the program then treating it as an indivisible monolith will lead to disasters (like simultaneous editing). Having a segmented source reduces such problems. Further, if the source is segmented along boundaries having well-defined interfaces, some amount of independent debugging may be possible (and well-defining the interfaces is likely to improve program 'correctness' and make modification simpler). This approach can be taken further, to allow two programmers to work independently on 'A' and 'B', both working to the same interface, possibly with compiler assistance to ensure that the interface is met.

In terms of 'A', 'B' and 'X' as above, we can envisage some other piece of source text 'C', such that A+C is also a program. In such a case, we need only compile 'C' and combine it with 'X' (the partially compiled form of 'A') to produce an executable program. This, of course, makes 'X' the pre-compiled form of a library, 'A'. Libraries can exist without separate compilation -

the programs A+B and A+C could merely share the source text 'A' - separate compilation has just allowed this sharing to occur at some lower level (eg. assembly code). At first sight, then, this would appear to be another instance of overcoming the expense and inefficiency of badly written compilers. It has, however, more fundamental consequences: with care on the part of the implementer, and assuming suitable loaders, hardware, etc., it is possible to share libraries such as 'A' at the level of binary in the computer. That is, it may be possible (particularly in segmented machines, but also in simpler ones) to physically share 'X' when the programs A+B and A+C are running. When such sharing facilities are available, they can produce large savings in valuable resources (such as store, load time, swapping), and are thus highly desirable. I can see no way, in high level language terms, of achieving such sharing without a powerful separate compilation mechanism. We will see later how this aspect of separate compilation is used in implementing the Algol68C runtime system for CAP.

Care must be taken to distinguish between 'separate' and 'independent' compilation. Mechanisms which achieve separate, but not independent, compilation occur when some of the output of an earlier compilation ('A' above) is used as input to a subsequent compilation ('B' or 'C' above). A degenerate form of this would be to pass on the source text of 'A', but most mechanisms do better, or claim to, by passing only information from 'A' that can affect the compilation of 'B' or 'C'. It is entirely possible to do without such information passing, and several systems do so. Perhaps one of the reasons for the perseverence of FORTRAN is that it is usually implemented with completely independent compilation of subprograms, allowing much pre-compilation of libraries. Unfortunately such simplicity and versatility does not come without its disadvantages; lacking, at compile time, information about 'A', the implementer faces a choice when compiling those parts of 'B' which depend on 'A'. He can assume either that information collected in 'B' about its interface with 'A' is correct (for example, the number and data types of arguments of procedure calls, or special statements included in 'B' making assertions about 'A'), or he must devise some means of checking this information later, when 'X' is available - this requires, usually, load time or run time checking. I consider the approach of not checking such information to be totally unacceptable - it is the source of many obscure bugs. Checking as late as load time at least avoids the bugs, but it can be expensive, may require a special loader, and produces worse diagnostics. Checking at run time is even worse - an operating system written using such an implementation which suddenly said 'INT cannot be coerced to REAL' on its operator console would not be welcome. Further, having the information passed at compile time avoids the programmer having to give the compiler so many extra statements (such as FORTRAN COMMON statements), although the extra statements, which increase the redundancy in the source, can be used to improve the error checking. We will see below that separate compilation, even

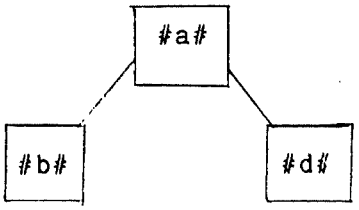
though not independent, can be arranged to have few drawbacks.

2.2 Separate Compilation in Algol68C

Different Algol68 implementations have different separate compilation mechanisms, and their implementers will each defend their own mechanism with vigour. In this section I will describe and defend, but also criticise, the Algol68C mechanism. To arrive at the Algol68C mechanism, we need only consider dividing a block structured program into several pieces. Consider:

```
BEGIN#a#
    .
    .
   INT i:=3;
    .
    .
   BEGIN#b#
      .
      .
      i:=7;
      .
      .
   END;
   .
   .
   PROC INT f = BEGIN#d#
                  .
                  .
                END;
    .
    .
END#a#
```

We can represent the (syntactic) structure of this program as a tree:



Each of these boxes can form a separately compiled program-segment in Algol68C. In general, given a program, any 'unit' (the basic syntactic construct of Algol68) may be separately compiled, even if it delivers a value. Taking the above example, the first program-segment presented to the compiler (call this segment 'A') would be:

```
BEGIN#a#
    .
    .
   INT i = 3;
    .
    .
   ENVIRON B;
    .
    .
   PROC INT f = ENVIRON D;
    .
    .
END#a#
```

The remainder of the program would subsequently be presented as segments 'B' and 'D':

```
        USING B FROM "file title"
        BEGIN#b#
           .
           .
           i:= 7;
           .
           .
        END
```

and:

```
        USING D FROM "file title"
        BEGIN#d#
           .
           .
        END
```

Each program-segment may define several 'ENVIRON's, and this, like block structure, may be nested arbitrarily. The semantics of the segmented program are identical to those of the original program - when the flow of control reaches an 'ENVIRON', control is transferred to the appropriate separately compiled segment; at the end of that segment, control reverts to the segment that contained the 'ENVIRON'. Each separately compiled segment is written by the programmer precisely as if it occured textually in place of the corresponding 'ENVIRON': the flow of control is the same, and the meaning of identifiers and indicants is the same. The only special statement is the 'USING' directive, which specifies which 'ENVIRON' is associated with the program-segment.

This mechanism is implemented by passing information between compilations in 'environment files'. When a program-segment containing 'ENVIRON's is compiled, in addition to the Zcode output, an environment file is produced. When subsequently a program-segment associated with one of these 'ENVIRON's is compiled, the environment file is read. These files contain all the information necessary to continue the compilation of the 'ENVIRON', in particular such things as declarations, storage allocations, label allocations. Thus in the above example, the

compilations would be as follows:

1) compile segment 'A', producing Zcode 'X' and environment file 'E'

2) compile 'B', reading environment file 'E', producing Zcode 'Y'

3) compile 'D', reading environment file 'E', producing Zcode 'Z'

Note that steps (2) and (3) could be in either order, or even simultaneous. The Zcode files Y and Z may each contain references to labels in 'X', and 'X' will contain references to the entry labels of Y and Z. Either the Zcode translator, or some subsequent processor such as a linking loader must resolve these label references. Some details of the implementation of environment files can be found elsewhere [10].

The Zcode translator for CAP takes as input the Zcode for a separately compiled Algol68C segment, and assembles the directly executable binary for a single machine segment. In order to allow resolution of references from subsequently compiled segments to this one, the translator outputs a 'linking file' if the current segment contains any 'ENVIRON's, and this is automatically read when translating those segments which use the ENVIRON's. The only references from this segment to the subsequent ones are jumps to the entry points, which are resolved by the translator allocating the machine segment numbers at which the subsequent segments will be placed. This thus allows preassembly of physically sharable segments, a very powerful facility which is used for the runtime system and for code shared between programs. This segmentation, in the virtual memory environment provided by the CAP operating system, corresponds to an overlay mechanism.

This implementation of separate compilation relies on the sufficiency of a one way flow of information at compile time between separately compiled segments; that is, it relies on the compilation of 'A' not depending on the content of 'B' or 'D'. In other words, it relies on the compilation of a construct containing a 'unit' not depending on the content of that unit. This condition, desirable though it may be (from a 'structured' point of view, apart from separate compilation), is not satisfied by Algol68. Firstly, compilation often depends on the data type delivered by a unit; for example, the operator '+' is defined between integers (as you would expect) and between strings (for concatenation) so that, for example,

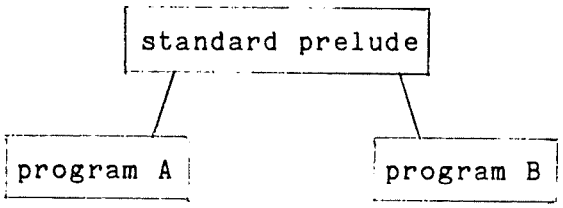
```
(ENVIRON A) + (ENVIRON B)
```

would leave the implementer not knowing which '+' was meant. (Other examples where the meaning depends on the mode delivered by a unit include the destination of an assignment and the

primary of a procedure call.) To avoid this problem, we allow 'ENVIRON' only in a context where we know which mode must be delivered ('strong' contexts, in Algol68 terminology), such as the source of an assignment; the subsequent compilation using the 'ENVIRON' will check that this mode is in fact delivered. The other instance of dependence on the content of a unit occurs when the unit is inside a routine; here, applied occurrences of identifiers inside the unit may affect the 'scope' of the routine, by affecting which stack frames may be accessed by the non-local applied occurrences in the routine. That is, they may affect the environment necessary for the routine. (In an implementation using a classical chained stack, where a routine is represented by its entry address and an environment pointer, passing a routine outside its scope makes its environment pointer refer to a stack frame that no longer exists.) This difficulty was overcome by specifying that a routine containing an 'ENVIRON' has minimal scope. The situation would be more satisfactory if constructs were less affected by the detailed content of their sub-constructs.

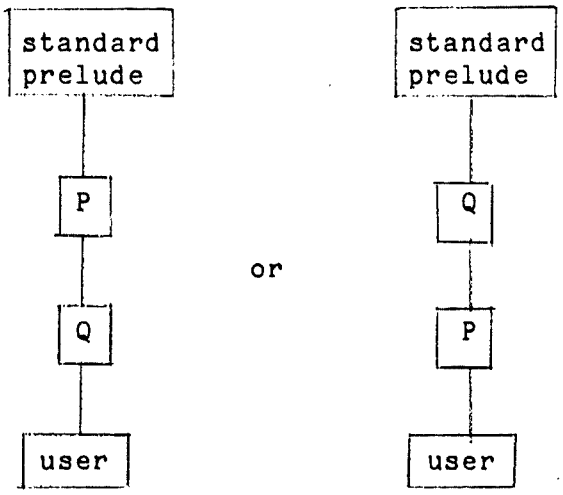
The major advantage of the Algol68C separate compilation mechanism is its simplicity. Because we treat separate compilation as a process of taking a program and subdividing it, the programmer has no extra concepts to learn - he can think of his source as a single program which has merely been textually divided. The error checking on a separately compiled segment is as thorough as it would be without separate compilation, since all the information is still available at compile time and we do not rely on extra assertions made by the programmer. No extra work need be imposed on the subsequent processors - the labels used for inter-segment references need present no more severe problems than in-segment references. This separate compilation mechanism is also used for linking to the run-time system, but this will be described later [C1.1].

Thus the Algol68C mechanism is highly satisfactory in achieving our primary aim of splitting a large program into separately compiled segments - the division can be made at any 'unit', with no serious constraints. If we consider our secondary aim, of providing pre-compiled libraries, it is less satisfactory. If (as in the CAP Algol68C system) we treat the standard prelude as an ordinary program segment, then it is the prime example of a pre-compiled segment shared between programs. Thus:

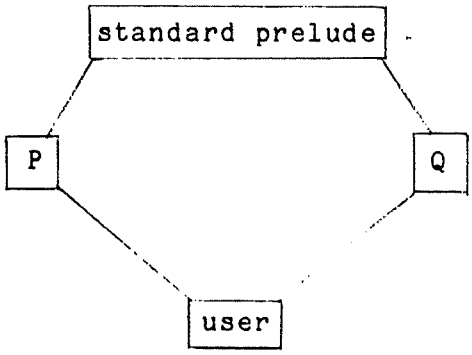


For this, the Algol68C mechanism is still satisfactory. However,

if we now desire also to have two libraries P and Q, which do not refer to each other, there is no satisfactory way to use the Algol68C mechanism to compile them separately; there are only two arrangements that would allow pre-compilation of P and Q while allowing a program to access them both. We can either compile Q as an inner block of P, or vice versa. Thus:

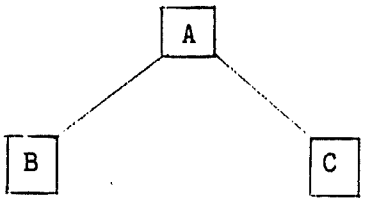


Either is unsatisfactory since, in the first version say, when Q is compiled inside P we have two undesirable effects: firstly, applied occurrences in Q can identify definitions in P when definitions in the standard prelude were intended (ie. P shields the standard prelude from Q); secondly, if Q defines some item of the same name as one in P then the user program may identify the wrong one without being informed that this shielding has occurred. Note that these effects are precisely what would occur if we compiled the corresponding single program with the above block structure. The general case, where we wish to provide a large selection of libraries P, Q, R, ... has precisely analogous problems. To try to avoid these effects in a single program we could place P and Q as parts of a single serial clause; then neither would legally shield definitions in the other, but the first one could still shield standard prelude definitions required by the second. Thus the facilities we desire for independence of the libraries P and Q can in no way be provided by a mechanism that follows the identification rules of conventional block structure. The structure we would like to have can be represented as:



This differs fundamentally from block structure, but is an essential practical requirement for a system wishing to offer pre-compiled libraries. We will show later how to provide this facility by a simple extension to the language [B2.4].

The present implementation of the Algol68C separate compilation mechanism suffers from a severe drawback - there is a constraint on the order in which the segments can be compiled. Briefly, an outer block must be compiled before an inner block. For example, if the structure is:



then we must compile A first, then B and C (in either order, or even simultaneously). This constraint comes from quite reasonable origins, since to compile B or C we need to know quite a lot about the definitions occurring in A. The most extreme examples of this are mode definitions and operator definitions. For example, if the segments are:

```
BEGIN #A#
      INT max = readint;
      MODE V = [1 : max] INT;
      OP + = (INT i, j) INT:.....;
      BEGIN
           INT max = 7;
           ENVIRON b ; ENVIRON c
      END
END
```

```
BEGIN #B#
  print (max + 3)
END
```

```
BEGIN #C#
      V w;
      FOR i TO UPB w
       DO w[i] := readint OD;
      print (w)
END
```

Here we would have grave difficulty in compiling B or C before A. In segment B we would not know which version of the polymorphic operator '+' to use, while in C we would know nothing about the mode 'V'. If we require complete freedom in the order of compilation, the only reasonable solution is then to require that a segment be prefaced by a list of those definitions which the segment requires from other segments, or that a separate 'interface file' be provided. This, some implementors are attempting [22]. Note that this approach should only be adopted if the implementer is willing to check, either when compiling A or at load-time, that the assertions made in B and C about A are in fact correct. Producing reasonable diagnostics in the case where the assertions are wrong would appear to be difficult. Since Algol68C allows separate compilation of any unit, we find that allowing such freedom in the order of compilation will entail serious run-time penalties on many systems. If we assume only a linking loader as provided on conventional operating systems, rather than a specialized Algol68 loader, then when compiling code in B or C for applied occurrences of definitions in A we must make unreasonable assumptions - such as assuming which display level is to be used to access the definition. The details are too complex to be worth investigating here, but the only general solution is to have an indirection vector for accessing definitions of A. This is a serious degradation in run-time efficiency (in both store and time).

It is thus in some ways undesirable to allow complete freedom in the order of compilation in Algol68C; I do not believe that freedom to this extent is really necessary, either. Experience with the Algol68C system so far has indicated that programmers do not object to compiling their segments in the prescribed order. What they object to is, when A is recompiled, being required to recompile B and C even though the texts of B and C, and their 'meaning', have not changed. What I believe to be highly desirable is to minimise this re-compilation; this can be more

readily achieved than complete freedom. There are several techniques which can help here - only the first has so far been used in Algol68C.

The first, and most obvious way of reducing the amount of re-compilation needed is to arrange that if the changes made to A are such that they do not affect the 'meaning' of B and C, then the interface (as specified by the environment file produced when compiling A) does not change. It is difficult to achieve perfection here. Some changes to A, such as changes inside the bodies of routines, should clearly not affect B or C . Adding or removing, or changing the mode of, a definition accessible to B or C clearly does change the environment; changing the value ascribed to an identifier of mode INT may or may not change the environment, depending on whether the compiler feels able to calculate the value at compile time. In order to make this a reliable system, we would need a facility for reading the old environment file and saying whether it differs (significantly) from the new one - a message could then be printed saying whether re-compilation is necessary. No such facility exists at present in Algol68C.

If the compiler reads the old environment file when compiling 'A', then with only a little additional complexity we could do much better. Since it would then know the stack and label allocations used in the previous compilation, the compiler could endeavour to use the same allocations in this compilation - if a definition is added, for example, it could be placed on the stack after the original ones, rather than shuffling them. It is possible, using this approach, to maintain a considerable degree of compatibility with an old environment, although there would still be occasions when we fail.

Re-compilation can be further reduced if we add to the language a facility to hide definitions. For example:

```
INT a,b,c;
HIDE (b,c) ENVIRON B;
```

Here only 'a' could be identified from inside the 'ENVIRON', so changes to the definitions of 'b' and 'c' should not entail re-compilation of sub-segments.

It is thus possible to considerably reduce, but not completely eliminate, re-compilation without including in the sub-segments assertions about the earlier segments. Note that the approach of including assertions is less satisfactory, also, when we wish to share the outer block between programs (as with the standard prelude) - here the techniques given above seem much more satisfactory.

We will see later that re-compilation is a less difficult problem in the library mechanism proposed below.

Thus the present Algol68C separate compilation mechanism is highly satisfactory for dividing up large programs (no alternative mechanism of comparable flexibility has yet been proposed); it could readily be implemented such that re-compilation is not a serious problem; it is a disaster for implementing pre-compiled libraries, but so is block structure itself.

2.3 Separate Compilation in Other Systems

We now consider what some other implementers have done about this desirable but elusive property, versatile separate compilation. We are looking for three gains, as described above [B2.1], namely: segmentation of source text, ease of recompilation of modified segments, and 'libraries' (ie. sharing of pre-compiled segments).

Undoubtedly the most versatile and successful separate compilation system is usually found in FORTRAN implementations. Here it is generally possible to compile any sub-program (ie. any FUNCTION, SUBROUTINE or BLOCKDATA) completely independently. The only communication the language allows between sub-programs is by calls and by common statements. There is enough information in each sub-program for the compiler to determine all it needs to know about any other sub-program referenced from this one. In principle, this system is ideal (if the language were not FORTRAN, but that is a separate argument); it satisfies all three of our aims, and large computer systems make successful extensive use of pre-compiled FORTRAN libraries. All that the implementer need do to make such a system ideal in practice is to check that information gathered from a subprogram about some other subprogram (eg. number and type of arguments) agrees with the definitions in that other subprogram. There is nothing to prevent this checking being performed at load time, but very few implementations perform it at all, even at run time. (The Titan FORTRAN system was one of the few to check that calls of an independently compiled subprogram had the correct number of arguments; I know of no-one who does the complete checking, and no-one who checks at load time or earlier.) Such checking is not at all difficult to implement (the general approach is obvious), but no-one seems to bother, and standard system loaders, which are usually designed specifically for FORTRAN, seldom provide such facilities.

The FORTRAN separate compilation mechanism is, in some ways, similar in design to Algol68C. One first decides what a complete program looks like, then subdivides the program into separately compiled segments at some natural boundary ('sub-programs' in FORTRAN, 'units' in Algol68C). Where FORTRAN gains is that its language is sufficiently modular for the segments to be compiled independently - the meaning of a sub-program is not affected by any other sub-program, whereas Algol68 units depend very much on

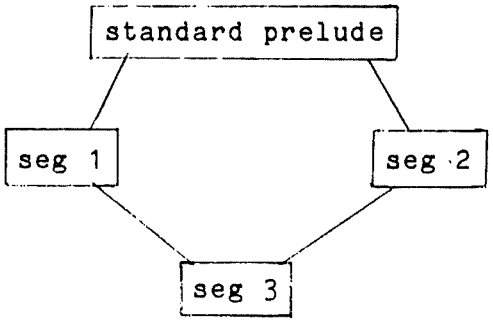
external information for their meaning. (It is interesting, in these days when language designers are producing constructs to promote 'modular programming', that FORTRAN is one of the most highly 'modular' languages around.)

A mechanism which more closely resembles that of Algol68C is the Algol68R mechanism of 'albums' [23]. Here, information is passed between compilations, as in Algol68C, but the overall structure provided is different. In Algol68C the segments form a tree whose root is the standard prelude; in Algol68R they form an acyclic directed graph having a single origin, the standard prelude. For example:

```
seg 1                              |   seg 2
BEGIN                              |   BEGIN
  PROC f = (INT i)INT:i*i:         |     PROC g = (INT j)INT:-j;
  SKIP                             |     SKIP
END                                |   END
KEEP f                             |   KEEP g
FINISH                             |   FINISH
```

```
seg 3 WITH seg 1, seg 2 FROM filename
BEGIN print (f(g(7))) END
FINISH
```

Here the segment structure is:



The restrictions imposed by this mechanism are similar to Algol68C: each segment must be compiled after any segment specified in its 'WITH' statement, and the only sharing permitted is of a subgraph starting at the root. As far as I can tell from their documentation, the recompilation rules for Algol68R are similar to those for Algol68C. However, since their segment structure does not correspond to a block structured program, it is possible for a segment (such as 'seg 3' above) to access definitions from several independently compiled shared segments (such as 'seg 1' and 'seg 2' above); this is thus a much more powerful library system than Algol68C allows, and is the major advantage of the Algol68R system. Since, as mentioned above, the

set of segments in an Algol68R program does not correspond directly to a single piece of Algol68 source text (we showed above [B2.2] that such correspondence forbids simultaneous access to independently compiled library segments), Algol68R has to describe separately the flow of control between segments. The designers of this Algol68R scheme have thus allowed the structure which we found lacking in the present Algol68C system. In practice, to date Algol68R only exists on the ICL1900 series, where a dedicated specially designed linking loader is used for Algol68R segments. Their requirement for a specialized loader appears to come more from their use of a single pass compiler than from the separate compilation mechanism, but it does give them more freedom.

A system with similar overall structure to Algol68R, but with no constraints on the order of compilation, has been proposed, and is being implemented, by a team in Berlin [22]. The freedom in the order of compilation is there achieved by prefacing each segment with a list of the definitions which that segment requires from other segments. Inevitably, a specialized linking loader is required to perform the final parts of type-checking. Again, the overall flow of control between segments is described separately.

Although Algol68R and Berlin provide a facility sorely lacking in Algol68C, I find their systems unsatisfactory in two ways. Firstly, the set of textual fragments which forms the complete source of a program does not correspond to any single piece of source program; the meaning of the set of fragments is determined by a set of ad hoc (and in practice, ill-defined) extra-lingual semantic rules. Secondly, there are facilities available in the Algol68C system which are not available in the others; that is, there are desirable ways of splitting up large programs which Algol68C allows and the others do not. This can be illustrated by two examples (taken, in fact, from the source of the Algol68C compiler). Consider a program of the form:

```
BEGIN.
      .
      .
      CASE i
      IN .
         . # very large number of cases #
         .
         # 37 # (.....),
         # 38 # (.....),
         .
         .
      ESAC
      .
      .
END
```

In a situation where the CASE clause is too large to compile it

conveniently as a single piece of text, only a scheme like the Algol68C one will suffice. Under other schemes, we would have to make those units of the CASE which we wished to compile separately into routines, then call them from inside the CASE. Although this would achieve the desired effect, it would disturb the desired structure of the program. An example more difficult to resolve by any scheme other than the Algol68C one occurs as follows:

```
BEGIN.
     .
     .
     PROC a = (ARG g) RES:
          BEGIN
              PROC b = (ARG h) RES:
                   BEGIN.
                        .
                        ...p:= g;....;
                        .
                        CASE i
                        IN...., b(x),.......a(y),....
                        ESAC
                        .
                        .
                   END;
                .
                .
          END;
     a(z)
END
```

A little explanation is required, although the structure of the program is really quite simple. The routine 'b' is nested inside 'a' in a non-trivial manner - not only does 'b' call both 'a' and 'b' recursively, but also 'b' contains applied occurrences of definitions local to an activation of 'a'. No scheme yet proposed in the style of Algol68R or Berlin will allow me to compile separately the body of 'b' (which is, in fact, a large piece of text). In Algol68C this separate compilation is straightforward, as for any other unit.

The examples which the Algol68C scheme allows, and no other scheme allows, are those where the position of the piece of text we wish to compile separately has implications for the flow of control or for the identification of non-local, but non-global definitions.

2.4 A Complete Separate Compilation Scheme

We have seen above how distinct approaches to separate compilation in Algol68 have arisen, each with its advantages and

its disadvantages. I wish to show here how the disadvantages can be eliminated, and a single scheme be produced having the advantages of both. The following scheme, and many variants of it, are the subject of continuing investigation by a sub-committee of IFIP WG2.1; the scheme depends heavily on previous and current ones produced by members of that sub-committee and by others. The members of the sub-committee are myself, H.Boom, C.H.Linsey and R.Dewar. The view presented here of the work is my personal one, and might be disputed by others.

The objection to the Algol68C scheme is that there are necessary facilities which it does not provide. These we can provide by adjoining a scheme like Algol68R or Berlin, but those have the objection that the meaning of the complete program becomes ill-defined, since they do not correspond to existing language features. This objection we can resolve by defining a language feature providing the appropriate facilities. Such a feature can be based on 'definition modules' - first proposed by S.Schumann [24]. Definition modules are similar, in the facilities provided, to Simula 'classes' [25] or to CLU 'clusters' [26]. For example:

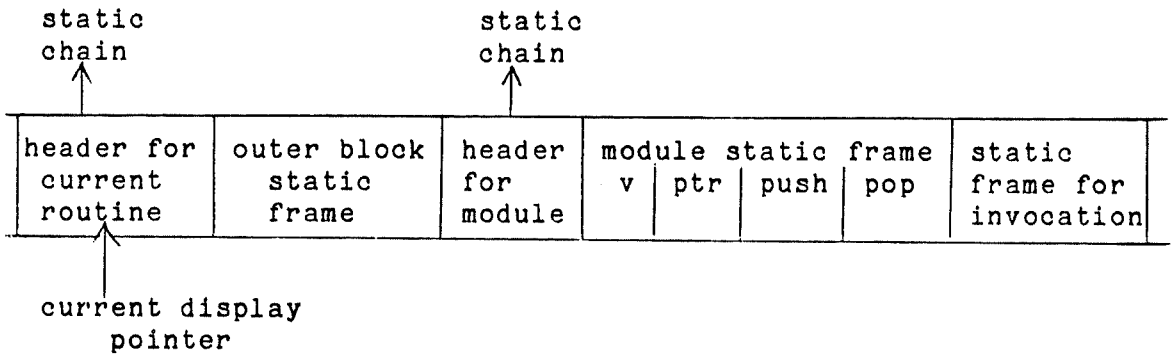
```
MODULE STACK = DEF [1:100]INT v; INT ptr := 0;
                   PUBLIC PROC push = (INT i)VOID:
                              v[ptr+:=1] := i;
                   PUBLIC PROC pop = INT:
                              v[ptr-:=:=1];
                   GAP;
                   print("Stack finished")
               FED;
```

This declares a module known as 'STACK', which has a 'prelude' declaring two private local definitions and two public ones, has a 'GAP', and has a 'postlude' which prints a message. This module could subsequently be invoked by a statement of the form:

```
INSIDE STACK ( FOR i TO 10 DO push(readint) OD;
               FOR i TO 10 DO print(pop) OD
             );
```

The invocation has the following effects: the prelude of STACK is executed (executing its definitions and initialisation), then the unit written at the invocation is executed instead of the GAP, then the postlude of STACK is executed. Non-local applied occurrences inside the module identify definitions according to the normal textual rules, except that the public definitions of the module are available as a layer immediately outside the unit of the invocation. An invocation can readily be implemented on a stack system that uses one display level per routine without increasing the number of display levels - the static frame for the module is adjacent to, and at the same display level as, that existing before the invocation. In the unit of the invocation, we know the static offsets for the

definitions made available by the module. The environment pointer for routines of the module point to an extra stack frame header at the start of the static frame for the module; this header contains the appropriate static chain pointer. Thus:



Assuming the existence of such a language feature, we can achieve a separate compilation facility allowing libraries by allowing separate compilation of definition modules. For example:

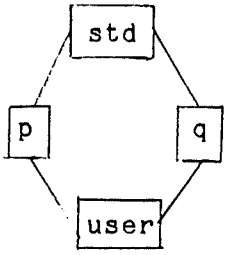
```
1)  BEGIN #standard prelude#
        .
        .
        MODULE P = SEP '.libs.p';
        MODULE Q = SEP '.libs.q';
        ENVIRON '.libs.std'
    END

2)  DEFINE '.libs.p' DEF#p#.....FED

3)  DEFINE '.libs.q' DEF #q#....FED

4)  USING '.libs.std'
    INSIDE p,q BEGIN
                .
                .
            END
```

This would correspond to our desired structure for allowing libraries:



Compilation of these segments is not difficult. Segment (1) produces files describing the environment for compiling (2) and

(3); these produce files used when compiling (4). The meaning of the complete program is well-defined, using our new language feature. This achieves independent compilation of the two libraries, but there are still problems. The first of the problems is that we still require an order of compilation; the possible solutions to this are the same as described earlier [B2.2]. The second is that in (1) we have had to specify the name and number of separately compiled modules. This would mean that if we wished to add another library (or if a user wished to make publicly available a private library), we would have to alter and recompile (1). This is unsatisfactory, and difficult to resolve tidily. The best proposal to date is to omit the module definitions from (1), and to specify the name of the ENVIRON in (2) and (3). Thus:

```
1)  BEGIN.
        .
        ENVIRON '.libs.std'
    END

2)  DEFINE '.libs.p' USING '.libs.std' DEF.......FED

3)  DEFINE '.libs.q' USING '.libs.std' DEF.......FED

4)  USING '.libs.std'
    INSIDE '.libs.p', '.libs.q' BEGIN.
                                    .
                                    .
                                END
```

This would allow pre-compilation of an arbitrary number of libraries.

One question remaining about definition modules is, if, in a program, there is more than one invocation of the same module, then under what circumstances are the invocations shared? That is, when do we allocate a new stack frame for the module? For example, with the STACK example, when do we allocate a new stack? There appear to be three reasonable answers - two extremes and a compromise. The first extreme would be to always allocate a new stack - this we term 'unshared'; this is similar to the arrangements for SIMULA classes. The second extreme would be to always share - this we term 'dynamic sharing', since we can determine whether sharing occurs only at run-time; this corresponds to FORTRAN COMMON storage - the storage for a module can be allocated statically, once and for all. The compromise is termed static sharing - we share if an invocation occurs in a textually enclosing block (or invocation); this can be determined at compile-time. The objection to dynamic sharing is that it seems un-structured - whether we get a new invocation depends on unrelated parts of the program. No sharing is tidy, but likely to be wasteful, except in the rare cases where it is essential to the correct functioning of the program. The most satisfactory choice appears to be static sharing, probably with a facility to

request an un-shared invocation.

3. Compiler Portability

The major purpose of any high level language is to allow the programmer to express his algorithm in terms of objects and operations which are closer to his problem than is the hardware of a particular machine. If his problem is an abstract one (such as inverting a matrix)), then in a suitable language (such as Algol68, PASCAL, PL/1, FORTRAN) he can program the algorithm without any knowledge of the machine on which he is running. However, even a problem that is inherently dependent on some computer hardware (such as converting a source program into machine code for a particular computer) may be programmed without knowledge of the computer on which the program is running. In the case of a compiler, the problem is inherently dependent on the machine on which the program will run (the 'target' machine), but the transformations required are, in general, independent of the machine on which the compiler is running (the 'source' machine).

Any necessary dependence on the source machine is concerned with how the compiler reads the source program; the most common such dependence is the character code, but it may also be encountered in other aspects, such as system dependent file titles. The compiling techniques used, or the language in which the compiler is written, may introduce other dependencies, but these are not necessary (although such considerations as the size or speed of the compiler may make the introduction of such dependencies desirable).

Now consider a programmer wishing to write an operating system for some machine 'A', in a high level language (Algol68!). Before an operating system is available for 'A', program development must be done on some other machine, 'B' say (at least to the extent of compiling on 'B' and transferring compiled programs to 'A'). So, initially, the compiler must run on 'B' (but as discussed above, the compiler need only be dependent on 'A'). Subsequently, when some sort of system is available on 'A', the programmer will want to dispense with 'B' and run the compiler on 'A'. To avoid rewriting the compiler, it is natural to write it in its own language (Algol68), compile it once on 'B' to produce code for 'A' then run it on 'A', where it can quite happily compile itself. However, to run the compiler initially, on 'B', we must also have an Algol68 compiler capable of producing code for 'B'. We are thus led to the conclusion that, to write an operating system for 'A' we want a compiler capable of running on 'A' or 'B', written in its own language, and capable of producing code for both 'A' and 'B'.

This is a conventional 'bootstrapping' situation, and several systems have used this approach. To enable the compiler to produce code for more than one machine, the compiler can be arranged in two parts. (In what follows I will use 'machine

dependent' to mean dependent on the machine for which code is being produced; dependencies on the machine on which the compiler is running are of little importance.) The first part of the compiler is machine independent; it reads the source of the program (or program-segment), performs some work on it, and outputs some form of intermediate code. The second part of the compiler is machine dependent; it reads in the intermediate code and produces machine code for the appropriate machine. To produce machine code for some new machine, only the second part of the compiler need be changed. The Algol68C compiler is similar to such a mechanism, but before considering Algol68C, we will look at how other bootstrapping compilers approach the problems.

3.1 Other Intermediate Codes

There are several systems whereby the machine independent part of the compiler is written in its own language, producing some intermediate code, notably BCPL [27] and PASCAL [28]. The major distinguishing feature of such systems is their intermediate code, and it is upon the choice of this code that the portability and efficiency of the compiler depend. The 'choice' of code, in practice, means designing a code specifically for this compiler.

Some attempts have been made at designing a code which could be used by several compilers (such a code is JANUS [29]), but the attempts have met with little success. Using a common intermediate code has the outstanding advantage that, to move the compilers for several languages on to a computer, it would only be necessary to write a single translator from that intermediate code into the particular machine code. Unfortunately, despite this advantage, common codes have won little acceptance, apparently for two reasons. Firstly, however general an intermediate code is, it is not adequate for some particular language (or so the particular implementer will think). That is, even if the designer of the intermediate code allows for all constructs occurring in existing languages, someone someday will come along with a new construct for which the code is highly unsuited. (This problem is similar to that found with compiler-compilers or syntax-directed compilers - one that was designed with ALGOL60 in mind is likely to be hopeless for Algol68.) Even if the language can be compiled into the code, the implementer will usually want extra facilities for which the code does not allow, such as program segmentation in some peculiar manner (cf CLU, above), or optional loading, or fancy debugging aids or storage maps. The second (and, perhaps, overriding) reason is what appears to be an inborn hostility in implementers (including myself) to using something designed by someone else. Whatever other arguments there may be, we can always convince ourselves that our code is in some way better or

more suitable (see below!); in some cases this may be true, particularly if we only wish to compile for a restricted number of computers, since then the code would be tailored to allow for those features which their architectures had in common.

In designing an intermediate code, one must bear in mind precisely what it will be used for. If the code is for only one language, then it will be useful to have one code operation for each 'primitive' operation occurring in the language (though what 'primitive' means requires further discussion). Breaking a language operation into several operations in the intermediate code makes less information available when translating for a particular target machine - this would be unfortunate if the target machine had available an operation corresponding to the original language operation. An example of this occurred in a preliminary version of the Algol68C intermediate code, Zcode. In Algol68C, it is possible to assign any object; the code produced for this is, in general, a store-to-store copy. In preliminary Zcode versions, there was no general store-to-store copy instruction, so, for example, to copy an object of mode STRUCT(INT a,b,c,d,e) we would generate five 'load register; store register' pairs of instructions. Since many machines (such as IBM360) allow store-to-store copy ('MVC' instruction), we were preventing the Zcode translator from recognising that 'MVC' was appropriate (unless the translator were to build our ten instructions back together into the original single operation). This is a particular example of the choice of how 'low-level' or 'high-level' the code should be, that is, how close to the computer hardware the operations should be. The effect of producing code at too low a level is liable to be that information is not available in the intermediate code which could be available to the translator, and this could force the translator to produce inferior machine code. Alternatively, producing a high level intermediate code will result in a very complicated translator, which would be a pity since we require one translator for each target machine. We will discuss below how Algol68C approaches this dilemma.

3.2 Portability in Algol68C

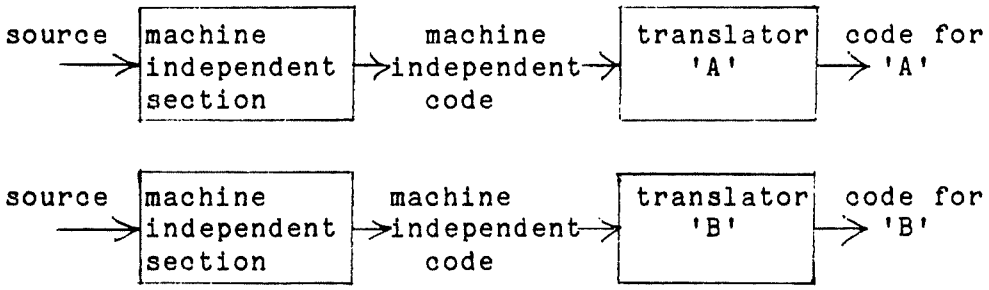
Like other systems, Algol68C uses an intermediate code to achieve portability. The intermediate code, 'Zcode' is defined elsewhere [8], and some discussion about it has been presented before [9]. The code is deliberately very low level. It takes the form of a sequence of instructions and loading directives; the instructions operate on values in registers and in store. Apart from constants, values in the store are in one of three areas: the 'static stack', the 'dynamic stack' and the 'heap'. The last two of these are controlled explicitly by the compiler; the 'static stack' is allocated within a procedure by the compiler, but a procedure call instruction is assumed to allocate

a new stack frame. The current stack frame is assumed in Zcode to be directly addressable, as is the global stack frame; other stack frames are accessed by a 'static chain' mechanism [app.Y]. The program, which is 'pure' (ie. execute-only), is assumed directly addressable. For computers having restricted addressability (eg. IBM360), the translator must take steps to ensure appropriate addressability. The Zcode of a program consists of segments (corresponding to the separately compiled program-segments), which may contain routines. The routines and the procedure-call instruction form a full recursive calling mechanism.

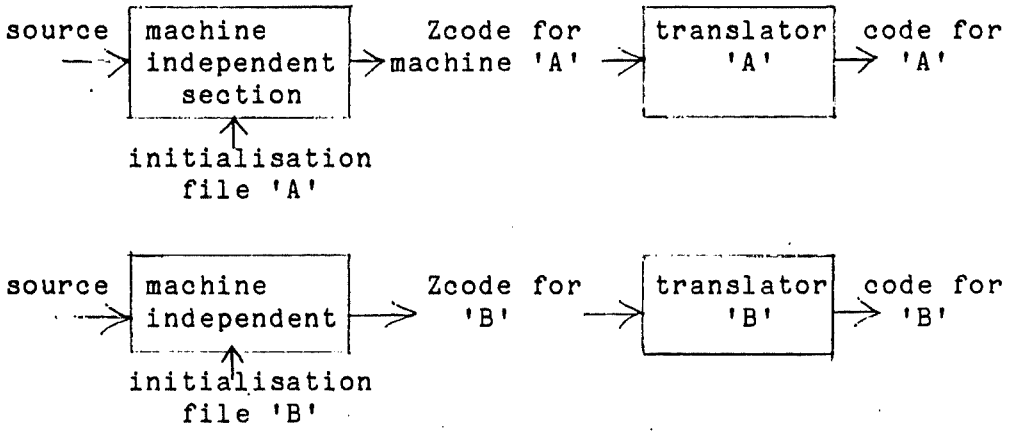
Whereas Algol68C has an infinity of data types, Zcode has only a small number (such as integer, real, character, address, union flag), corresponding fairly closely to the types commonly found in computer hardware; several Zcode types may map on to a single hardware type (for example, integer and character may not be distinguished on many computers). When producing Zcode, the compiler maps Algol68C types on to Zcode ones, so the translator is only concerned with fairly simple objects. For example, STRUCT(REAL re,im) produces Zcode handling a pair of 'real' objects; REF[]INT, whose representation is a 'descriptor' is manipulated in Zcode as a quadruple consisting of an address and three integers ('stride', lower bound, upper bound).

In producing the Zcode from its internal parse tree, phase three of the compiler has three tasks to perform: register allocation, storage allocation, and expressing Algol68C objects and operations (such as loop-clauses) in terms of Zcode objects and operations. It is the intention of the Algol68C system that the Zcode translator should be as simple as possible; to achieve this, Zcode is low level, that is it should correspond as closely as possible to the hardware of the target computer. The distinctive feature of the Algol68C system is that to achieve this intention, Zcode is, in fact, a family of intermediate codes, each differing in the number of registers, their properties, the sizes and alignments of objects in store, and the operations available. Instead of producing a machine independent intermediate code, Algol68C produces a machine dependent code which is as close as possible to the real hardware of the target machine.

This is best illustrated by considering what happens if we want to compile code for two computers, 'A' and 'B'. A conventional system produces its fixed intermediate code, and has two translators:



In the Algol68C system, the machine independent part of the compiler reads data (the 'initialisation file') describing the particular, machine dependent, Zcode to be produced:



By using this technique, we are attempting to avoid the problem that having a low level intermediate code would otherwise induce, namely making decisions when generating the code which are inappropriate for the real hardware (such as having too many, or too few, registers). Another way of looking at the Algol68C system is as follows. In the conventional approach, for the code to be successful, no decisions can be made which might not match the hardware; consequently, in translators for 'similar' hardwares identical or 'similar' algorithms will appear (for example, register allocation). What Algol68C has attempted is to move such algorithms into the machine independent part by parameterising the algorithms and placing the machine dependent parameters in the initialisation file. In this way, we simplify the translators, making the operation of producing code for a new machine much easier, whilst retaining the machine independence of the rest of the compiler.

Ideally, we would take this approach to its logical conclusion, that the translator would be little more than an assembler and the Zcode would correspond to a subset of the machine code of the target machine. Unfortunately, we are not very close to this ideal; in Zcode, we make some assumptions about the basic architecture of the target machine, the most fundamental one being that it is a register machine. Even assuming a register machine, it is not possible to allow for all the baroque sets of registers with which hardware designers endow computers; we can

provide reasonable flexibility, but at some point we are bound to fail. (This is similar to the problems faced by a designer of a general purpose intermediate code - however versatile the design, someone will produce requirements that it cannot meet.) Another problem in this approach, which we have not yet had to face, is that as the parameterisation becomes more sophisticated, the machine dependent initialisation data will become more complicated and manufacturing one could present more of a problem than writing the translator; the initialisation data would become, in effect, a special purpose language.

In subsequent sub-sections I will discuss how, and to what extent, this parameterisation has been achieved.

3.2.1 Storage

Storage in Zcode is either program or data. Program storage presents few problems; Zcode assumes uniform addressability (that is, any program address can be written, as a label, directly in an instruction), so the translator must handle any problems which do arise (such as limited offsets, eg. 4096 bytes on IBM360). Hardware segmentation of program can conveniently be mapped onto groups of Algol68C program-segments. Other sophistications (such as interleaving various types of instructions) are so machine dependent that they seem incapable of parameterisation. To ease machine dependent transformations by the translator, Zcode contains directives specifying 'basic' blocks (as in the FORTRAN specification), which are groups of instructions having a single entry and a single exit.

Data storage is much more complicated. Algol68 requires two distinct areas, a stack (for storage required only during the current block) and a 'heap' (for storage of longer lifetime). Each area can be logically subdivided, and, as described below [app.Y], the Algol68C system assumes that the stack is divided into 'dynamic stack' (for array elements, and storage whose size cannot be determined at compile time) and 'static stack' (for other values: the activation record for each frame, store associated with declarations and for temporary results). A third data area is for constants; Zcode allows this to be interleaved with program, but since it is syntactically distinguished the translator can separate it into, say, a separate hardware segment. All data areas are considered to consist of a sequence of storage 'units'. The size of a storage unit is chosen when writing a translator for a computer, subject to the constraint that incrementing an address by 1 produces the next storage unit (in contiguous areas of data). Storage on the dynamic stack and heap are allocated by the compiler generating a procedure call to one of two routines in the runtime system (the routines are specified in an environment file [B2.2]), with an argument specifying how many storage units are required. Static stack is allocated at compile time; data is accessed from, or written to, there by specifying an offset from a base. The base will be one

of: another, previously calculated, static stack address in a register; the base of the current stack frame; the base of some previous stack frame, held in a register; the base of the global stack frame. A new stack frame is allocated by the procedure call instruction, which specifies at which offset on the current stack frame the new frame should start. The following areas of storage are each assumed to consist of contiguous storage units: the entire static stack; any area allocated by a single call of the dynamic stack or heap allocation routine; constant multiple values. (The assumption takes the form of adding offsets to the base of any such area). Note that this prevents interleaving the dynamic stack between frames of the static stack; this was a deliberate decision and is contrary to the decision made by most other implementers [30] - it is justified elsewhere [app.Y].

In the above description, an 'offset' is a number of storage units, and this number, like the argument to the dynamic stack and heap allocation routines, is calculated at compile time from the machine dependent initialisation data. As mentioned above, Zcode contains a small number of data types (in the current version, precisely: address, integer, real, character, union flag). The compiler decides on a representation of each Algol68 object in terms of these data types. (The representations are mainly obvious, the only non-obvious ones being REF-to-rowof, which is represented by the descriptor of the rowof rather than its address, and a routine, which is represented by its entry address and a pointer to the stack frame which is its environment.) The compiler can then calculate an 'offset' or a 'number of storage units' by considering, for each of the constituent Zcode objects in an Algol68 object, two integers - the 'size' (the number of storage units occupied by the Zcode object) and the 'alignment' (which allows for such constraints as reals having to be on doubleword boundaries). These integer pairs are given, for each Zcode type, in the initialisation data. They are also given for the 'activation record' at the head of each stack frame (which contains at least the return link, static chain and dynamic chain, but may also contain such things as debugging or trace data, dumps of registers used for program addressability, etc.). Further details of how this data is presented are given elsewhere [10].

This storage scheme appears to be very satisfactory. The only desirable facilities that it does not allow for are such things as aligning an object at a given offset from a multiple (eg. at one byte beyond a multiple of four bytes, which would be the alignment required for addresses considered as 3-byte objects on some computers) or using spare bits inside a value. In general, it seems sufficiently acceptable not to require any improvement.

3.2.2 Registers

Zcode represents a register machine, in that some instructions operate on 'registers'. Registers differ from ordinary storage in that they are directly addressed, occupying a special position in some instructions, and may only be used for certain types of value (ordinary store is homogenous - any value can be placed in any of it, subject to size and alignment constraints). They are intended to map directly into the registers of the hardware of the target machine, although the implementer is at liberty to translate them into store locations if this seems more convenient. Zcode 'registers' can be considered to be merely a notational mechanism, but they are intended as actual registers - the implementation dependent data should be able to specify the register properties sufficiently for them to closely resemble (some of) the target machine registers.

There are two classes of Zcode registers: system registers and work registers. The system registers are not parameterised; they are:

0 - constant integer 0
1 - constant integer 1
4 - 'dynamic stack management register' (DSMR)
5 - base of current stack frame
6 - base of global stack frame
7 - immediately before a procedure call, this is set to the value to be used for the new static chain; it has no other use.

Of these, only R4 is explicitly written to by the compiler; the implementer is unlikely to map all of these into actual hardware registers.

The Zcode work registers are where most operations take place, and where results are left (eg. on exit from a call, or from each branch of a conditional). Their number and properties are specified by the initialisation data. The data given is:

the number of work registers;
for each register, which Zcode types can be placed in that register.

To help the translator further with registers, Zcode contains 'R' directives (which specify, at the start of each 'basic block', which registers are in use and which Zcode type each contains) and 'K' directives (which say when the value in a register is no longer required).

The machine independent part of the compiler is responsible for register allocation; in particular, it will attempt to avoid copying registers to store unless strictly necessary, but will write them to store if, for example, a sequence of operations produces more temporary results than there are registers. The

compiler treats registers as a form of cache for the main store - whenever there is a value in a register, a corresponding store address is available, although the value may never in fact be written to store. In practice, the present compiler is not very clever in its register handling, but it is not my purpose to discuss register allocation algorithms here. What I am claiming as an advance is that to produce better register allocation algorithms for every machine for which Algol68C is available, we need only rewrite a single piece of program (phase three of the machine independent part of the compiler), rather than rewriting each translator. This should be contrasted with the situation for the other, higher level intermediate code systems. The fact that phase three of the compiler is at present somewhat badly written does not affect the gains which our approach could produce.

This register parameterisation is clearly not adequate. It is doubtful whether any such system could ever be sufficiently powerful for all hardware, but there are several common features which could be allowed for and which would greatly ease the problems of translating Zcode. Such features include the following. On several machines, using one register requires that some other register be not in use (eg. even-odd pairs of registers on IBM360 series during multiply and divide operations) - this could easily be specified by a bit matrix specifying the inter-relationships of the registers. Another fairly common mechanism in hardware is to allow certain operations only in particular registers (eg. IBM360 multiply (again!), or many operations on CTL modular 1); it is difficult to allow for this, and the most convenient solution at present is for the translator to dynamically map a Zcode register onto varying hardware registers (the Zcode 'R' and 'K' directives make this reasonably straightforward). A more satisfactory solution would be to allow the initialisation data to specify, for each operation, which Zcode registers can be used; this can be done by grouping the operations suitably, and, instead of specifying Zcode types for the registers, to specify operation groups (each group of operations would only place a single Zcode type in a register, so the present information would still be implicitly available). When this technique is used in conjunction with specification of inter-relationships between registers (which allows several Zcode registers to map into a single hardware register), we could then allow quite well for most existing register-oriented hardwares.

The major problem to be surmounted in our register parameterisation, however, is the existence of computer hardware having no registers, that is 'stack machines'. For such computers, all our efforts at register allocation are of no avail - such computers are built so that the hardware perfoms the 'slaving' functions at run-time much better than our attempts at slaving in registers. True, our registers can be mapped into ordinary storage on the computer, and the number of Zcode work registers can be made small (we require a minimum of two for some operations), but this is only a process of minimizing the harm

done by our register allocation - we really need stack-machine operations (such as are used in most other intermediate codes, and certainly in all the high level codes). Although we could persuade the Zcode-generator to produce such instructions, they would not be Zcode. We have to admit that Zcode is not suitable for stack machines, and be content with the gains it gives us for the many register machines.

3.2.3 Operations

One of the problems inherent in generating a low level intermediate code is that by producing several hardware oriented operations for a single language operation, we are removing information, which the translator must painfully reconstruct if its hardware is such that our low level operations are not suitable. It is basic to the design of Zcode that its operations are upon only the basic Zcode data types, so that if the target computer has, for example, hardware to manipulate array descriptors, a translator would find it extremely difficult to take advantage of such hardware. This constraint is as fundamental to the design of Zcode as are its registers. The choice of which data types to recognize in Zcode is, of course, a design decision - we could have included, and often considered including, descriptors as a Zcode type - but as we include more complicated types, while simplifying translators taking advantage of such hardware facilities as descriptors, we complicate all other translators. There is a trade-off between avoiding loss of information (by including more complicated types) and our basic approach of producing simple, low level code. In view of the rarity of hardware support for the more complicated types we feel justified in our present choice of Zcode types. (But we must bear in mind the growing influence of machines with descriptor hardware, such as MU5 and with vector hardware, such as Texas ASC.)

Given some decision as to the data types on which the intermediate code operates, we should clearly have one intermediate code operation available for each language operation on these types. That is, when code generating a language operation, we should only break it into simpler operations to the extent that is required to express it in terms of the intermediate code's data types. Breaking it into simpler operations than this is unnecessary, and may cause unnecessary trouble to the translator. For example, Algol68C allows an expression of the form 'a/:=b' meaning 'a:=a/b'. Code generated for this would be of the form:

```
LOAD  r  b    #register:=b#
SDIV  r  a    #store:=store/register#
```

although equivalent code would be:

```
LOAD  r  a
DIV   r  b    #register:=register/b#
STOR  r  a
```

Clearly, for a machine having a divide-to-store instruction, the former code sequence is preferable, but it is equally clear that for a machine not having such an instruction, the latter sequence is preferable (in order to simplify the translator). If we generate a single operation where the hardware requires several, the translator can do the conversion, but this has two disadvantages: many translators will contain the same algorithm for the conversion (though the conversions in question are so straightforward that this is a minor consideration), and, if the alternative sequence requires registers or storage, then we are reducing the effectiveness of having the machine independent part of the compiler perform register and storage allocation.

The approach which can be adopted to overcome such problems is to include data in the initialisation file (and in practice also in the machine environment) specifying which of alternative code sequences to generate. In particular, for each Zcode operator we specify whether 'op:=' and 'op:=:=' can be generated as a single Zcode operation or whether the alternative sequences should be produced. There are other similar choices (such as the 'CASE' instruction, and the store-to-store copy instruction), although the alternative sequences are not yet implemented.

Thus the Zcode generator takes language operations and expresses them as operations on Zcode types; it will optionally express them as particularly simple operations on Zcode types. This allows adjustment of Zcode to hardwares with restricted operation sets. However, we currently have no way of producing suitable code for hardware supporting higher level types (such as descriptors). One approach to this would be to use similar techniques - include higher level types in Zcode, but allow the machine dependent data to specify that these types should be broken into simpler types. Taking this approach to its logical conclusion, we would be willing to produce code at any level from Algol68 types down to primitive hardware types. This would give us a great deal of flexibility, allowing us to model very closely most register machines. However, the present code-generator does not exploit these techniques very much: the main reason is that, since it does not yet support the complete language, little effort has been expended in allowing selection of code sequences. Nevertheless, I believe (and my belief is supported by the architecture of those register machines which we have examined) that such techniques present little difficulty in the Zcode generator, and greatly ease the problem of producing good target code.

3.3 Summary and Conclusions

It is highly desirable that the compiler we use should be portable. The way this is achieved is by use of an intermediate code, and we can choose either a high level code or a low level one. High level codes merely produce a textual representation of the parse tree (and perhaps of some auxiliary data); low level codes attempt to resemble the architecture of the target computer. High level codes make few assumptions about the target computer, and require complicated translators; low level codes can use much simpler translators at the expense of making machine dependent assumptions. Algol68C generates a very low level code, but attempts to parameterise the machine dependent assumptions so that the intermediate code can resemble the target hardware. This, ideally, would give us the portability of high level codes with the ease of translation of low level codes. This parameterisation occurs in three aspects: registers, storage, and operations.

The major drawback to the Algol68C system is its assumption that the target computer is a register machine, and that in practice the parameterisation techniques have not been developed as fully as we would like. It is possible that a code using stack operations (like OCODE or PCODE) instead of register operations would be more suitable, but there are so many register machines, and good register allocation is so complicated, and to such a large extent machine independent, that the gains of centralizing the register allocation algorithm in the machine independent part of the compiler seem to outweigh the disadvantages. With a little more work, we could parameterise it sufficiently to map onto most register machines with little effort - the difficulties we encounter at present are in areas where the necessary distinctions were not sufficiently clear to us when we were writing the Zcode generator. Work at present in progress on phase three of the compiler should help alleviate these problems.

The key to parameterisation of storage and operations is the way in which Algol68 types are broken down into Zcode types. The choice of Zcode types determines, to a large extent, how 'low level' the intermediate code is. Our aim is that Zcode types should closely resemble target hardware types (we will se later how this can help in some of the problems peculiar to system programming [C2.2]). None of the implementers who have transferred the system to the various target machines has had any complaints about the parameterisation of storage allocation, and none has wanted lower level Zcode types; there are a few computers (MU5, Texas ASC) for which higher level types would be a convenience (in that they would allow production of better machine code), but it seems likely that the lower level types would still be used for initial bootstrapping on to such computers, the higher level types being introduced as a subsequent optimisation.

The compiler has shown itself in practice to be highly portable. On a convenient machine, I can produce a working system in under three months; a more normal figure, to reach a 'production' system would be six man-months. To date, the compiler has been transferred to IBM360 and 370 series, to PDP11/45, to ICL4130 and CAP; it is in the process of being transferred to PDP10, Texas ASC; a cross-compiler also exists for a DEC GT40. The system thus appears to be sufficiently portable for our purposes. Note that 'system' includes any library and run-time system, but these topics are discussed elsewhere [C1]. The work required to port the Algol68C compiler compares quite well with BCPL using OCODE (BCPL, having only one type, has fewer problems.) and PASCAL using PCODE.

More detailed discussion of Zcode and intermediate codes is available elsewhere [9,31], though I do not necessarily agree with their conclusions. I remain convinced that the approach of producing a low level machine dependent code gives us a system very convenient for our purposes, although more work is required to provide a sufficient degree of parameterisation.

Section C: System Programming Facilities

We have seen in previous sections how mechanisms were arranged, and facilities added, which would be of use in achieving our aim of writing a large operating system completely in the language. Features such as separate compilation and compiler portability are extremely useful, if not essential, to achieving our aim. However, there still remain several severe problems which must be overcome before we can claim success. As pointed out earlier, most algorithms in an operating system can be expressed in abstract terms, and using the concepts and facilities of any reasonably versatile general purpose high level language, but we must now address ourselves to the difficulties which arise, peculiar to the task of writing an operating system.

The first class of problems arises from the fact that not all parts of the system are concerned purely with abstract operations on abstract objects. We will occasionally have to write programs performing physical operations on physical objects. These operations and objects will be defined by the hardware of the target machine, as distinct from abstract operations and objects, which are defined by the compiler writer or by the programmer. Note that although there is a hardware representation for abstract objects, the program using such an object is unaware of the nature of the representation. There are hardware-defined objects (eg. the physical layout of capabilities) and operations (eg. initiating an I/O transfer) which our language must allow programs to express. Further, in many circumstances we will have to accept or provide arguments specified in an extra-lingual manner.

A second class of problems exists which tends not to have been generally recognised by writers of other systems. One easy way out of many of the problems of writing the system in a high level language is by use of some form of 'run-time system'. That is, the code produced by the compiler is not capable of running on the bare machine, but only in the environment produced by the run-time system. The run-time system is thus a form of kernel producing an environment in which the compiled code executes. There is no objection to a run-time system as such, but if it is not itself written in the language then we will have failed in our aim of writing the operating system in the language. We will see below the consequences of insisting on writing entirely in the high level language.

A third problem is not strictly part of our original aim of writing the system entirely in a high level language, but is sufficiently closely related to be worthy of attention. We may wish a program to interface with programs in other languages using interfaces defined by those other languages. An alternative expression of this is that we may require the ability to specify to the compiler the environment in which the program will

run. Note that if we could solve this problem entirely, we would have a system capable of producing code for imbedding in machine code programs - we could take an existing system and replace parts of it by code produced from program written in a high level language.

A language system which solved all three of the above problems would clearly be an extremely useful tool. Not only could it be used for writing an entire operating system without recourse to machine code, but it could also produce code to run in harmony with code written in other languages and it could be used to modify existing machine code systems. I am not aware of any existing system which has solved all these problems, but we will see below that the solutions do not appear to be impossible.

1. Run-time System

An implementation of a general purpose high level language normally includes a run-time system. This system will typically be concerned with storage management, input-output and run-time debugging facilities. It may also include such facilities as overlay mechanisms and dynamic program loading, and will probably cater for interrupts provoked by user errors. Such run-time systems can become extremely large (consider FORTRAN I/O packages or co-ordination and storage management in SIMULA or CPASCAL), and can require a considerable programming effort. They are almost inevitably written in machine code, and linking of programs to routines in the run-time system is achieved by mechanisms which are distinct from those used for linking to other parts of the program. The run-time system is treated specially by the compiler - normally the names of entry points, or label numbers, or even entry addresses are built into the compiler. All code produced by such a compiler assumes the availability of the run-time system.

Such an arrangement is not acceptable for our aim of writing the operating system entirely in a high level language - we must insist on being able to write the run-time system in the language, for otherwise we have only postponed the problem; writing the machine code run-time system has all the pitfalls we are trying to avoid by use of a high level language. Note that even the excuse that there is only one run-time system and several programs has little force, since we are likely to want only one operating system for each machine. In any case, the run-time system is likely to be subject to considerable modification as the operating system develops. Additionally, on the CAP, different run-time systems are appropriate for different programs (eg. the co-ordinator and the command program). In particular, for several parts of the operating system it would be most inappropriate to attempt to implement the sophisticated I/O and storage management required for other parts. A second difficulty of a compiler relying on a run-time system is the danger that the code produced by such a system cannot execute without it and so we would have difficulty in producing code for imbedding in other environments (such as when called from another language, or imbedded in existing machine code). Thus we must retain the ability to compile code to run in very simple environments.

It will be useful, then, to investigate the reasons for implementers wishing to have a run-time system. Although I will describe places in Algol68 where one is tempted to resort to such a privileged run-time system, the comments are also applicable to almost every other language, in greater or lesser degree.

One of the most drastic examples of a facility which would normally use a run-time system is the Algol68 construct called a 'parallel clause'. This is a facility allowing a programmer to specify a set of actions to be performed asynchronously, with a 'semaphore' facility for synchronisation purposes. The units of

a parallel clause correspond fairly closely to 'asynchronous processes'; the overall effect is similar in its end result to monitors in concurrent PASCAL [28], and to some features of SIMULA [25], although the Algol68 facility matches neither of the others in their elegance. For our present purposes, however, any such facility is a drawback, rather than an advantage, if it is considered as a primitive construct in the language. This is because its implementation requires, in effect, a process co-ordinator to be available and to be called implicitly by the compiled code. That is, the process co-ordinator would be part of the run-time system. The objection to the facility would be removed if parallel clauses were not primitive, ie. if they were defined in terms of simpler constructs. Such a definition would include a co-ordinator, but it would then be being called as sub-program by the normal language facilities; it would merely be a library routine and it, and parallel clauses, would no longer be part of the syntax and semantics of the language. The objection to parallel clauses, thus, is not to the facility (which might possibly be useful), but to the fact that they have to be treated specially, as part of the language - they should be something which the programmer is at liberty to define, not something that the language implementer must provide. As a word of warning, although such high level facilities have a place in some languages, their provision in a general purpose language (and particularly in a language for writing operating systems including process co-ordinators) seems dangerously far down the road which leads to the program being:

```
UNTIL finished DO work
```

and the run-time system being correspondingly complicated. Parallel clauses are not available in Algol68C, nor in most other implementations of Algol68.

A common cause of code in the run-time system is input and output ('transput', to use one of the more successful parts of Algol68 jargon). A lot of the transput code is not in fact 'run-time system' in our sense, since it is merely a set of subroutines called in the normal way; there are however some situations where the compiler has to react specially. Although most of our system programs will be unlikely (and probably unable!) to indulge in transput, it will be instructive to look at some parts of it. The most strikingly complicated part of transput is 'formats'. Analogous constructs exist in many other languages (FORTRAN, BCPL, PL/1, COBOL), although they reach a peak of complexity in Algol68. Formats virtually constitute a mini-language imbedded in Algol68 to describe certain transput operations. Although their complexity has perturbed many implementers, our objection to them here is the fact that they implicitly call various routines. This implicitness indicates that the routines are in some way treated specially by the compiler, and are not ordinary programs written by ordinary programmers.

Thus, once we start considering the run-time system as ordinary program compiled in the normal way, we can begin to see features such as formats as indicating situations where the language designer has found the normal interface mechanisms (ie. procedure calls etc.) to be inadequate. However, the designers response was not to make the interface mechanisms more powerful, but to bridge the gap by adding ad hoc syntax for his special purposes. This is strikingly illustrated by the section of the Algol68 report defining transput. This section is written very largely in the form of a program (only 'form of', since it is written in such a style that one would not consider executing it in a practical system), but at several critical points it lapses into ordinary English. These points are places where the interface mechanisms were inadequate. Examples are formats, and 'straightening' [1]; a less obvious example is 'print', whose argument is a vector of objects, each of which can take any of an infinite number of data types - here the interface is defined in the language, except that the data type of the argument could not be defined in the language. It would seem that if the designer found the interfaces inadequate, then a programmer would be justified in bemoaning their inadequacies.

The ideal solution to these problems would be to redesign those parts of the language, and to add extra interfacing mechanisms, so that they need no longer be treated specially by the compiler. We could then write those routines, and link to them in the normal ways. Such redesign is clearly not difficult once we have recognised the need for it. However, the solution I adopted for Algol68C is rather less than ideal: since incompatible changes to the language accepted by the main Algol68C compiler were considered undesirable, a method was required which would allow such parts of the run-time system to be written in Algol68C, while not changing the constructs (such as 'print') which use them. The general technique was to make certain identifiers known to the compiler (by placing them in the initialisation file), then when code generating the special constructs, the code generates normal procedure calls to those identifiers. To avoid those identifiers being 'reserved', the compiler only considers definitions of them in separately compiled segments marked as being part of the 'prelude'. The mechanism (for 'print' only) is defined more precisely in the Algol68C reference manual [7]. This mechanism is not entirely satisfactory, since it treats the constructs specially, but it does allow us to compile the routines of the run-time system in the normal way; the compiler has built into it only the names of the routines. Although the mechanism has at present only been used for 'print', it can clearly also be used for calling such things as the storage allocation routines (and thence, for example, the garbage collector, if we had one).

There are parts of the run-time system which present difficulties when we try to write them in our high level language, for reasons other than interface problems. Examples of such parts are initialisation and termination code, and areas

which require access to data generated by the compiler (such as storage maps for the garbage collector, and routine names for postmortem facilities). All such difficulties appear to come into the classes of problems mentioned earlier, namely writing code to access externally defined objects, and allowing for the environment in which the code is to execute. The same remarks apply to parts of the run-time system concerned with hardware defined operations. Such problems are discussed in a later section [C2.2].

Apart from register initialisation and handling of entry and exit points from the program, the compiler, by using the techniques outlined above, need have no knowledge of, or dependence on, a run-time system. A complete source program, in this view, consists of the user's source plus the source of the run-time system. The run-time system is typically pre-compiled and shared, but this is a facility available equally to parts of the user's program [B2.2]. There will be certain constructs which, when used, imply a call to a routine with a certain identifier, but this routine is defined in the normal way - it is in no way special, and in principle could even use, in a recursive manner, the facility it makes available.

This gives us considerable freedom in providing a run-time system. We can readily provide run-time systems for differing machines or environments without having to inform the compiler; programs can be written to rely on as much or as little run-time system as desired. Indeed, the run-time system does not need to be considered as part of the language system; a standard one (or a selection of them) would be provided as default by the language implementor, but the user (particularly if he is a system programmer) would be under no compulsion to use the default. As part of the Algol68C separate compilation mechanism, the source text of a segment nominates, in its heading, what ENVIRON it is to be compiled in; this, in the first segment of the user's text indicates what run-time system is desired - if omitted, the default one providing the standard Algol68 system is assumed. Thus the code generated by the compiler for a piece of source text makes no assumptions about the environment in which this text is to execute, other than the information specified in the appropriate environment file. The initial environment file specifies the minimal (ideally empty [C3.2]) environment.

1.1 The CAP Run-time System

In developing the CAP operating system, there has been need for run-time systems of differing complexity for the different constituent programs of the system, and it was considered essential that machine code should be avoided if at all possible. These requirements have been met in the Algol68C system by taking advantage of the compiler's independence from the run-time

system. I have been able to produce run-time systems for the various situations, pre-compiled and shared between programs. Many parts of the run-time system are machine independent, and can be made to run on another machine - this has been done for a PDP11 by I.Walker. The run-time system is treated in no way specially by the compiler, and is compiled in exactly the same way as segments of a normal user program (but less often). The run-time system has considerably developed and altered as the operating system developed, and at one time was available under two operating systems - none of this has caused any serious difficulty.

The run-time system consists of five Algol68C segments, known as MC, MIN, SER, SYS, USE. (A sixth segment, MATH, providing various mathematical functions has recently been added but need not concern us here.) Each of these five segments uses the environment provided by the preceding one, except MC which uses an empty environment. The programs of the operating system use the environment provided by (MC + MIN) or by (MC + MIN + SER) or by all five. The default environment, providing the full facilities of Algol68, uses all five (now six).

The segment MC performs those actions which are necessary before code generated by the compiler can be executed. It is the residue of traditional run-time system whose existence is assumed by the compiler, and is necessarily written in a mixture of Zcode and machine code, rather than in Algol68. Ideally, MC would not be necessary, but at present the best we can do is try to minimise it. Future possibilities for this are considered later [C3.2]. MC can be considered as converting the bare machine into a virtual machine on which compiled code can execute. The actions performed are few and simple. At each entry point (there are three - one defined by hardware, two by software) the following registers are loaded or restored from a previous dump:

Register 1	-	a constant 1, assumed by the translator
Register 11	-	the 'dsma' [app.Y]
Register 12	-	the global stack base
Register 14	-	the current stack frame base

From the main entry point (word 0 of segment P0), control is then transferred either to the start of the compiled code, or as if exiting from the 'return' co-routine (described later [C3.1]). A second entry point is word 1, to which control is transferred by software if a trap (program error) has occurred - from here a variable subroutine, 'run-time error' is called; this subroutine is compiled code, by default in MIN, SER or USE but optionally provided by the user program. The third entry point is used by software to interrupt a program to indicate the occurrence of a 'quit' signal; depending on a user-settable flag this causes either immediate resumption of the interrupted program, or a call of a variable routine 'attention routine' which can be in USE or provided by the user. There are two possible exits - either the end of the compiled code is reached, in which case MC executes a

'return' instruction [A2.2], or the 'return' co-routine is called, in which case MC preserves enough information (ie. registers 11 and 14) to allow resumption on next entry.

Additionally, at present MC contains the routines for allocation of heap and dynamic stack storage. It would be preferable if these were compiled code in MIN, but unfortunately the present compiler links to them using a label number nominated in the environment files rather than using the identifier technique described above. This undesirable situation could clearly easily be resolved by using this technique, but this has not yet been done.

The total size of MC is about 220 orders. The remainder of the run-time system is compiled as normal Algol68C segments, treated in no way specially. Where these segments are concerned with operations or objects not available in Algol68, 'CODE' sections are used to imbed explicit Zcode; where the objects or operations cannot be expressed in Zcode, explicit machine orders are imbedded.

The segment known as MIN is compiled using an environment file, 'MCENV', which corresponds to the segment MC. This environment contains only the program entry and exit label numbers, the label numbers for the storage allocation routines, and data required by the compiler for the in-line operators; it was originally manufactured by hand, but could readily be used for other systems or other machines. MIN is concerned mainly with operations which require no interaction with the operating system. These include such items as string concatenation, selection and comparisons, and facilities such as the fast MOVE order for bulk data transfers. In MIN we meet also the major example on the CAP of a data object defined by the hardware with which programmers will be concerned. This is, of course, the capability. In the absence of advanced techniques for defining types and operations such as discussed later, and of language and compiler changes such as have been proposed for handling capabilities [41], the following facilities have been provided.

A new Algol68 mode, called SLOT has been defined. This is represented in Zcode and machine code as the address of a capability. Note that SLOT cannot be of the form REF CAP, since de-referencing would have to cause a capability to be brought into ordinary store, which the hardware does not allow. SLOT objects, being merely addresses, can be stored, copied and so on, in ordinary storage with no special care. However, various routines are provided in MIN to perform special operations on them. For example, 'MOVECAP', 'INDINF', 'SEGINF', and 'ENTER' perform the corresponding machine instructions [A2.2]; note that 'ENTER' takes special care to preserve and restore such items as the stack pointer. Also, routines are available to allocate store in a workspace capability segment (yielding the corresponding SLOT value) and to explicitly relinquish such workspace. A problem arises when the programmer knows that some

SLOT value corresponds to a segment containing data in a certain format (yielded by calling some protected procedure, or given as argument to his protected procedure, or linked in when his protected procedure was created). Then we must provide him with some means of accessing the data from Algol68. This, again, is a question of accessing an externally defined object, and is the subject of discussion later [C2.2]; the solution at present is a routine taking a SLOT and yielding a REF[]INT referring to the corresponding segment. This, albeit imperfect, solution has proved sufficient for all requirements in the operating system. Clearly, a more general solution is desirable, but it is neither necessary nor urgent.

The run-time system used by the co-ordinator consists of MC + MIN. The next segment, SER, defines subroutines for exercising each possible entry to the co-ordinator. These entries are almost all constrained by the co-ordinator such that they must be made via a protected procedure (available in each sub-process) known as ECPROC, so SER routines mainly use the 'enter' routine provided by MIN to enter ECPROC. The run-time system formed by MC + MIN + SER is that used by the vast majority of the protected procedures of the operating system. The SER segment is based on routines originally written by C.J.Slinn [32].

The fourth segment, SYS, is concerned with the various public interfaces made available by the operating system. It contains numerous routines, typically one for each service provided by the operating system; these routines are mainly implemented by calls on the 'enter' subroutine - their purpose is to arrange the names, arguments and results in a convenient and mnemonic form. With the facilities thus provided, ordinary user programs rarely contain explicit calls on 'enter', using instead the appropriate service subroutines - the user does not need to know the complexities of any of the system interfaces.

The segment known as USE is mainly concerned with providing the definitions to implement the Algol68C transput system [7]. This is implemented by various pieces of relatively straightforward code; these implement the routines required by the Algol68C transput by calls on the service routines provided by SYS. These routines in USE are machine and system independent, requiring only a small number of primitive service routines, and they have been used as a basis for implementing the Algol68C transput elsewhere. Other facilities implemented in USE are placed in that segment because they use the transput. The most important of these are for handling faults (traps), and attentions (such as 'quit' signal from a console). As mentioned above, MC causes a procedure call on a variable routine ('run-time error' or 'attention routine' respectively) when these events happen. The MIN and SER segments assign to 'run-time error' a routine which terminates the program (by 'GOTO stop'), as this is the most appropriate default for users of those segments, and sets a flag causing MC to resume the program if an attention occurs (since almost all system protected procedures

wish, by default, to ignore attentions). In USE, these defaults are replaced by new ones by further assignment to these variables. The effect of the new default is that on all faults (except one special one) a message appropriate to the fault is printed (this message is generated on request by a protected procedure of the operating system), and then a 'backtrace' listing the routines active on the stack is printed. Producing this backtrace involves another example of the use of an extra-lingual data object, namely one which gives access to information in the activation records of the recursive stack frames. Fortunately (but deliberately), this object can be described by an Algol68 mode:

```
MODE FRAME = REF STRUCT (FRAME dynchain, INT link,
                         FRAME statchain, REF STRING name)
```

Thus the 'backtrace' routine uses a CODE section only to obtain an initial FRAME object referring to the current frame and is written thereafter in ordinary Algol68.

The segment MATH has recently been added to the run-time system to provide a package of floating point mathematical routines. The package was written by P. Kemp and was intended to be transportable [33]; I transported it from an IBM370 (with 57-bit sign-and-modulus mantissa, 7-bit base-16 exponent and truncation of excess bits) to the CAP (with 24-bit two's-complement mantissa, 8-bit base-2 exponent and unbiased rounding) in less than one day with no difficulty.

The run-time system at present contains no provision for a garbage collector. This is a reflection of the absence of garbage collection facilities in the compiler. However, it would appear to be fairly straightforward to add these facilities. The garbage collection code would be part of the heap storage allocation routine (which should be in MIN rather than MC). To operate, the garbage collector needs access to a storage map and mode templates generated by the compiler; with care, there is no reason why these should not be valid Algol68 data objects (as has been done at present with stack frames). The garbage collector also needs marker bits, which on the CAP would probably have to be a bit map, and needs a subroutine which, given an address, would indicate the corresponding bit. Having determined which parts of the heap are accessible, the remainder is free for allocation. The situation is complicated slightly by the presence of objects allocated not by the language system, but by other protected procedures. These objects all occur in segments other than the stack and heap. We must either manufacture (in some way) templates for them, or take advantage of the assertion (which could not be verified) that for the correct operation of the current protected procedure, objects obtained from other protected procedures must not contain inter-segment references. This assertion must be satisfied, because external protected procedures cannot validly know the arrangement of the address space of this protected procedure, and to ensure correct opera-

tion the object program must not contain arguments containing addresses without checking their validity. This effectively means that the architecture is such that data objects passed as argument must not contain addresses, so the garbage collector problem caused is not serious.

Implementing the run-time system as far as possible in Algol68, with minimal CODE sections, has proved very satisfactory during the development of the system and has allowed a large number of facilities to be made available with little difficulty and few errors. The fact that the compiler makes no assumptions about the run-time system has provided great flexibility, including provision of differing systems, and ease of modification - sometimes by those writing the operating system, with no knowledge of the compiler. The run-time system contains only 26 explicit machine instructions, and provides sufficient subroutines for handling hardware and externally defined objects that very few parts of the operating system (mainly the peripheral handlers) have found need to use CODE sections. The run-time system now amounts to 5887 compiled orders and provides 325 definitions.

2. Hardware Objects and Operations

We have been assuming until now that the language being used for our system is an ordinary, general purpose high level language, typically Algol68, but we must now consider those situations where the ordinary facilities prove inadequate. None of these situations is unique to operating system programs; they can occur on the CAP in totally unprivileged user programs.

The first such problem could be termed 'storage allocation': a program is liable to find that the storage management regimes provided by the standard language (in Algol68, a stack and a garbage-collected area of global storage), are inadequate for its purposes. On the CAP, a programmer might appeal to the operating system for some extra storage for various reasons: allocating storage in separate segments might produce a better pattern of virtual memory traffic for his program, or he might be taking advantage of facilities allowing him to change the sizes of the segments independently from each other. In a typeless language, such as BCPL, this is no problem: the programmer accesses each segment as a vector of the basic data object ('words' in BCPL). However, in any strongly typed language, such as Algol68, we have a serious problem: the program must be able to reference parts of its storage as a particular data type. This implies some form of type transfer from the unallocated store to a specified data type. This has been picturesquely termed the 'white store' problem.

The second problem we will have to consider is the most difficult. Our language includes a set of abstract data types (such as 'integer' and 'boolean') operations on them, and presumably mechanisms for specifying and defining operations on new abstract data types built up from the old ones (such as Algol68 'mode definitions', 'structured values', etc.). However, our hardware and other programs will inevitably have available, and hence some programmer will inevitably want to use, data types which cannot be represented in terms of the types already built into the language and the available constructions. For such 'hardware types' we will need to investigate more powerful type construction techniques than are presently available in the language; as will be seen later, examples exist which defy most existing techniques.

A third problem arises when we consider the set of operations available in our standard high level language. The contents of this set will depend on the generosity of the language designer, but it will certainly not be sufficient for all purposes. For example, it will not allow the author of the teletype-driving program to write his 'read character' subroutine. It must be borne in mind here that we are considering a system written entirely in the language - there is no let-out by putting the subroutine into some 'run-time system'. Other examples will occur even in user programs - all I/O actions, for example, will reduce to some operation not expressible in the language.

Care must be taken not to over-rate the importance of hardware-defined types and operations. The situations in which they occur can be kept to a minimum, and they can be handled by sordid techniques applied in centralized, carefully written and thoroughly debugged libraries. Good solutions to the above problems are desirable and the problems are worth investigating, but the system can survive without the good solutions. Indeed, at the present time, the system available on the CAP contains no elegant solutions to these problems, although an implementation of a solution to the 'white store' problem is becoming urgent; the CAP nevertheless contains a complete working operating system written in Algol68C.

2.1 Storage Allocation

In Algol68, the programmer can explicitly allocate storage (variables) in two ways: 'local generators' and 'heap generators'. The former is stack-like - the storage is considered free at the end of the block in which it was allocated, and there are language restrictions preventing attempts to access the store after this. The latter allocates 'global' storage, which is not freed while there are references to it, with no restrictions as to how these references are passed around. In practice, the only way in which this global storage can be found to be free is by a garbage collector. Other storage is allocated implicitly from time to time, for such purposes as storing subroutine links and temporary anonymous results; all such implicitly allocated storage can be freed in a stack-like manner. Situations arise in which the facilities offered by these regimes are inadequate; I have already mentioned the case of a programmer wanting to segment his storage (for a number of reasons), and other difficulties can be envisaged such as wanting to use user-supplied allocation routines, wanting to allow explicit relinquishment of storage or storage control by use counts. (Invokation of a garbage-collector, apart from being expensive, can be embarassing in real-time situations such as a peripheral driving process.)

It is not difficult to propose mechanisms which allow such extra facilities, but they tend to also introduce pitfalls for the programmer. The following mechanism was proposed for Algol68C [34], but has not yet been implemented, because of doubts about its suitability. A new mode, 'ADDR' is introduced, intended to correspond to the address of basic units of storage of the computer, and a mode 'AREA' intended to be a block of storage (typically on the CAP, a segment). A dyadic operator '!' is defined, of mode PROC(AREA,INT)ADDR which yields the ADDR for a given cell in a given area. A new form of generator is allowed, with syntax:

heap-symbol, primary, actual-declarer.

where the required mode of the primary is

```
STRUCT( PROC(AREA,INT,INT)ADDR x, AREA y )
```

The programmer might then proceed as follows:

```
AREA seg = getseg(1000) #ask for new segment#;
INT ptr := 0;
PROC get = (AREA a, INT size, align)ADDR:
   BEGIN
      INT n = ptr align * align + size;
      ptr := n;
      a ! n
   END;

REF[]INT v = HEAP(get, seg)[0:100]INT
```

The heap-generator would cause 'get' to be called with arguments: 'seg', alignment for []INT, number of cells needed for the object. The user's routine determines which cells to allocate, yielding a suitable ADDR, and this is yielded in the REF[]INT value of the generator. It should be noted that the operation of punning the 'white store' into the appropriate data type is performed as part of the generator; it is not directly available to the programmer. This is deliberate, since a 'punning' mechanism is a dangerous tool in the wrong hands; it is a necessary tool for storage allocation, but we should restrain it as much as possible. However, we have no way of preventing the user writing his allocation routine in such a way as to allocate the same store twice. Since such over-allocation, apart from being unreasonable programming practice, would also potentially confuse any garbage-collector it must be forbidden (which is easy to implement - for example by keeping a bit-map of the AREA). Similar schemes, differing in detail, have been proposed elsewhere [35].

Arrangements for relinquishment of storage are less simple, because of the diversity of techniques: garbage-collection, explicit-relinquish, use-counts. Garbage-collection presents no difficulties, provided we have prevented the user over-allocation, for example by allowing the user to explicitly invoke it giving an AREA and a routine of mode PROC(INT,INT)VOID to be called with the size and offset of blocks found to be free. Explicit relinquish is similarly simple. Use-count schemes can be achieved by calling a user routine whenever a reference into the AREA is replicated or destroyed, but the expense is likely to be prohibitive.

As can be seen, then, a solution to the 'white store' problem, albeit an inelegant one, is not difficult to produce. It is likely that I will shortly implement some variant of the above scheme for the CAP, since the present facility (which treats all segments as REF[]INT) is becoming increasingly inadequate.

2.2 New Data-types

What we wish to achieve is to allow the programmer to express in our system programming language, operations on objects which are not included in those provided by the original high level language, in order to be able to manipulate objects defined by the hardware or by other programs (possibly written in other languages). Many languages already provide type extension mechanisms allowing the construction of 'new' data types - for example the constructors STRUCT(...), UNION(...), REF..., [,,]... in Algol68. However, all these are ways of constructing new types by composition of existing types. There is no way in such languages to augment the set of basic data types. Since there may exist hardware types not definable in terms of the given basic types, it is useful to investigate more fundamental ways of defining data types. It should be noted that identical arguments apply to the set of type-constructors - in the most general solution to our problem, we should also be able to define new constructors.

Topics such as this bring to mind the various extensible languages, but in practice these always construct new data types from existing ones, since this is entirely sufficient for purposes other than ours. However, a mechanism has been outlined by P.Jorrand [36] which aims precisely at allowing the creation of new basic types. I will give below an outline mechanism which is a transformation of his techniques arrived at by considering them in the context of a system such as the CAP Algol68C; the reader is strongly recommended to read also Jorrand's papers. When considering type extension, it would be inappropriate to omit the recent developments in languages oriented to type abstraction, a typical example of which is Liskov and Jones CLU [26]. However, we should be able to achieve sufficiently powerful mechanisms without the radical re-design of the high level language that introduction of CLU techniques would entail.

Let us first consider Jorrand's scheme. Having pointed out that other systems do not allow the definition of basic types, he considers what a data type entails. A type is a set of objects, together with internal and possibly external representations of the objects, and a set of operations that can be performed on the objects. For example, in Algol68, the type INT specifies a set of objects, with external representations such as the character sequence '100' and internal representations such as the bit patterns for fixed point numbers in some particular hardware. A set of operations is available, such as addition, subtraction, copying. To define new types, then, requires defining sets of objects, defining the representations of particular objects, and defining operations on the objects - more precisely, operations on the internal representations of the objects. To enable the user to make such definitions, Jorrand first postulates a 'base

language'. The base language is assumed to have a primitive binding mechanism (such as in lambda calculus), and is assumed to be capable of specifying any possible operation on any internal representation. The user can then, for example, define 'C' to be a set of objects (with, as yet, unspecified membership); he can define an object 'A' with internal representation given by some expression of the base language 'exp' to be a member of C; he can define operations on members of C by specifying in the base language the corresponding operations on their representations. Thus, in Jorrand's notation:

```
DEF(C,CLASS)
DEF(A,AS(C,exp))
DEF(P, PROC(C,D,lambda-expression))
APPLY(P,A)
```

Jorrand also describes definitions of set relations between types (such as inclusion, cartesian product, complement), and implicit type conversions (coercions, in Algol68 terminology) but, although these would be useful in an extensible language, they need not concern us here.

Let us now consider Jorrand's scheme in relation to the Algol68C system. The critical question is what to use as the base language. We need a language that allows us to express any operation or representation that can occur in our hardware; this is a severe demand, to which ultimately there can be only one answer - machine code. However, with care we can present this in a palatable form. Algol68 is a language defined in terms of abstract objects and operations on them; it is the task of the compiler to convert these into operations on their internal representations. Zcode, the intermediate code produced by the Algol68C system, is thus a language for expressing operations on internal representations of objects. It is the task of the Zcode translator to convert operations on Zcode objects into operations on hardware objects - the design of Zcode is intended to be such that this transformation is straightforward, with simple 1-1 mappings from Zcode objects into hardware ones. We can thus see three levels of language - Algol68, Zcode, machine code - each of which has its primitive types. If the programmer is to introduce a new primitive Algol68 type he will have to specify the Zcode for its objects and operations, just as he would give the base language expressions in Jorrand's scheme. Similarly, circumstances may arise where the Zcode types and operations are inadequate, and he will have to give machine code specifications of them. Thus, the infamous 'code section' appears as a necessary tool for introducing new primitive types. However, if we bear in mind Jorrand's scheme, we should be able to include code sections in a controlled and well-structured manner.

Several modern languages have been based upon mechanisms for defining abstract types. The constructs used generally resemble the Simula 'class' mechanism [25], but are more specifically oriented to specifying all the properties of a type, rather than

just grouping and invoking sets of variables and operations. Constructs have been proposed for adding such grouping mechanisms to Algol68 [B2.4], but they fall far short of complete type definitions. The language CLU, designed by Liskov at MIT is based on the ideas of abstract types, and contains powerful type definition mechanisms. Consideration of the CLU mechanisms will be useful for designing a primitive type definition mechanism. CLU is based not on conventional block structure, but on a sequence of independently compiled definitions; these definitions are either of procedures or of 'clusters'. CLU procedures are straightforward, taking items of specified data types as argument, and performing manipulations on the arguments using whatever operations or procedures may be available. A cluster defines a type, and operations on the type; in particular, it defines the representation of objects in that type. An instance of the type specified by a cluster can be created by invoking the cluster with arguments as required by the cluster. A cluster makes available to users of the cluster a set of operations, many of which will take an item of the type defined by the cluster as argument. For example, if we have defined a cluster called 'stack' with operations called 'push' and 'pop', and if we defined an item called 'a' as an instance of a stack, we could invoke push or pop as:

```
stack.push(a,item);
itemvar:=stack.pop(a)
```

The crucial difference between operations defined inside a cluster and procedures defined outside it is that the operations have available to them the components of the representation of the type. No information about the representation is available externally. When an operation takes an object of the type as argument, it can specify that inside the operation the object is to be available as its representation. For example, if our stack is represented as an array and a pointer, then inside the operation 'push', the array and pointer of the particular stack object given as argument will be available. Thus the cluster might be defined as follows. (The syntax should be self-explanatory, and is similar to PASCAL.)

```
stack: CLUSTER (size:INT)
       IS push, pop #externally available operations#;
          REP(sz:INT) = ( tp:INT;
                          stk:ARRAY[1..sz] OF val
                        );

          CREATE #called by system when creating a 'stack'#
             s: REP(size)
             s.tp := 0;
             RETURN s;
             END;

          push: OPERATION(s:REP, v:val);
             s.tp := s.tp + 1;
             s.stk[s.tp] := v;
             RETURN;
             END;

          pop: OPERATION(s:REP)
             s.tp := s.tp - 1;
             RETURN s.stk[s.tp+1];
             END;

       END #stack#;
```

For the general handling of abstract types, I find schemes based more closely on SIMULA (where the representation is the set of variables defined in the body of the class) more elegant, but for our present purposes, the explicit REP of CLU is more suitable. We could clearly use the formalism of clusters for the definition of new primitive types, by having expressions of the base language for the bodies of the representation function and of the operations.

2.3 Conclusions

It is clear that there is a need for some language extensions along the lines outlined above to enable the programmer to handle such situations. One must bear in mind in proposing any such extensions the bewildering variety of data objects and operations with which hardware designers and implementers of other language systems can present us. A few examples should emphasise this point. A well-known example is the difficulty of passing Algol68 multiple values (which can be scattered around the storage in a sparse manner) to FORTRAN without copying. This is a serious difficulty in providing inter-language library calls. An example occurring on the CAP is the handling of capabilities. Although these objects can be manipulated by the programmer, they cannot reside in ordinary storage and can only be manipulated by special instructions - we must always use a long spoon when handling capabilities. A simpler example is in the execution of input or

output instructions - here there is no possible alternative to having explicit machine code instructions. Even within a single language program we have encountered situations where we must handle extra-lingual objects, such as mode templates and storage maps.

For each example produced, tidy ad hoc solutions can be found [41, C2.1], each requiring its own language changes. It would be much preferable to have a general mechanism for the description of such objects and operations, along the lines outlined above. The 'base language' could conveniently be developed from the Algol68C facilities for imbedding Zcode in a program (and CAP Algol68C facilities for imbedding machine code in Zcode), since Zcode (and the imbedded machine code) is concerned with operations on the primitive objects from which all objects must be constructed. In the present CAP Algol68C system, imbedded Zcode has proved to be a sufficient, but sometimes inconvenient, tool for all such situations. A more general mechanism would make it much more convenient, and much safer, to use.

3. The 'Program'

3.1 Program Structure

At an early stage in the design of the language system, a decision has to be taken as to how the Algol68 concept of 'program' maps on to the CAP architecture. There are various possibilities: the complete operating system, or a single process, or a number (possibly one) of protected procedures, or a single program segment of a protected procedure. Having the complete system as a single program was rapidly discarded, for a number of reasons. It would require language or implementation mechanisms for describing process hierarchy, process structure and protection structure. It would considerably blur the distinction between operating system and language, and would almost inevitably hide the causes and effects of operating system design choices. It would prevent the facility, thought to be desirable, of allowing the system to be written in more than one language. On the CAP, where there is no form of supervisor state or privileged mode (other than that caused by the relationship of a coordinator to its junior processes), there is no precise distinction between 'operating system' and 'ordinary' programs; as far as possible, system functions are hived off into protected procedures having minimum privilege, which run as ordinary user programs. Accordingly, it is difficult even to conceive of the operating system as a single program. Precisely the same arguments apply against a process being a single program.

We thus always consider, in any language on CAP, a 'program' to be a set of one or more protected procedures. Note that we have therefore reduced the possible pitfall of imbedding the operating system in the language implementation to a very shallow one. Use of several protected procedures for a single program could be convenient for several languages (e.g. CLU), but not for Algol68. Having a conventional block structure, with the ability to access non-local identifiers, does not correspond to protected procedures with their water-tight encapsulation and narrow, explicit interfaces. Further, the notion of protected procedure was designed to facilitate enforced checking of aspects of program execution, whereas Algol68 is designed in such a way that very few runtime checks are needed, and those which are needed are more amenable to the checks provided by segmentation. The expense of protected procedure entry and exit cannot be justified for intra-program calls in Algol68, only for inter-program calls. It follows from this remark that the idea of allowing multiple Algol68 programs inside a protected procedure was discarded.

An Algol68 'program', then, is to be considered as a complete single CAP protected procedure. Segmentation in the hardware sense is mapped onto the Algol68C separate compilation mechanism: sets of separately compiled segments may be placed in separate hardware segments, to facilitate sharing (and control of virtual memory traffic where this occurs) - the complete operating system

contains only one copy of each segment of the Algol68C runtime system.

A choice was available as to how to provide the stack workspace used by a program. An early scheme was that when a protected procedure is called, the calling program should pass as one argument a sub-segment corresponding to the unused portion of its stack (or other workspace). The alternative was that each protected procedure should contain its own private workspace. The former approach seemed desirable on grounds of store efficiency, but undesirable on protection and structural grounds. If the former were used, programs would have to be exceedingly careful to prevent a caller, accidentally or maliciously, providing a segment in some way unusable (too small, for example, or a non-resident segment for a resident, or a segment subject to some external interlock); conversely, the called procedure could keep a capability for the workspace and so obtain later access to some of the caller's data. It thus seemed preferable, though more expensive, for each stack to be local to a protected procedure. The same stack could of course be shared between different instances of a program within a process (this is done for directory managers, for example).

When an Algol68 protected procedure is entered, this corresponds to initiating execution of the program at the 'BEGIN' in the runtime system; when execution reaches the corresponding 'END', a RETURN instruction is executed. In other words, each time a protected procedure is entered the effect is a complete run of the corresponding program (although data can be passed as the argument to a particular run). Division of the operating system in this way into separate programs gives a very suitable degree of modularisation.

However, it was soon found that this structure was not adequate, because the underlying subroutine-like nature of protected procedure calls was inadequate. Treating each entry as a separate run of the program did not correspond to the desired operation of several of the system programs. For example, the program implementing interactive stream wished to perform some initialisation (such as acquiring message channels) on its first entry, then accept numerous entries requesting input or output on the stream, then after a final 'close' entry it would terminate its activity and return an error to any subsequent entry. This could have been achieved by using variables outside of the normal Algol68 storage (remember that even the heap is initialised at the start of the runtime system), but this would involve writing the program in a most unnatural style, and one much less convenient than the one made possible by the solution I produced.

To arrive at the solution, we consider each protected procedure at any time to be either 'active' or 'inactive'. The state of this flag is maintained by the runtime system, and is retained between entries to the protected procedure. Initially (when the protected procedure is created), its state is 'inac-

tive'. When the procedure is entered, if its state is 'inactive' control starts at the beginning of the program. If control reaches the end of the program, its state is set to 'inactive' and a RETURN instruction is executed. A routine named 'return' is provided by the runtime system which, when called, sets the state to 'active', preserves the stack pointers and executes a RETURN instruction. When the protected procedure is next entered it is found to be in the 'active' state, so the stack pointers are restored by the runtime system, and control is transferred as if by subroutine exit from the call of 'return'. The interactive stream protected procedure could now be written in the following form:

```
BEGIN
   .
   .  #initialisation#
   .
   UNTIL entry reason = close
   DO CASE entry reason
      IN .
         .  #work#
         .
      ESAC;
      return(result)
   OD;
   .
   .  #tidy up#
   .
   return(0);
   DO return fault(illegal entry) OD
END
```

The relationship between the 'enter' routine in the calling protected procedure and the 'return' routine in the called protected procedure is very similar to that between co-routines.

This mechanism has proved very satisfactory in practice, and is used by most of the programs of the operating system. Even for programs which do not wish to preserve data or flow of control information between calls, use of the form

```
DO CASE entry reason
   IN .
      .
      .
   ESAC;
   return(result)
OD
```

is convenient in avoiding the overhead of runtime system initialisation code. Returning from a call of 'return' takes only 10 instructions.

It was subsequently noted that the form of program obtained

by use of 'return' is remarkably similar to that of programs which service requests received as messages from some other process. For example:

```
BEGIN
   .
   .  #initialise#
   .
   DO receive message;
      CASE request
      IN .
         .  #work#
         .
      ESAC;
      return message(result)
   OD
END
```

Work has been done to develop an architecture to take advantage of this similarity, where the form of a call is the same to another process as to another protected procedure in this process [37]. The similarity with monitors should also be noted.

A minor improvement which could be made to the enter/return mechanism would be to alter the way in which arguments are presented and received, so as to bring out more clearly the symmetry between 'enter' and 'return'.

3.2 Program Environment

The Algol68C compiler makes no assumption about the target computer, on which the program it compiles will run. The initialisation file for the compiler, together with the translator, specify all that needs to be known about the architecture of the target computer. The environment file corresponding to the MC segment of the runtime system specifies the state in which the target computer will be on entry to the program. It is this state which I wish to discuss here. Apart from the information in the environment file for MC, Zcode assumes that the three stack pointer registers (4,5,6) are properly loaded - it is the responsibility of MC and the translator to ensure that this is so. We can consider MC as converting the environment which we are given into an environment in which compiled code can execute, and the environment file for MC then contains details of this environment. Thus, by changing MC and by changing the environment file, we could arrange to run (MC+program) in some different environment. Further, we can run without MC, provided that the environment with which we are provided conforms to the constraints on the stack pointers and can be described by an environment file. The compiler would be willing to compile code to run in any such environment. This

holds out a prospect of a departure from the normal approach to using a high level language. It has previously been assumed that a compiler will compile a complete 'program' which will run in the precisely constrained environment provided by the language runtime system. In the CAP version of Algol68C, we have seen how all of the runtime, except MC, is treated as ordinary program compiled in the normal way. We can thus consider there to be no runtime system other than MC; the other segments, MIN, SER, SYS and USE, are merely a particularly useful and commonly used library. If we now were able to take the step of allowing users to specify different environment files in place of that corresponding to MC, then users would be able to compile their code to run in any suitable environment.

The severely limiting factor in applying this technique at present is the 'suitability' of an environment. The compiler applies several constraints to the environment. At present the compiler requires the existence of the heap storage allocation routine, but we have seen above how this requirement could be removed. The major difficulty then would be the stack. Ideally, we would wish to be able to run without a stack, but there is little prospect of this in the present compiler. A considerable improvement would be obtained if the required environment was just a description of the base and limit of the workspace, with sufficient freedom in where and how this description was to be found.

Given an easing of the suitability constraint, and given a convenient way of manufacturing environment files (at present an ill-defined and error-prone task), the compiled code could run in many different environments. The environment file would, of course, contain descriptions of where arguments are to be found - these would be in the same form as, in an environment file used by a separately compiled segment, the descriptions of non-local identifiers are presented in the current system.

Such a programming system would greatly increase the utility of a language system; as well as being able to compile programs in the traditional manner, one could compile code to imbed in existing machine code environments, or in environments provided by other languages. There are many major operating systems and pieces of software written in assembler or low-level languages, and the ability to include high-level language sections could greatly ease the problems of their repair and extension.

No work has been done on further investigation in this line. The two aspects requiring most work would be the description of the environment, and resolving the problems of handling externally defined objects [C2]. Such a system would derive quite readily from the approach adopted throughout the development of the CAP Algol68C system.

Section D: Summary and Conclusions

It was the intention, when the work described in this thesis was initially embarked upon, that most of the research would be concerned with the invention of new language features. However, the system finally produced and now existing has no such language features (other than 'CODE' sections). Indeed, not only does the language system produced have no features which are peculiarly 'system programming language' features, but equally it has no features peculiarly oriented towards the architecture of the target computer, the CAP. It is not obvious, then, why such a language system should have had the success that it undoubtedly has had in being favoured by those writing system programs for the CAP.

Admittedly, there have been circumstances, external to the design of the language system, working in its favour. The programmer supporting and developing the Algol68C system for CAP (me), was at all times closely in contact with those writing the system programs, and so was able to respond to requirements as they arose. Also, the machine independent compiler was written and maintained in Cambridge. However, these remarks are equally applicable to the alternative high level language system available, BCPL. We must also bear in mind that Algol68, despite the obscurity and inaccessibility of its definition, and despite the fact that one can readily write very bad Algol68 programs, is actually a very convenient language to write in once you have tried it. All other things being equal, I would much prefer to write in Algol68 than in BCPL, and I believe this view to be shared by most programmers with experience of Algol68.

What, then, has made the CAP Algol68C system convenient for system programming?

The first feature is compiler portability: with regard to which machines and operating systems we could compile on, and with regard to which machines and operating systems we could compile for. This has been achieved by maximising the machine independence and system independence of the compiler (particularly by use of a low-level intermediate code) and of the runtime system (by writing as much of it as possible in a high-level language). This has given us freedom, as the CAP operating system developed, to cross-compile firstly from a different machine, then from a different operating system running on the CAP. In effect, the compiler has followed wherever the programmers wished to lead it.

Enforcing strict separation between the compiling system and the runtime system has been very beneficial. This separation has been made to the extent that neither the compiler nor the translator contain any built-in assumptions about the runtime

system. This has made it a straightforward matter to change the runtime system as and when the operating system facilities used changed. Because the runtime system does not entail changes to the compiler or translator, no specialist knowledge or expertise are required - those writing the operating system can, if desired, change the runtime system without reference to a language expert. Equally, the same compiling software can be used with more than one runtime system. At one stage, three were provided - for running under the temporary operating system , for running on an empty machine, and for sub-processes of the coordinator. (There is currently only one runtime system.)

A consequence of this separation and of the manner in which the runtime system is provided has been that there are no pre-requisite mechanisms. If a program segment does not use certain facilities normally provided by the runtime system, then they need not be provided. This feature is an essential requirement if we are to be able to write the more intimate parts of the operating system, such as the coordinator or the process driving the swapping disc. Achieving this effect comes naturally, given the separation, if we write the runtime system in the language itself, compiled in an unprivileged manner; then we can compile another source text, such as the coordinator, instead of all or part of the runtime system - there is no possibility of this not being acceptable to the compiling system. At the other extreme, a large and elaborate runtime system is available to those programs which require it.

In achieving the above effects, the separate compilation mechanism has been useful. Not only does it allow pre-compilation of the runtime system, and segmentation to optimise virtual memory traffic, but it allows the executable code of the runtime system to be shared on a system-wide basis. As described above, several extensions to the separate compilation mechanism are desirable.

The technique of considering the operating system, not as a single program, but as a set of co-operating programs, is generally felt to have been a success. This is partly due to the CAP having a conveniently modular architecture, and partly due to the lack of a good modular construct in Algol68, but it also has merits in its own right. Primarily, it avoids having operating system design decisions imbedded in, or even pre-empted by, language system design decisions. Also, it is attractive to have language independent interfaces to operating system modules; this is, in any case, essential for the publicly accessible modules. I believe that multi-processing and inter-process communication facilities, although desirable in a language when experimenting with such mechanisms, are not desirable in an implementation system.

There remain, of course, unresolved problems. Foremost amongst these is the handling of extra-lingual objects - objects defined outside the language system, in terms of another

language, or of the hardware. The Algol68C 'CODE' section, although strictly sufficient to handle these objects, is not a satisfactory solution. It is good enough for imbedding extra-lingual operations, such as input-output orders, and has allowed the definition of a satisfactory set of routines for the handling of capabilities, but a better solution is highly desirable for the general problem. This, hopefully, would arise as the outcome of further research along the lines suggested in our discussion on such objects earlier. The impact of this problem has been very much reduced on the CAP by a general strategy, in building the operating system, of restricting each data structure so that only one program is responsible for operations upon it. This is highly desirable to promote integrity and security in the operating system, but has also had the effect that complicated objects are never passed between programs. Calling interfaces are mainly a set of (up to 4) integers, with occasionally a single integer array or other capability. Thus, although the problem should not be ignored, other design criteria counteract its effect. Note, however, that any language system of sufficient power for writing an operating system such as envisaged here must contain facilities for executing specific machine code orders (for I/O), and facilities for allowing the bit-level specification of data objects. This can be seen even in specially designed 'System Programming Languages' [38,39].

Another area where more research could profitably be made is, as discussed above, to attempt to maximise the flexibility available in specifying the environment available to compiled code.

Finally, two conclusions emerge from this work. Firstly, it is important to separate the design of the operating system from the design of the language system being used to write it. Secondly, by judicious design of the implementation of the language system, system programming language features become less important.

Bibliography

1. 'Revised Report on the Algorithmic Language Algol68', A.van Wijngaarden et al; Acta Informatica vol.5, 1-3 (1975).

2. 'Abstraction and Verification in Alphard: Introduction to Language and Methodology', Wm.A.Wulf et al, Carnegie-Mellon University technical report (1976).

3. 'Programming with Abstract Data Types', B.Liskov et al, SIGPLAN Notices vol.9, 4 (Apr 1974).

4. 'On System Programming Languages', G.Goos, working document for WG2.1 meeting in Fontainbleau (1972).

5. 'A Syntax-directed Compiler for Algol60', E.T.Irons et al, CACM vol.4, 15 (1969).

6. 'On the Implementation of Algol68', B.J.Mailloux, Mathematical Centre, Amsterdam (1969).

7. 'Algol68C Reference Manual', S.R.Bourne et al, Computer Laboratory Cambridge (1975).

8. 'Zcode, a Simple Machine', S.R.Bourne, Computer Laboratory Cambridge (1975).

9. I.Walker, forthcoming Ph.D. dissertation, Computer Laboratory Cambridge.

10. 'Algol68C Implementors' Guide', A.D.Birrell, Computer Laboratory Cambridge (1975).

11. 'Protection of Computer Information', J.Saltzer and M.Schroeder, Proc.IEEE, vol.63, 9 (1975).

12. 'The Structure of a Well-protected Computer', R.D.H.Walker, Ph.D. dissertation, Computer Laboratory Cambridge (1973).

13. 'CAP Hardware Manual', Computer Laboratory Cambridge (1976).

14. 'CAP Operating System Manual', Computer Laboratory Cambridge (1976).

15. 'The CAP Project, an Interim Evaluation', R.M.Needham, in Proceedings of SOSP6, Operating Systems Review vol.11, 5 (1977).

16. 'The CAP Filing System', A.D.Birrell, in Proceedings of SOSP6 (1977).

17. 'CAP System Programmers Manual', Computer Laboratory Cambridge (1976).

18. 'The Cambridge CAP Computer and its Protection System', R.D.H.Walker and R.M.Needham, in Proceedings of SOSP6 (1977).

19. 'The Text of OS-Pub', C.J.Strachey, University of Oxford.

20. 'Algol68C Compiler Technical Description', S.R.Bourne et al, Computer Laboratory Cambridge (in preparation).

22. 'The Berlin Algol68 Implementation', W.Koch et al, SIGPLAN Notices vol.12, 6 (June 1977).

23. 'Algol68R Users Guide', I.Currie et al, H.M.S.O (1973).

24. 'Definition Modules', S.Schumann, working document for WG2.1 meeting at Fontainbleau (1972); also in Algol Bulletin.

25. 'The Simula 67 Common Base Language', Dahl et al, report S-22 Norwegian Computing Centre, Oslo (1970).

26. as [3].

27. 'BCPL Reference Manual', M.Richards, Computer Laboratory Cambridge.

28. 'The Programming Language PASCAL', N.Wirth et al, Acta Informatica 1 (1971).

29. 'JANUS', W.M.Waite, University of Colorado.

30. 'An Optimised Translation Process and its Application to Algol68', P.Branquart et al, Report R204, MBLE Brussels (1974).

31. 'Intermediate Languages for Compilers', E.F.Elsworth, Ph.D. dissertation, Computer Laboratory Cambridge (1976).

32. 'Aspects of a Capability Based Operating System', C.J.Slinn, Ph.D. dissertation, Computer Laboratory Cambridge (1976).

33. 'Writing the Elementary Function Procedures for the Algol68C Compiler', P.Kemp, NSF/ERDA Workshop on Portability of Numerical Software (June 1976).

34. Thesis Proposal, A.D.Birrell, Computer Laboratory Cambridge (1974).

35. 'MARY - a Portable Machine Oriented Programming Language', M.Rain et al, MOLB 3 (Oct 1973).

36. 'Data Types and Extensible Languages', P.Jorrand in Proceedings of International Symposium on Extensible

Languages, Report FF2.0143 IBM Grenoble (Sep 1971)

37. Forthcoming Ph.D. dissertation, D.J.Watson, Computer Laboratory Cambridge.

38. 'Early Experience with MESA', C.M.Geschke et al, Xerox Palo Alto Research Center (Oct 1976).

39. 'S3' International Computers Limited.

40. 'Storage Management for Algol68', A.D.Birrell, SIGPLAN Notices vol.12, 6 (June 1977).

41. CAP Project internal notes, Computer Laboratory, University of Cambridge.

Appendix X

CAP Algol68C Documentation

Algol68C on CAP

Andrew D. Birrell

1. The A68C Command

This command invokes the Cambridge Algol68C compiler. The compiler will accept the source of a single Algol68C program segment, and the command can be used to produce the intermediate code for the segment, or to produce executable binary for the segment, or to update a PCB, or to create (and possibly execute) a new PCB. It thus allows a range of uses, from compile-and-go for a simple program, to maintenance of a large multi-segment program. The choice between these uses is controlled by the presence and content of the 'ZCODE' and 'NAME' heading items, and the keywords 'ZCODE' and 'PCB' on the command line, as described below and in the description of the ZCAP command.

The keyword parameter HEAP should be set to an integer value if you wish to alter the amount of store used for compiler workspace. The default value is sufficient for quite large program segments, and can profitably be reduced (to, say, 10K) for compiling small programs. The maximum useful value is at present 32K.

Strings specifying input or output for the compiler may each be as described below for strings presented to 'sysopen'.

The compiler's main input stream, on which it expects the source of the program segment, is specified by the string given as the first positional parameter on the command line.

The keyword parameter 'SYSPRINT' may be used to re-direct the compiler's diagnostic output stream. The default is '/M'.

The program heading is presented in up to two portions; the compiler looks first at the string, if any, given as the value of the OPT keyword parameter, then accepts heading items from the start of the main input.

The compiler will accept the following heading items:

USING {handle} FROM {string}
This is described in the Algol68C Reference Manual. The string specifies the environment file to be read. Note that

the string need not be the same string as was used when producing the environment in an earlier compilation. Note that environment files read when compiling the parent of this segment are also read when compiling this segment. If no USING directive is found, a default is assumed which gives access to libraries providing the full facilities described in Chapter 6 of the Manual - see below.

TITLE {tag}
The tag is verified on the compiler's diagnostic output stream.

ENVOUT {string}
If the segment contains any ENVIRON statements then this string is used to open the output stream for the environment file. By default the string 'ENVOUT' is used, and a suitable keyword parameter may be supplied on the command line when invoking the compiler.

XREF
Requests that the compiler produce cross-reference data regarding this segment, for subsequent processing by the A68XREF program. See also the 'XREFOUT' heading item.

XREFOUT {string}
If the heading item XREF is present, then this string is used to open the output stream for the cross-reference data. By default the string 'XREF' is used, and a suitable keyword parameter may be supplied on the command line when invoking the compiler.

ZCODE {string}
This string is used to open the output stream for the intermediate code produced by the compiler. By default the string 'ZCODE' is used. If the string 'ZCODE' is used, and the keyword ZCODE was not supplied on the command line when invoking the compiler, then the Zcode is sent to an anonymous stream, and after the compilation the Zcode translator ('ZCAP') is automatically invoked. In this case, the translator uses this anonymous stream as its only input, and has available to it the command line used to invoke the compiler. See below for description of the action of the translator.

NAME {string}
This string is passed to the Zcode translator - see below.

TRACE {bits-denotation}
A facility for compiler debugging

KEY {bits-denotation}
Ignored at present

CASESTROP

Selects as 'stropping' convention from this point onwards the convention that tags are written in lower case letters, with digits allowed, and indicants are written in upper case letters. This is the default stropping convention.

QUOTESTROP

Selects as 'stropping' convention from this point onwards the convention that tags are written in letters of either case, with digits allowed, and indicants are written in letters of either case, and each indicant must be preceded by the apostrophe character. Upper and lower case letters are treated as equivalent.

UPPER

Selects as 'stropping' convention from this point onwards the convention that tags are written in lower case letters, with digits allowed, and indicants are written in upper case letters, with digits allowed, and each indicant may optionally be preceded by a dot.

POINT

Selects as 'stropping' convention from this point onwards the convention that tags are written in letters of either case, with digits allowed, and indicants are written in letters of either case, with digits allowed, and each indicant must be preceded by a dot. Upper and lower case letters are treated as equivalent.

2. The ZCAP Command

This command invokes the CAP translator for Zcode (the intermediate language used by Algol68C). The translator is usually invoked by the A68C command, but may be invoked directly. When invoked by the A68C command, the input stream for the translator consists of the Zcode produced by the compiler. When invoked directly, the input consists of the concatenation of the streams specified by the serial parameters on the command line, in numerical order from 1.

The input stream should contain Zcode for one or more Algol68C program segments. The program segments must form a single sub-tree of the tree of Algol68C segments which form the complete program, and they must be presented in a top-down order. The Zcode is assembled into a single CAP machine segment. The address which this segment must have in the resulting protected procedure is allocated by the translator and verified on the main output.

The translator can dispose of the segment containing the assembled binary in a number of ways. If it can determine a file title for preserving the binary then it will do so and stop (otherwise the binary will be placed in a PCB, as described below). If the command line contained the keyword 'BIN', then the string given as its value is used to preserve the segment. Otherwise, if when the segment was compiled the heading had a NAME directive containing

"S=filetitle"

then this file title is used to preserve the segment. In all other cases, the translator will place the binary in a PCB. If the translator needs to create the PCB, it is initialised with a suitable sized stack and heap. If the translator knows the file titles for the superior segments of this one, then it will update the PCB to contain these. If the command line contains the keyword 'PCB' then the string given as its value nominates the PCB, which is created if it does not already exist. Otherwise, if the heading had a NAME directive containing

"P=filetitle"

then this file title nominates the PCB, which is created if it does not already exist. Otherwise a PCB is created, initialised, updated, and entered to execute the compiled program.

If one or more of the Algol68C program segments presented in this run of the translator contains an ENVIRON statement for which the corresponding Algol68C program segment is not presented in this run of the translator, then each such program segment presented in this run must have had in its heading a NAME directive containing

"L=filetitle"

Further, all these file titles must be identical. This file title is then used during the present run of the translator for the output of linking information, and the same file title is opened for input when assembling the program segments corresponding to such ENVIRON statements.

In addition to standard Zcode, the translator will accept the directive:

M op ba bm N insert machine code instruction (given in octal), eg. M123 10 11 +123

Some other extensions exist, but they are not useful to the ordinary user. Registers in M directives are octal CAP machine register numbers. The translation from Zcode registers is as follows:

Zcode (octal)	= CAP (octal)	= CAP (decimal)	Use
0	0	0	constant 0
1	1	1	constant 1
4	13	11	{system}
5	16	14	local stack
6	14	12	global stack
7	15	13	{system}
10	2	2	work, result
11	3	3	work
12	4	4	work
13	5	5	work
14	6	6	work
15	7	7	work
16	10	8	work
17	11	9	work
20	accumulator		

Offset parts in M directives are as for Zcode instructions; they are treated as if the architecture allowed 32 bits. This is achieved by, as necessary, converting n-type orders into s-type ones, or inserting MOD orders.

All ZCODE values except floating point are manipulated in CAP registers 0 to 14. Floating point values are manipulated in the accumulator, and exist in main storage as standardised 24+8 numbers only.

Strings between CODE and EDOC in programs are passed to the translator, as described in the Algol68C Reference Manual. If a CODE section contains an applied occurrence of an identifier, then the Zcode produced by the compiler will contains in its place a corresponding Zcode register number and offset. Note that it is not sensible to have applied occurrences inside 'M' directives.

3. The CAP Algol68C Libraries

The libraries provided for Algol68C on CAP provide numerous facilities beyond those specified in the Algol68C Reference Manual. The authoritative definition for the facilities described in the manual is the manual; an outline of those facilities and of the extensions is given here. Other definitions than those listed here are available to the user program, but this is due only to the inadequacies of block structure and such definitions should not normally be used, and certainly not without consulting the author.

The library is arranged in four machine segments:

```
MIN is P0
SER is P1
USE is P2
MATH is P3
```

The default USING directive gives access to MATH, USE, SER and MIN with the user program starting in P4. This default may be over-ridden by providing a USING directive as follows:

```
USING USER FROM ".**.A68.ENV.MIN"
    gives MIN, with user program from P1

USING SERVICES FROM ".**.A68.ENV.SER"
    gives MIN and SER, with user program from P2

USING USER FROM ".**.A68.ENV.USE"
    gives MIN, SER and USE, with user program from P3

USING USER FROM ".**.A68.ENV.MATH"
    gives all four segments, with user program from P4.
```

The protected procedure manufactured (by linking a PCB, by calling 'MAKEENTER', or by a SYSGEN) must contain the appropriate library segments, the user program segment(s) in the slots nominated by the CAP Zcode translator, a workspace segment for the stack in I0 and a workspace segment for the heap in I1. The file titles which can be used to retrieve the library segments are of the form .*.A68.BIN.<level><serial>, for example .*.A68.BIN.MIN84, and all have the same value of 'serial'. In most cases, the translator will ensure that a nominated PCB contains the appropriate segments, as described above. The access permitted for the protected procedure to the library and user program segments must be at least read-execute (RE), and to the stack and heap must be at least read-write (RW). The size of the stack segment required for a particular program is difficult to assess, but a minimum of 300 words is recommended. The heap is not required by library levels MIN and SER unless provoked by the user program, but USE will require at least 300 words for buffer space (more if a default input stream is provided).

In addition to the definitions described below, and initialising variables defined below, each segment of the library performs various actions. The MIN level, in addition to initialising 'runtime error' and the 'ignore attention' flag, calls 'reserve slot' to mark that the segments used for the stack and heap are not to be allocated. The MIN level also executes 'ENVIRON USER', but only if no attention has been notified to the program during the initialisation. On exit from the program, a RETURN order is executed, with in B2 the last argument that was presented to 'set return code' (or 0 if there has been no such call). The SER level calls 'makeind(3)' (as does the 'return' subroutine on re-entry), and assigns to 'runtime error' the routine

```
(STRING s, INT i)VOID:
    ( return fault(i); set return code(i); GOTO stop )
```

If no attention has been notified to the program during the initialisation, the SER level executes 'ENVIRON SERVICES'. If the program has been entered under the 'command' interface, and if the keyword parameter 'HEAP' is present, the USE level reads its value. If this is not an integer then 'runtime error' is called, otherwise the heap segment is adjusted to this size (in words). The USE level assigns 'a68c error' to 'runtime error' and 'a68c attention' to 'attention routine', and calls 'allow attention'. If no attention has been notified to the program, the USE level executes 'PRELUDE USER'. On exit from the program, the USE level closes all files that have not been otherwise closed. The MATH level executes 'PRELUDE USER'.

Note that all REAL arithmetic is performed using 24-bit mantissa and 8-bit base-2 exponent. This gives about 6.9 decimal digits of accuracy. The rounding algorithm provided by the hardware is close to being unbiased. The mathematical functions (except REAL**REAL) are believed to be very accurate (close to 6.9 digits).

Operators specified in the manual are not included in the description below. All such operators are available at any level, except that ** for REAL operands and all operators for COMPL are available only when using the MATH level.

3.1 Definitions provided by MIN

MIN includes subroutines for allocation of space on heap and stack.

3.1.1 Arguments from caller

enterarg1	INT	the value B1 had on entry or re-entry.
enterarg2	INT	the value B2 had on entry or re-entry.
enterarg3	INT	the value B3 had on entry or re-entry.
enterarg4	INT	the value B4 had on entry or re-entry.
enterarg5	INT	the value B5 had on entry or re-entry.
enter access	INT	the access bits obtained from the ENTER capability used to enter the program.
real arg	REAL	the value in the accumulator on entry or re-entry

3.1.2 Environment enquiries

int lengths	INT	see manual;	1
real lengths	INT	see manual;	1
bits lengths	INT	see manual;	1
bytes lengths	INT	see manual;	1
int shorths	INT	see manual;	1
real shorths	INT	see manual;	1
bits shorths	INT	see manual;	1
bytes shorths	INT	see manual;	1
int size	INT	see manual;	1
real size	INT	see manual;	1
bits size	INT	see manual;	1
bool size	INT	see manual;	1
char size	INT	see manual;	1
bytes size	INT	see manual;	1
int align	INT	see manual;	0
real align	INT	see manual;	0
bits align	INT	see manual;	0
bool align	INT	see manual;	0
char align	INT	see manual;	0
bytes align	INT	see manual;	0
maxint	INT	see manual;	2**31 - 1

bitswidth	INT	see manual;	32
byteswidth	INT	see manual;	4
max abs char	INT	see manual;	255
blank	CHAR	see manual;	"*S"
error char	CHAR	see manual;	"**"
null character	CHAR	see manual;	REPR 0
flip	CHAR	see manual;	"T"
flop	CHAR	see manual;	"F"
int width	INT	see manual;	10
real width	INT	see manual;	8
exp width	INT	see manual;	2

3.1.3 Fault numbers

error number mask	INT	ABS16rf0ff00ff
error data mask	INT	ABS16r0000ff00
error count mask	INT	ABS16r0f000000
previous error	INT	fault number issued after giving suitable message for an error; = ABS16r80000000
stack full error	INT	Fault number issued when the stack is full
heap full error	INT	Fault number issued when the heap is full
i indy full error	INT	Fault number issued when the I capability segment is full
bad open idf error	INT	Fault number issued if an illegal string is presented to 'lookup idf', 'sysopen', or 'open'
file closed error	INT	Fault number issued when using a file which has been closed
non digit error	INT	Fault number issued when a character other than a digit was the first character encountered when reading an integer or exponent-part
heap size error	INT	Fault number issued if a non-integer was was given as the value of the 'HEAP' keyword parameter
mathlib error	INT	Fault number issued if an illegal argument was given to one of the mathematical routines
not segment error	INT	Return code issued if 'lookupidf' or 'sysopen' is asked to open for input a file title indicating a PCB or directory
sysread error	INT	Return code issued if the string ""

		could not be opened on 'standin channel' due to an error in the 'SYSREAD' keyword parameter
file ended error	INT	Fault number issued on attempting to read beyond the end of a book
charpos error	INT	Fault number issued on attempting to position outside the bounds of the current line
backspace error	INT	Fault number issued on attempting to backspace before the start of the line
not open error	INT	Fault number issued on attempting to perform transput on a FILE for which 'open' failed
reserve error	INT	Fault number issued on attempting to reserve a SLOT in the 'I' capability segment, the SLOT being already in use
no digit error	INT	Fault number issued if no character could be read when reading an integer or exponent-part
int overflow error	INT	Fault number issued if the digits of an integer or exponent-part being read would exceed maxint
readbool error	INT	Fault number issued if neither flip nor flop is encountered when reading a BOOL
real overflow error	INT	Fault number issued when reading a REAL if the magnitude of the value would exceed maxreal
plus i times error	INT	Fault number issued if 'I' or 'i' are not encountered at the appropriate point when reading a COMPL value
fix float error	INT	Fault number issued if the arguments to one of the REAL or COMPL output or conversion routines are inconsistent
normal error	INT	Fault number issued if the standard deviation for 'normal random' is not acceptable
return correct	INT	see manual; 0
return warn	INT	see manual; 1
return soft	INT	see manual; 2
return hard	INT	see manual; 3
return fatal	INT	see manual; 4

3.1.4 BITS, BYTES, CHAR and STRING

bitspack	=([]BOOL a)BITS:	see manual
bytespack	=(STRING a)BYTES:	see manual
packbytes	=(REF[]CHAR a)BYTES:	see manual
punstring	=(REF[]INT v)STRING:	a punning operation yielding the string assumed to be in 'v', whose stride must be 1
packstring	=(REF[]INT v)INT:	see manual
unpackstring	=(STRING s, REF[]INT v)VOID:	copies the characters of 's' into 'v' (whose LWB must be 0) in a format suitable for 'packstring'
copystring	=(REF[]INT v)STRING:	allocates heap storage and copies the packed string from 'v' into it
putbyte	=(CHAR c,REF[]INT v,INT i)VOID:	places ABS c in the i'th byte of 'v', whose stride must be 1. Bytes are numbered as for the PUTBYTE instruction
char in string	=(CHAR c,REF INT i,STRING s)BOOL:	see manual
=*	operator	defined for all combinations of CHAR and STRING. Meaning as for '=', except that upper and lower case letters are treated as equivalent
~*	operator	the negation of '=*'

3.1.5 Operations on SLOT's

SLOT	mode	used to represent the address of a capability
+	=(SLOT s,INT i)SLOT:	the address of offset 'i' from 's'
G	=(INT i)SLOT	the i'th capability in the G capability segment

```
A                   =(INT i)SLOT     the i'th capability in the A
                                     capability segment
N                   =(INT i)SLOT     the i'th capability in the N
                                     capability segment
P                   =(INT i)SLOT     the i'th capability in the P
                                     capability segment
I                   =(INT i)SLOT     the i'th capability in the I
                                     capability segment
R                   =(INT i)SLOT     the i'th capability in the R
                                     capability segment

seg addr            =(INT i,j)SLOT:  the j'th capability in the i'th
                                     capability segment

n0                  SLOT             N0
n1                  SLOT             N1
n2                  SLOT             N2
null capability     SLOT             G3

maparray            =(SLOT s)REF[]INT:
                                     yields an array with LWB=0 and
                                     UPB=65534 corresponding to the segment
                                     whose capability is or will be at 's';
                                     the capability is not touched in any
                                     way

mapsegment          =(SLOT s)REF[]INT:
                                     yields an array with LWB=0 and correct
                                     upper bound for the segment currently
                                     described by 's'

mapstring           =(SLOT s)STRING: yields the STRING currently existing
                                     in the segment described by 's'

indinf              =(SLOT s)INT     the result of an INDINF order
seginf              =(SLOT s)INT     the result of a SEGINF order
segsize             =(SLOT s)INT:    the number of words in 's'
cseginf             =(SLOT s)INT     the result of a CSEGINF order

movecap             =(SLOT s,t)VOID  performs a MOVECAP order from 's' to
                                     't'
movecapa            =(SLOT s,t)VOID  performs a MOVECAPA order from 's' to
                                     't'
refine              =(SLOT s,INT i,SLOT t)VOID:
                                     performs a REFINE order with base 's'
                                     and size and access 'i' to 't'

makeind             =(INT i)VOID     performs a MAKEIND order with argument
                                     'i'
```

3.1.6 Fault and attention handling

runtime error	REF PROC(STRING,INT)VOID:	the routine referred to by this variable is called whenever the library detects an error, or when an error is indicated to the program by the system The STRING argument in such calls is "" and the INT argument is the fault number. This variable is initialised by MIN to cause exit from the program.
ignore attention	PROC VOID	sets the 'ignore attention' flag to inhibit calling of 'attention routine' (see below)
allow attention	PROC VOID	unsets the 'ignore attention' flag, and if 'attention happened' would yield TRUE then calls 'attention routine'
attention routine	REF PROC VOID	the routine referred to by this variable is called whenever an 'attention' is notified to the program, unless at that time the 'ignore attention' flag is set. Exitting from this call will cause a fault. This variable is not initialised in MIN.
attention happened	PROC BOOL	yields TRUE if an attention has been notified to the program (regardless of 'ignore attention'), unless 'clear attention' (in SER) has been called subsequently.

3.1.7 Enter and return

enter	=(SLOT s, INT i,j,k,l,m)INT:	enters the cpabaility 's' with 'i' to 'm' in B1 to B5; yields the value B2 has when control returns to this program
enter2	=(SLOT s, INT i,j,k,l,m, REF INT p,q,r,s)VOID:	as 'enter' but the values found in B2 to B5 on return are assigned to the variables 'p' to 's'
set return code	=(INT i)VOID:	see manual; left in B2 on exit from the

		program
return	=(INT i)VOID:	executes a RETURN order with 'i' in B2, but when this program is next entered control will resume as if by exit from the call of 'return'. 'enterarg1' to 'enterarg5' and 'enter access' will have been updated to the values found in B1 to B5 and B14 on re-entry. This facility can be used in a manner resembling co-routines.
return2	=(INT i,j,k,l)VOID:	as 'return', but 'i' to 'l' are placed in B2 to B5 before executing the RETURN order.
stop	label	after the 'ENVIRON USER' statement

3.1.8 Move, moverow, movestring

move	=(REF[]INT v,w, INT i)VOID:	copies the contents of the first 'i' elements of 'v' into those of 'w'. Both LWB's must be 0 and the UPB's must both be at least 'i'
movestring	=(STRING s, SLOT t)VOID:	places in 't' a capability for 's'
moverow	=(REF[]INT v, INT i, SLOT s)VOID:	places in 's' a capability for 'v', whose LWB must be 0 and UPB must be 'i'

3.1.9 Allocation routines for SLOT's

i indy size	INT	the number of capabilities in the I indirectory
getslot	PROC SLOT	allocates a capability in the I indirectory; will not allocate a capability which has been marked by 'reserve slot', and will not re-use a capability until it has been freed by 'freeslot'. 'getslot' will call 'runtime error' if it cannot allocate a

		slot.
reserve slot	=(INT i)SLOT:	indicates that the i'th capability of capability segment 'I' is to be assumed to be in use. Yields the corresponding SLOT value.
free slot	=(SLOT s)VOID:	if 's' is in the I indirectory, indicates that it is free for re-allocation. The capability at 's' is over-written with a null capability.

3.1.10 Store management

heap slot	SLOT	the capability used for the heap segment (I1)
stack available	PROC REAL	see manual; the stack is separate from the heap
heap available	PROC REAL	see manual
store available	PROC REAL	see manual
store used	PROC REAL	see manual

3.1.11 Layout of capabilities

cap mctype	INT	d31=1, d30=1; the field used by the microprogram to distinguish types of capability
cap type	INT	d29,28,27,26=1; the field used by software to distinguish types of capability
cap access	INT	d21,20,19,18,17,16=1; the field used for access bits in a store capability
cap length	INT	d15 to d0 = 1; the field used for 'length' in a store capability
hardware bit	INT	d31=1, d30=0; indicates 'store-type' capability to the microprogram
enter bit	INT	d31=0, d30=1; indicates 'enter-type' to the microprogram
store capability	INT	d29=1, d28=0, d27=1, d26=1; indicates a normal store-type capability to the system software
exec access	INT	d16=1; access right in store capability
read access	INT	d17=1; access right in store capability
write access	INT	d18=1; access right in store capability

```
rcap access       INT          d20=1; access right 'RC' in store
                               capability
wcap access       INT          d21=1; access right 'WC' in store
                               capability

send capability   INT          d29=0, d28=1, d27=0, d26=0; for
                               messages
receive capability INT         d29=0, d28=1, d27=0, d26=1; for
                               messages
reply capability INT           d29=0, d28=1, d27=1, d26=0; for
                               messages

null message                INT  d23=0, d22=0; message type
data message                INT  d23=0, d22=1; message type
segment message             INT  d23=1, d22=0; message type
full message                INT  d23=1, d22=1; message type
reply message               INT  d24=1; message type
data reply message          INT  data message ! reply message
seg reply message           INT  segment message ! reply message
full reply message          INT  full message ! reply message

channel access              INT  d29=1, d28=1, d27=0, d26=0; for SETUP
permission capability       INT  d29=0, d28=1, d27=1, d26=1; for ECPROC

process create permission   INT
signal attention permission INT
capability permission       INT
store permission            INT
peripheral permission       INT
info permission             INT
system crash permission     INT
channel permission          INT

clear attention capability  INT  d29=1, d28=1, d27=0, d26=1; for ECPROC
```

3.1.12 Layout of words

```
rhword            INT          ABS16r0000ffff
lhword            INT          ABS16rffff0000

byte0             INT          ABS16r000000ff
byte1             INT          ABS16r0000ff00
byte2             INT          ABS16r00ff0000
byte3             INT          ABS16rff000000

bit0              INT          ABS16r00000001
  .
  .
  .
bit31             INT          ABS16r80000000
```

3.2 Definitions provided by MIN

```
ecproc          SLOT          G1
setup           SLOT          G2
```

3.2.1 Sending messages

```
send null message =(SLOT s)VOID: send null message down 's'

send data message =(SLOT s, INT a,b,c,d)VOID:
                                  sends (a,b,c,d)

send segment message =(SLOT s, t)VOID:
                                  sends 't'

send full message =(SLOT s, INT a,b,c,d, SLOT t)VOID:
                                  sends (a,b,c,d,t)

send null message wait event =(SLOT s)VOID:
                                  send down channel 's' then wait

send data message wait event =(SLOT s, INT a,b,c,d)VOID:
                                  as above

send segment message wait event =(SLOT s, t)VOID:
                                  as above

send full message wait event =(SLOT s, INT a,b,c,d, SLOT t)VOID:
                                  as above

receive null message =(SLOT s)VOID:
                                  receive from 's'

receive data message =(SLOT s, REF INT w,x,y,z)VOID:
                                  assign into (w,x,y,z)

receive segment message =(SLOT s, t)VOID:
                                  movecap into 't'

receive full message =(SLOT s, t, REF INT w,x,y,z)VOID:

receive reply data message =(SLOT s, r, REF INT w,x,y,z)VOID:
                                  receive from 's', assign into
                                  (w,x,y,z), reply capability 'r'

receive reply segment message =(SLOT s, r)VOID:

receive reply full message =(SLOT s, r, INT w,x,y,z)VOID:
```

```
return message    =(SLOT r)VOID:  null reply to message

return data message =(SLOT r, INT a,b,c,d)VOID:
                                  reply (a,b,c,d) down 'r'

return message wait event =(SLOT r)VOID:
                                  reply then wait

return data message wait event =(SLOT r, INT a,b,c,d)VOID:

messages          =(SLOT s)INT:   the number of messages waiting to be
                                  received by receive capability 's'
```

3.2.2 Miscellaneous Coordinator facilities

```
wait event        PROC VOID       coordinator entry

reserve for reading =(SLOT s)VOID:
                                  reserve segment or wait

reserve for writing =(SLOT s)VOID:
                                  reserve for unique access or wait

release reservation PROC VOID     what it says!

clear fault       PROC VOID       ensure program no longer in 'fault
                                  state'

cause fault       =(INT i)VOID:   notify a fault 'i' immediately

return fault      =(INT i)VOID:   place in fault state 'i', but don't
                                  cause any transfer of control in this
                                  protected procedure; safe even if
                                  already in fault state

signal attention =(SLOT p, INT i,j)VOID:
                                  notify process number 'i' of an
                                  attention at level 'j'; permission 'p'

clear attention   =(SLOT p)VOID:  clear attention state; 'p' is
                                  permission or null capability. Also
                                  clears 'attention happened'.

create process    =(SLOT p, s)VOID:
                                  commence running a process whose PRL is
                                  's'; permission 'p'

create prl capability =(SLOT p, s, INT i,j)INT:
                                  create a capability (i,j) in the PRL of
                                  this process, and yield a capability
```

```
                                    for it in 's'; permission 'p'

create capability =(SLOT p, s, INT i,j)VOID:
                                    create an indirectory-level capability
                                    (i,j) and yield it in 's'; permission
                                    'p'

update prl capability =(SLOT p, INT i,j,k)VOID:
                                    update the PRL capability at offset 'i'
                                    to contain (j,k); permission 'p'

read prl capability =(SLOT p, s, REF INT w,x)INT:
                                    assign into (w,x) the words of the PRL
                                    capability referred to by 's';
                                    permission 'p'

read capability =(SLOT p, s, REF INT w,x)VOID:
                                    read the indirectory-level capability
                                    's', permission 'p'

prlgarb          PROC VOID          call the PRL garbage collector

claim device     =(SLOT p, INT i, SLOT s)VOID:
                                    direct future interrupts from device
                                    'i' to this process; yield pstore
                                    capability for device in 's';
                                    permission 'p'

release device   =(SLOT p, s)VOID:
                                    device with pstore 's' no longer needed
                                    by this process; permission 'p'

system crash     =(SLOT p, INT i)VOID:
                                    request the coordinator to stop the
                                    system immediately with reason 'i',
                                    permission 'p'
```

3.2.3 Channel set-up

```
setup receive    =(SLOT s, REF SLOT t, INT i)VOID:
                                    set up a 'receive' capability according
                                    to channel 's' and message type 'i';
                                    result allocated by 'getslot' and
                                    assigned to 't'

setup send       =(SLOT s, REF SLOT t, INT i)VOID:
                                    'send' capability

setup send with reply
                 =(SLOT s, REF SLOT t, INT i, REF SLOT u, INT j)VOID:
```

```
                                    setup send capability for channel 's',
                                    message type 'i', assigned to 't', with
                                    reply type 'j', capability for
                                    receiving replies assinged to 'u'

setup reply         =(REF SLOT s, INT i)VOID:
                                    set up reply capability for sending
                                    replies of type 'i' assigned to 's'

setup reply with store =(REF SLOT s, INT i, SLOT t)VOID:
                                    setup reply capability for replying to
                                    segment-type messages whose segment
                                    will be placed in 't'
```

3.2.4 Timer facilities

```
timer info          =(REF INT rtc, time, data, day)VOID:
                                    assign the current clock values

clock               PROC REAL       see manual
```

3.3 Definitions provided by USE

storeman	SLOT	G4
make enter	SLOT	G5
ioc	SLOT	G6
fault	SLOT	G7

3.3.1 Interface from outside world

user idf	STRING	name of user provided in G8
job number	REF INT	the number of this job, provided in G8
command line	STRING	provided if called by 'command' interface, in A0
command dir	SLOT	directory provided by 'commmand'; A1
command parms	SLOT	'parms' provided by 'command'; A2
current dir	REF SLOT	directory used by routines provided to interface to DIRMAN; initialised to 'command dir'
parms	REF SLOT	program called by routines provided to interface to PARMS; initialised to 'command parms'
machine	PROC BOOL	whether called by 'command' interface
command	=(SLOT prog, parms, STRING line)INT:	enters 'prog' as if called under the 'command' interface, with 'parms' as parameter decoder, and command line 'line'

3.3.2 Interface with DIRMAN

delete access	INT	'D' right in access matrix element, at d16
update access	INT	'U' right in access matrix element, at d16
alter access	INT	'A' right in access matrix element, at d16
max access	INT	indicates maximum obtainable access

```
                                when given as argument to 'retrieve'

create access       INT         'C' access to directory, at d16
vmode access        INT         'V' access to directory, at d16
xmode access        INT         'X' access to directory, at d16
ymode access        INT         'Y' access to directory, at d16
zmode access        INT         'Z' access to directory, at d16

modify access       INT         'M' access to PCB, at d16
inspect access      INT         'I' access to PCB, at d16
link access         INT         'L' access to PCB, at d16

swc access          INT         'S' access to software capability, at
                                d16

all seg             INT         access /RWE, at d16
all dir             INT         access /CVXYZ, at d16
all pcb             INT         access /MIL, at d16
all swc             INT         access /s, at d16

default seg am      INT         matrix /ADURW/ARW/RW/R
exec seg am         INT         matrix /ADURE/ARE/RE/RE
default dir am      INT         matrix /ADCV/ACX/CY/Z
default pcb am      INT         matrix /ADMIL/AMIL/IL/L
default swc am      INT         matrix /ADUS/AS/S/

unknown             INT         VMO type 0
segtype             INT         VMO type 1
dirtype             INT         VMO type 2
pcbtype             INT         VMO type 3
swctype             INT         VMO type 4

retrieve            =(SLOT s, STRING t, INT i)INT:
                                retrieve capability with title 't' from
                                'current dir', with requested access
                                'i' into 's'; yields DIRMAN return code

remove              =(STRING s)INT: remove entry 's' from 'current dir';
                                yields DIRMAN return code

preserve            =(SLOT s, STRING t, INT i)INT:
                                preserve capability 's' in 'current
                                dir' as entry 't' with access matrix
                                'i'; yields DIRMAN return code

alter               =(STRING s,INT remove,add)INT:
                                alter access matrix for 's' in 'current
                                dir' by removing bits 'remove' and
                                including bits 'add'

examine             INT         DIRMAN entry reason
file details        INT         DIRMAN entry reason
file examine        INT         DIRMAN entry reason
```

3.3.3 Interface with STOREMAN

ensure	=(SLOT s)VOID:	update VMO 's' on disc; yields 0
outform	=(SLOT s)VOID:	recommendation to RSM about VMO 's'
change size	=(SLOT s, INT i)VOID:	change size of VMO 's' by amount 'i', which may be positive or negative.
newseg	=(SLOT s, INT i,j)VOID:	allocate segment-type VMO of size 'i' with access 'j'; result to 's'.
capinf	=(SLOT s)INT:	yields information about capability 's'; result is negative if 's' is not for a complete VMO, nor a software capability. Otherwise result is type,access,SIN (see STOREMAN documentation).
get size access	=(SLOT s)INT:	yields size in d23 to d0, access in d31 to d24 (n.b.)
open window	=(SLOT s, INT b,a)SLOT:	yields a SLOT windowing 's' from base 'b' with size and access 'a'
move window	=(SLOT s, INT b,a)VOID:	moves window 's' to base 'b' with size and access 'a'
close window	=(SLOT s)VOID:	close, and write up if needed, 's'
details	=(SLOT s, REF INT size, acc, type)VOID:	assigns details about 's'
new instance	=(SLOT s)VOID:	updates 's' to be a new version of 's'

3.3.4 Calls to i/o system

input	INT	stream direction
output	INT	stream direction
ttr standard	INT	stream state: reflect, escapes, ignore parity
ttr no reflect	INT	stream state
stdout state	INT	stream state: c.r. l.f. after records
overprint state	INT	stream state: c.r. after records

```
sameline state   INT              stream state: force out after records
newpage state    INT              stream state: form feed after records
tr standard      INT              stream state: ASCII, no escapes, parity
tr binary        INT              stream state
tp binary        INT              stream state
close option     INT              state at end of input stream

wrrecord         =(SLOT s, REF[]CHAR v)VOID:
                                  send buffer 'v' to stream 's'

rdrecord         =(SLOT s, REF[]CHAR v)INT:
                                  read record from 's' into 'v'; yields
                                  number of characters, or -1 if a stream
                                  state change was found.

close stream     =(SLOT s)VOID:   If 's' is in capability segment 'I',
                                  then enter 's' to close it; if the
                                  stream was created by 'stream from seg'
                                  or by "/A" to 'sysopen' then the
                                  segment is left in NO.

read state       =(SLOT s)INT:    yields the current stream state of 's'

set state        =(SLOT s, INT i)VOID:
                                  set the stream state of 's' to 'i'

extract doc      =(SLOT s, d, INT t)INT:
                                 extract from stream 's' a document with
                                 terminator 't' into slot 'd'.

insert doc       =(SLOT s, d)INT:
                                 insert document 'd' into stream 's'

stream from seg  =(REF SLOT s, INT d)VOID:
                                  Creates a stream, with direction 'd'
                                  (which should be 'input' or 'output'),
                                  to/from the segment 's'; freeslot(s) is
                                  called, and the stream is placed in a
                                  newly allocated slot assigned to 's'.

lookup idf       =(STRING s, BOOL b, REF SLOT t)INT:
                                  Obtains a capability as specified by
                                  's' and assigns it to 't'.  's' can be
                                  any string as accepted by 'sysopen'.
                                  The stream is for input if 'b' is TRUE.
                                  Capability produced is a segment if
                                  this is reasonable, otherwise a stream.
                                  Yields results as for 'sysopen'.
```

3.3.5 Low-level transput

CHANNEL MODE see manual; is of the form REF X

SYSFILE MODE see manual; is of the form REF Y

sysfile from slot =(REF SYSFILE sf, SLOT s, CHANNEL c)VOID:
analogous to 'sysopen' but takes a SLOT and has no yield (faults in error cases). The channel must be 'standin channel' or 'standout channel'. If the slot is a segment, 'stream from seg' is called to produce the stream; the segment will be available in NO after calling 'close' or 'sysclose'.

sysopen =(REF SYSFILE sf, STRING s, CHANNEL c)INT:
see manual. The meaning of the string is as follows. If the channel is 'parameter channel', then the book produced contains a single line which is the string value given as the value of the keyword parameter 's', or if s = "", then the keyword parameter 'OPT'; but a negative return code will be given if the program was not called under the 'command' interface or if the keyword parameter was not set or was in some way wrong. If the channel is 'string channel' then the book produced has a single line which contains the characters of the string 's'. If the channel is 'standin channel' or 'standout channel' then the following possibilities exist except that only those which are sensible for input are available on 'standin channel', and only those for output on 'standout channel'. If the string is "" then an attempt is made to open using the string 'SYSREAD' (for input) or 'SYSPRINT' (for output); if this fails, then (for output only) the string '/M' is used. If the string begins with '.' it is assumed to be a file title and is retrieved with read access, or created and retained, from/in the directory 'current dir'. If the string is '/M' then the book corresponds to the main output (G9) or input (G10) of the process. If the string is '/A' on 'standout channel' an anonymous segment is created for output; this segment

will be available in NO after calling 'close' or 'sysclose'. If otherwise the string begins with '/' then it should specify a route name known to IOC, such as /LP or /TP or /TR1 or /TT2 and a stream to/from the corresponding route is obtained. If the string commences with a decimal digit, then under the 'command' interface the string given as value of the corresponding serial parameter is used, otherwise the corresponding capability in the A capability segment. If the string begins with a letter then it is treated (if under the 'command' interface) as the name of a keyword parameter, and the corresponding string value is used.

Any other string will cause 'bad open idf error' to be yielded.

The value yielded is 0, or a return code from PARMS, or a standard negative fault number, and can be presented to 'error message'. A suitable value is always assigned to 'sf'.

sysclose =(SYSFILE sf)BOOL:
see manual. Any partially written line is output. Any argument (even 'SKIP') is acceptable.

sysendline =(SYSFILE sf)BOOL:
see manual. For input, the next line is not physically read from the stream until the next call of 'sysfileended', 'sysendline', or any routine actually requiring the line.

sysfileopen =(SYSFILE sf)BOOL:
whether 'sf' is currently open (i.e. the call of 'sysopen' yielded 0 and 'sysclose' has not been called for 'sf')

sysfileended =(SYSFILE sf)BOOL:
see manual

syslineended =(SYSFILE sf)BOOL:
see manual

sysreadmood =(SYSFILE sf)BOOL:
see manual

sysmaxpos =(SYSFILE sf)INT:

```
                                    see manual

syscharpos          =(SYSFILE sf)INT:
                                    see manual

syssetpos           =(SYSFILE sf, INT i)BOOL:
                                    see manual

sysmovepos          =(SYSFILE sf, INT i)BOOL:
                                    see manual

syscharsleft        =(SYSFILE sf)INT:
                                    see manual

syslinepos          =(SYSFILE sf)INT:
                                    see manual

syswritechar        =(SYSFILE sf, CHAR c)BOOL:
                                    see manual

force upper case REF BOOL           When TRUE, causes 'sysreadchar' to
                                    convert all alphabetic characters into
                                    upper case. Initialised by 'USE' to
                                    FALSE.

sysreadchar         =(SYSFILE sf, REF CHAR c)BOOL:
                                    see manual, 'sysendline'  and 'force
                                    upper case'.
```

3.3.6 User-level transput

```
standin channel  CHANNEL            see manual and 'sysopen'
standout channel CHANNEL            see manual and 'sysopen'
parameter channel CHANNEL           see manual and 'sysopen'
string channel   CHANNEL            see 'sysopen'

print            TRANSOUT           see manual
read             TRANSIN            see manual

FILE             mode               see manual

chan             =(REF FILE f)CHANNEL:
                                    see manual

make term        =(REF FILE f, STRING s)VOID:
                                    see manual

on logical file end =(REF FILE f, PROC(REF FILE)BOOL p)VOID:
                                    see manual
```

```
on physical file end =(REF FILE f, PROC(REF FILE)BOOL p)VOID:
                                see manual

on line end      =(REF FILE f, PROC(REF FILE)BOOL p)VOID:
                                see manual

on value error   =(REF FILE f, PROC(REF FILE)BOOL p)VOID:
                                see manual

on char error    =(REF FILE f, PROC(REF FILE, REF CHAR)BOOL p)VOID:
                                see manual

char number      =(REF FILE f)INT:
                                see manual

line number      =(REF FILE f)INT:
                                see manual

file open        =(REF FILE f)BOOL:
                                calls 'sysfileopen(b OF f)'

set char number  =(REF FILE f, INT i)BOOL:
                                see manual

open             =(REF FILE f, STRING s, CHANNEL c)INT:
                                see manual and 'sysopen'

file from slot   =(REF FILE f, SLOT s, CHANNEL c)VOID:
                                analogous to 'open' and 'sysfile from
                                slot'

close            =(REF FILE f)VOID:
                                see manual and 'sysclose'; must be
                                given a FILE which was created by
                                'open' or 'file from slot'.

scratch          =(REF FILE f)VOID:
                                as 'close', but does its best to
                                prevent any output reaching its
                                destination.

newline          =(REF FILE f)VOID:
                                see manual and 'sysendline'

set file state   =(REF FILE f, INT i)VOID:
                                set the stream state of the
                                corresponding stream.  Will fault if
                                the file was opened on 'string channel'
                                or 'parameter channel'

endrec           =(REF FILE f)VOID:
                                as 'newline', but with the stream state
                                set temporarily to 'sameline state'
                                Note that this can be called  from
```

'print'

overprint	=(REF FILE f)VOID:	as 'endrec' but with 'overprint state'
newpage	=(REF FILE f)VOID:	as 'endrec' but with 'newpage state'
space	=(REF FILE f)VOID:	see manual
backspace	=(REF FILE f)VOID:	see manual
readchar	PROC CHAR	see manual, 'sysendline' and 'force upper case'
readint	PROC INT	see manual
readbool	PROC BOOL	see manual, 'flip' and 'flop'
readbits	PROC BITS	see manual
readbytes	PROC BYTES	see manual
readstring	PROC STRING	see manual
printchar	=(CHAR c)VOID:	see manual
printint	=(INT i)VOID:	see manual
printbool	=(BOOL b)VOID:	see manual
bitsradix	REF INT	see 'printbits'; initialised to 0
printbits	=(BITS b)VOID:	If 'bits radix' is 0, see manual. Otherwise prints 'b' as a sequence of digits to base 2 (for bitsradix=2), or 4 (bitsradix=4) or 8 (bitsradix=8) or 16 (otherwise). Leading zeroes are suppressed.
hexdigit	[]CHAR	When subscripted with 0 to 15, yields the corresponding digit ('0' to '9' or 'A' to 'F')
printhex	=(BITS b)VOID:	prints 'b' as an 8-digit hex number.
printbytes	=(BYTES b)VOID:	see manual
printstring	=(STRING s)VOID:	see manual
standout	REF FILE	see manual. The library attempts to open 'standout' on 'standout channel'

		using "".
standin	REF FILE	see manual. The library attempts to open 'standin' on 'standin channel' using "".

3.3.7 Fault and attention handling

errout	FILE	The value referred to by 'standout' after it has been presented to a call of 'open' by the library
error message	(INT i)VOID:	prints, on standout, the string corresponding to fault number 'i', as provided by the FAULT program (G7).
a68c error	=(STRING s, INT i)VOID:	The USE level of the library assigns this routine to 'runtime error'. Calls 'return fault(i)', assigns 'errout' to 'standout', prints a message for the fault number (or 's' if this is not ""), calls 'return fault (ABS16r80000000)', calls 'backtrace', sets return code to 'i', and jumps to 'stop'.
backtrace	PROC VOID	prints on 'standout' a list of the currently active routines.
a68c attention	PROC VOID	The USE level of the library assigns this routine to 'attention routine'. It calls 'return fault(previous error)', assigns 'errout' to 'standout'. If the attention can be cleared with a null capability (i.e. the attention was level 'I') it does so and calls 'backtrace', otherwise it calls 'scratch(standout)'. In either case it then jumps to 'stop'
stop	label	set after 'PRELUDE USER'

3.4 Definitions provided by MATH

maxreal	REAL	see manual; exponent 127, mantissa 16r7fffff; about 1e39.
smallreal	REAL	see manual; exponent -22, mantissa 16r400001; about 1.19e-7
pi	REAL	see manual

3.4.1 Input and output for REAL and COMPL

readreal	PROC REAL	see manual; exponent characters can be 'E' or 'e' or '\'.
readcompl	PROC COMPL	see manual; plus-i-times characters can be 'I' or 'i'.
NUMBER	mode	UNION(INT,REAL)
printfixed	=(NUMBER n, INT w,a)BOOL:	see manual
printfloat	=(NUMBER n, INT w,a,e)VOID:	see manual
printwhole	=(NUMBER n, INT w)BOOL:	see manual
printreal	=(REAL r)VOID:	see manual
printcompl	=(COMPL z)VOID:	see manual
fixed	=(NUMBER n, INT w,a)STRING:	see manual
float	=(NUMBER n, INT w,a,e)STRING:	see manual
whole	=(NUMBER n, INT w)STRING:	see manual

3.4.2 Mathematical subroutines

sqrt	=(REAL x)REAL:	see manual; calls 'runtime error' for negative arguments
exp	=(REAL x)REAL:	see manual; calls 'runtime error' if result would cause overflow
ln	=(REAL x)REAL:	see manual; calls 'runtime error' for negative arguments
sin	=(REAL x)REAL:	see manual
cos	=(REAL x)REAL:	see manual
tan	=(REAL x)REAL:	see manual; calls 'runtime error' for arguments near singularities of tan
arcsin	=(REAL x)REAL:	see manual; calls 'runtime error' if x>1 or x<-1
arccos	=(REAL x)REAL:	see manual; calls 'runtime error' if x>1 or x<-1
arctan	=(REAL x)REAL:	see manual

3.4.3 Random number generators

SYSRANDOM	mode	see manual
RANDOMDATA	mode	see manual
get random	=(INT i)RANDOMDATA:	see manual
save random	PROC RANDOMDATA	see manual
change random	=(RANDOMDATA r)VOID:	see manual
random	PROC REAL	see manual
normal random	=(REAL m,s)REAL:	see manual

Appendix Y

'Storage Management for Algol68'

Storage Management for ALGOL68

A. D. Birrell

Abstract This paper describes some of the techniques which can be used for managing the run time storage required for an ALGOL68 program. The emphasis is on stack storage, since garbage collection techniques would require another paper. The problems caused by some ALGOL68 constructs are described; the solutions given are mainly those adopted in the Cambridge ALGOL68C system.

1 Representation of objects.

ALGOL68 is a language concerned with internal objects and operations upon them. In designing the storage management for ALGOL68 one of the first questions to be faced is how to represent these objects. In other words, when a value of some mode (data type) is assigned, or yielded, what bit patterns are physically moved around the store of the machine.

For some modes, the choice of representation is straightforward:

int => appropriate (machine-dependent) bit pattern - typically a single word.

char => single byte (if possible).

real => floating point number.

Other modes can be built out of simpler parts:

struct(...) => concatenation of the fields, possibly with gaps for boundary alignment.

union(...) => (marker, value)

routine => (entry address, environment pointer)

Note that although the ALGOL68 report does not talk of values of mode union(...), a value which has been united is universally represented as the value with a small marker indicating its mode. The 'environment pointer' in a routine is used in addressing items in blocks outside the routine - this is considered later.

It should be pointed out here that it is possible to achieve considerable simplification of many of our problems by an indirection. By representing values of the more complicated modes by a pointer to data in a global storage area, the whole stack organization is simplified; this substitutes the problems

of managing the global storage area for the problems described below. I believe such a technique has been adopted in the Carnegie-Mellon implementation of ALGOL68S.

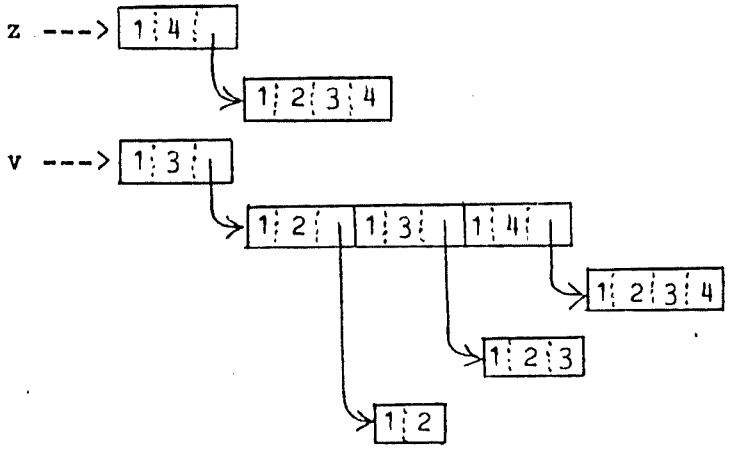
The representation of multiple values is a cause of many problems. There are several causes. Firstly, we must always know the bounds of an array (multiple value), so that we can generate code to copy the array, and for array bound checking. Often, situations arise such that we cannot know these bounds at compile time and so must store them as part of the run time representation. Secondly, the facility of trimming a multiple value allows the program to manipulate sub-arrays. Unless we copy the elements of an array when we trim (which seems needlessly expensive), we must store separately from the elements a block containing the bounds of, and a pointer to, the elements. Thirdly, when subscripting an array we require a uniform distance between the addresses of the elements. When the elements are themselves arrays, we must carefully consider how to achieve this.

For example:

```
[][]int v= ( []int x=(1,2)
                   y=(1,2,3),
                   z=(1,2,3,4);
               (x,y,z)
           );
```

We require that (address for v[1]) - (address for v[2])
= (address for v[2]) - (address for v[3])

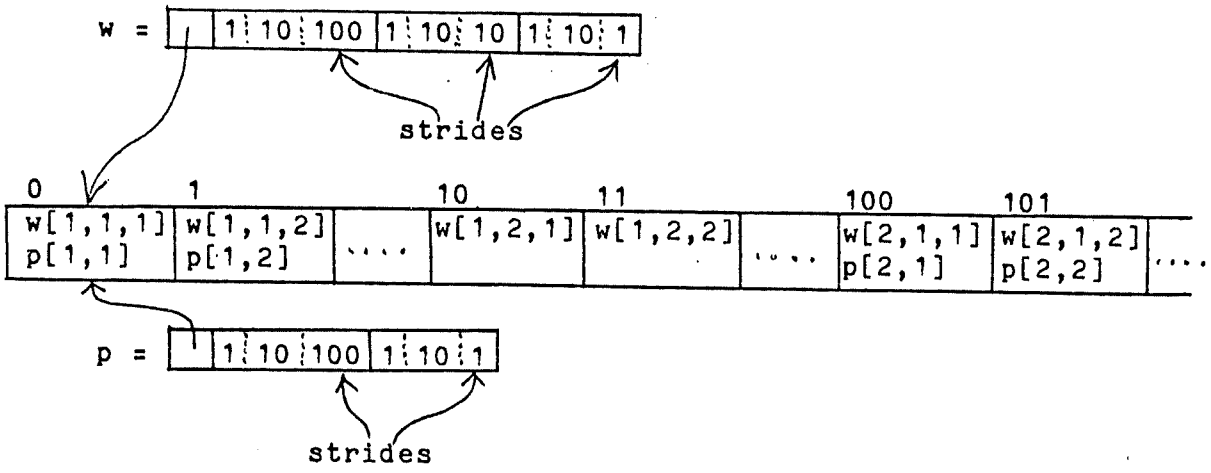
All these requirements can be satisfied by using a descriptor containing bounds and a pointer to the elements. Thus:



Additionally, the descriptor contains a stride for each dimension, to indicate the spacing between elements. For example, if we have

```
[,,]int w = (......);
[,]int p = w[,1,];
```

then the representations of 'w' and 'p' are:



A final step in the representation of such objects is usually that the pointer, instead of being the address of the first physical element, is the address which would be that of the element whose subscript is 0 in each dimension, even if this is not physically part of the array (or even a legal address). This simplifies subscripting, since the computation no longer involves the bounds (unless for checking purposes).

Thus in general objects involving array elements can be complicated tree structures. When such an object is assigned, or storage for it is generated, we must generate code to manage such tree structures. Innocuous looking declarations or assignments can generate considerable tracts of code, and the presence of such objects is the major reason for complexity in our stack management. In future we will term the first level of such an object the static part of the object (its size is known at compile time); the remainder of the object is the dynamic part (its size may not be known until run time).

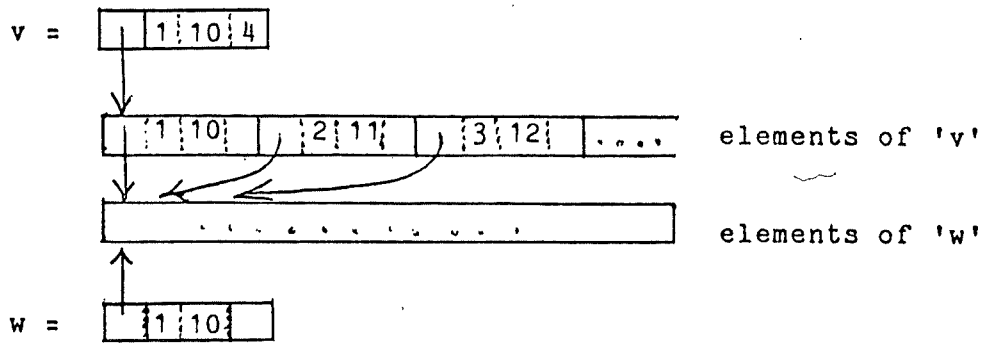
The representation of names, that is objects whose mode is of the form ref amode, is mainly straightforward - the address of an amode object. However, a complication arises if amode is of the form [...]bmode. This complication is caused by trimming; the scope of the name yielded by trimming is the same as that of the name being trimmed. Thus the descriptor produced can

be required to exist longer than the block in which the trimming occurs, and so cannot be allocated on the present stack frame. For example

```
[1:10] ref[] int v;
[1:20] int w;
for i to 10 do int j; ... ; v[i]:=w[i:i+10 at i] od;
```

Here, the descriptors for the arrays referred to by the elements of 'w' are created when trimming 'w' inside the loop, but their storage cannot be allocated at this point. For this reason, when allocating store for an object whose mode is of the

form ref[....]bmode, we also allocate store for a descriptor. For example the declaration of 'v' allocates store for 10 extra descriptors. An immediate optimisation is to represent such names as the descriptor, rather than its address. For example,



Note that this causes the implementor some tedium, since ref[...]bmode must always be handled as a special case. It also causes minor complications in handling identity relations for objects of such modes.

2 The stack

The stack organisation we will develop is based on the conventional ALGOL60 stack, which is summarised here.

The stack consists of a number of frames, one for each block which has been entered but has not yet terminated.

At any time there is a current display. A display consists of a number of display registers (or levels), each containing the address of some frame. The current display consists of registers addressing the current and each textually enclosing block.

To access a word of the stack, we use an address of the form

[display register] + offset

where the display register (but not its contents) and the offset are known at compile time.

Due to recursion, there may be more than one frame for each block.

On entry to a block, a new frame is started and an extra display register added to the display in order to address the frame.

On entry to a procedure, the complete display is reset to correspond to the block in which the procedure (not the call) occurred. To allow this operation, the representation of a

routine value contains an environment pointer indicating the values to load as the new display. This routine value is known as a closure; due to recursion, there may be different closures for a single routine text.

On exit from a block or procedure, the display is restored to its previous value.

In ALGOL68, there are several categories of data we wish to store on the stack.

a) A heading containing the address of the previous frame, subroutine link, information for setting up the display, etc.

b) values for definitions: for 'int i' the value referred to, for 'real x = random' the value itself.

c) anonymous results created during elaboration of the block.

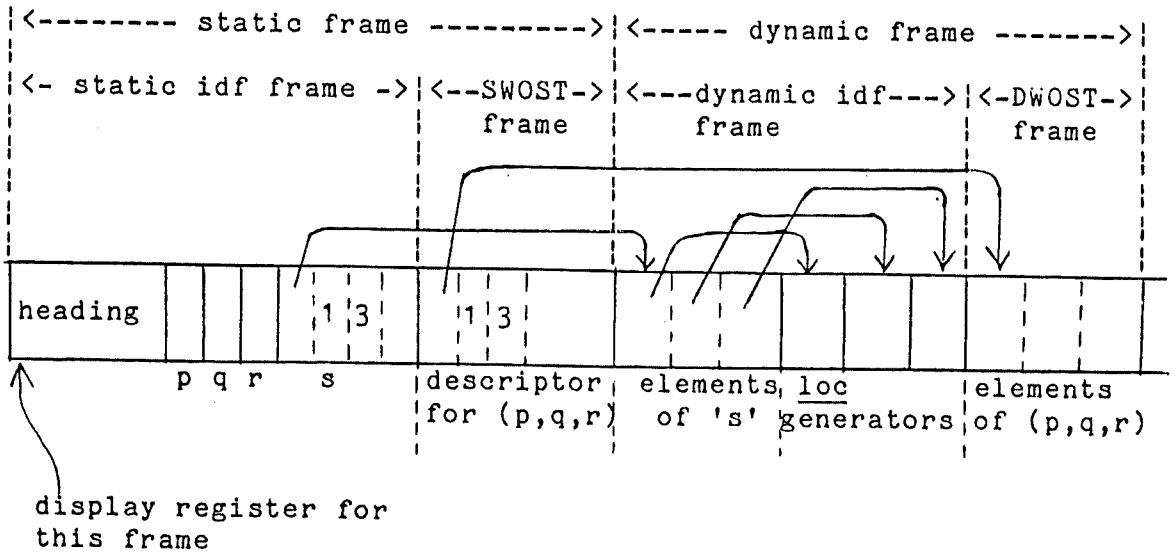
d) array elements

e) storage for explicit loc generators.

Of these (a), (b) and (c) are straightforward, but (d) and (e) are difficult since the amount of storage required may be large and may not be known at compile time.

It is common practice to store (a), (b), (c) at the start of the frame, with (d) and (e) at the end. This has two advantages: the offsets written in instructions are smaller (many machines place severe limitations on such offsets), and we always know the offsets for identifiers at compile time. We can thus consider each frame as being divided into a static frame containing (a), (b), (c), and a dynamic frame (possibly empty) containing (d) and (e). Further, it is convenient to treat (c) separately as the SWOST (static working stack) frame, calling the remainder of the static frame the static idf frame. Similarly, we can sub-divide the dynamic frame into the DWOST frame containing the dynamic parts of SWOST objects, and the dynamic idf frame. For example:

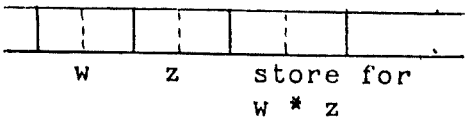
```
begin
   int p, q, r, [1:3]ref int s;
   for i to 3 do s[i] := loc int od;
   .
   .
   (p,q,r) #row display#
   .
   .
end
```



Management of the static frames is mainly straightforward. Storage is allocated on the static idf frame only at definitions. The structure of the language is such that at a definition, none of the anonymous results which have been created in the current block still exist, so the SWOST part of the frame is empty. Thus the static idf frame is contiguous storage starting at the end of the heading area. The SWOST, then, is always placed at the end of the static idf frame. However, situations can arise which produce holes in the SWOST; these are typically when we are constrained to produce the result of some action without overwriting its parameters. An example might be:

```
compl w := ...,  z := ..  ;

...  w * z ....
```



It is always possible to avoid such holes by copying, and with sufficient care most holes can be avoided without copying. In ALGOL68C, we decided that the extra expense of allowing the holes was not great enough to justify the complexity (or expense, if we copy) of avoiding them. Accordingly, holes are allowed to occur on SWOST; however since SWOST for a block is always empty at a semicolon and before a definition, such holes are generally of short duration. It should be noted that all static frame offsets are known at compile time. No run time management is required.

Before considering the management of dynamic frames, we should look at an optimisation available to us. Within a single procedure, we know at compile time all the offsets inside each

static frame. If, then, we place all static frames first, followed by the dynamic frames, we will be able to address all the static frames with a single display register, pointing to the base of the first static frame. This optimisation is often described as having one frame per procedure, but this is not really an accurate description. In terms of when store is allocated and recovered, and in the interleaving of static idf frames with SWOST, we are still running one frame per block. The only alteration is to omit some display registers, and move the dynamic frames. The dynamic frames are still, in every sense, one per block. This optimisation gives us several gains. The number of display levels is drastically reduced, being limited to the textual nesting depth of procedures - in practice, we have never encountered depths greater than 5, although ALGOL68C allows for 64. The number of display registers required is in fact less than the textual nesting depth, since if an enclosing frame is not referenced from inside a procedure, it can be omitted from the display. (This is allowed by, and required by, the rules on the scope of routine values.) It is possible, instead of keeping the complete display, to keep only a pointer to the static frames of the current procedure, and store there a pointer to the frames of the enclosing procedure. Then accessing a frame of an enclosing procedure is achieved by indirecting down this static chain. Since the number of levels on the static chain is typically less than 5, these indirections never go very far. In ALGOL68C we maintain a pointer to the outermost level, and one to the current procedure; in this way only about 2% of static frame accesses require indirection down the static chain (and then, less than 4 indirections). These indirections can be further reduced by remembering which registers currently address outer levels. It should be noted that, using the above techniques, if a block requires no dynamic frame then no run time cost is incurred by block entry or exit. This means that the programmer can freely use begin/end for structuring his program without worrying about extra code being generated. The environment pointer of a routine is now a pointer to the frames of the enclosing routine, and is used for the static chain when the body of the routine is elaborated.

The mechanism used by ALGOL68C for run time management of dynamic frames is unusual. At first sight it appears too complicated, but by paying a little in in conceptual complexity we have attempted to minimize run time actions, and as far as possible to eliminate them completely for blocks or procedures with no dynamic frame. We define a drange to be any range (block) which, excluding inner ranges, allocates storage on a dynamic frame, and a droutine to be any routine containing a drange. For each dynamic frame we will require a pointer to the top of that frame - this we call the dsmd (dynamic stack management data) for the frame. ALGOL68C always keeps the dsmd stored on the static frame at an address known as the dsma - as will be seen, this simplifies our run time actions. With the stack organization as described above, we would perform the following actions; these will be modified in the light of changes to be described later.

a) On entry to the outermost drange of a droutine, we allocate a dsma and initialise its dsmd to the top of the static frames.

b) On entry to an inner drange, we allocate a new dsma and initialise its dsmd to the previous dsmd.

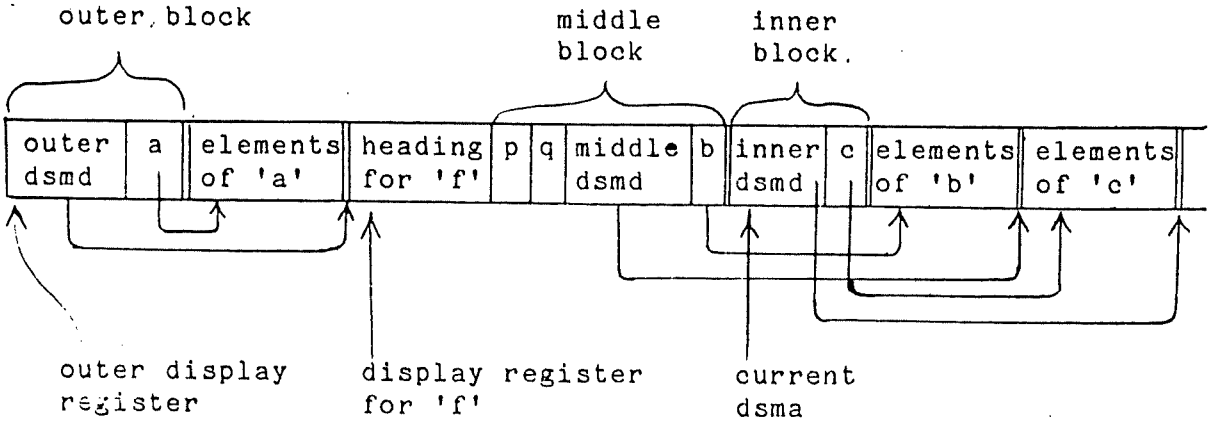
c) To allocate storage on a dynamic frame, we use and update the current dsmd (as addressed by the current dsma).

d) On exit from an inner drange, we revert to the outer dsma. Note that this is not a run time action, since we know the dsma (as a static frame offset) at compile time.

This mechanism is simpler at run time than the alternative of keeping a single dsmd and preserving/restoring it; it is the only tenable mechanism for the stack organization described below. Under this scheme, jumps present no problem - at the target label, we revert to the appropriate dsma. It is difficult to produce an alternative scheme which does not have to preserve the dsmd at every call in case there is a jump out of a drange in some inner routine; such preservation has the effect that you pay for dynamic frames even if you do not use them. An example of our stack organization would now be:

```
begin
   [1:10]int a;
   proc f = int:
      begin
         [1:10]int b;
         int p, q;
         begin
            [1:10]int c;
            .
            .
            .
         end
      end;
   f
end
```

The stack after declaring 'c' might be:



Considerable difficulty is presented by argument passing, when the arguments have dynamic parts allocated during their elaboration. For example:

```
( random < 0.5 | f | g )(a, loc[x:y]int, b)
```

Firstly, consider which display register to use for addressing the static parts of the arguments while inside the called routine.

a) Using the display register of the calling routine is not possible, since inside the called routine we would not know the offsets for the arguments.

b) We could use a separate display register solely for the arguments, but this would double the number of display levels. (This solution is quite commonly adopted by other implementors.)

c) The only other possibility is to address the static parts of the arguments using the display register of the called routine.

Assuming choice (c), then, we must consider where to place the dynamic frame allocated for the arguments.

a) We cannot place it before the static parts of the arguments, since we do not yet know its size.

b) We can place it after the static parts of the arguments only if we place it after the other static frames of the called routine, but in ALGOL68C we do not know the size of the called routine's static frames while we are elaborating the call. Some implementors do arrange to maintain this information at run time.

By this stage in the design of the stack we have accumulated (albeit implicitly) several problems.

1) Where to place the dynamic parts pf arguments.

2) How, at the calling end, to address the static parts of the arguments since we do not at that stage have a display register for them.

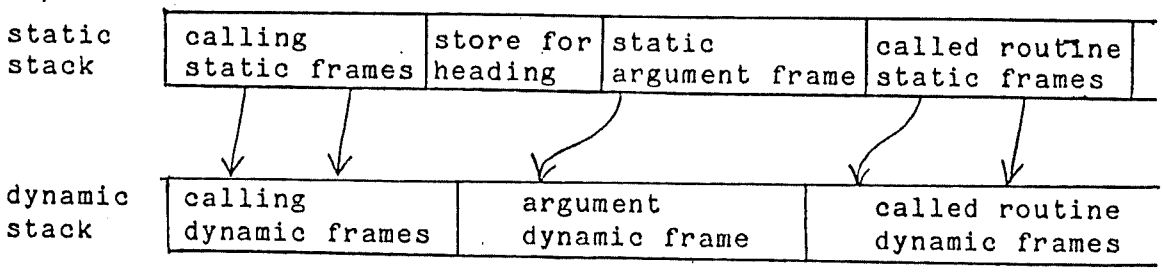
3) Storage is wasted since dynamic frames start at the high water mark of the static frames of the routine.

4) The ALGOL68C separate compilation mechanism would require a display register for each segment.

5) Any proposed solution of (1) to (4) with this form of stack organisation appears to be much too complicated.

In ALGOL68C, to solve these problems we made a drastic re-arrangement. Instead of continuing attempts to organize a single stack, we split the storage into two independent stacks. The static stack contains all static frames, and is addressed by display registers and offsets; the dynamic stack contains all dynamic frames, and is referred to from the static stack.

We can now have a simple solution to the argument passing problem. The static frame for the arguments is addressed at the calling end using the display register of the calling routine - since there are no intervening dynamic frames we know all the offsets at compile time. Inside the called routine, the arguments form the first static frame. The dynamic frame (if any) for the aguments is treated as any other dynamic frame, with no additional problems; if there is such a dynamic frame, the arguments will constitute a drange. Thus:



The wasted static frame storage is eliminated, and we do not need a separate display register for separately compiled segments.

With this revised organisation, we must revise the actions to be performed for managing the dynamic frames. Since we no longer know at compile time the base for the first dynamic frame of a droutine, this information must be passed with the call. Since we do not know at the call whether the called routine is, or will call, a droutine, the information must be passed with every call. To avoid this causing a run time action on every call, we always have the current dsma available at run time (in a particular register, say, or in a fixed store location). The actions to be performed are then:

a) On entry to any drange, allocate a new dsma, initialise its dsmd to the previous dsmd, and reset the run time dsma.

b) On exit from any drange, restore the run time dsma to its previous value.

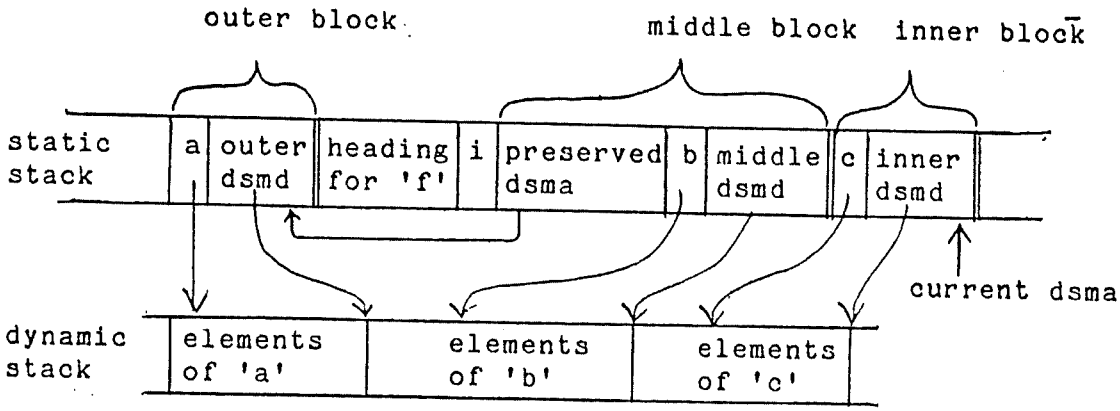
c) To permit (b) in the outer drange of a droutine, preserve the dsma on entry to a droutine.

To allow for labels and jumps, we include as a droutine any routine containing a label; at a label we reset the dsma to the appropriate value. Note that these arrangements still satisfy the dictum that if you don't use the dynamic stack then you

shouldn't pay for it (except at labels). Our example might now be as follows:

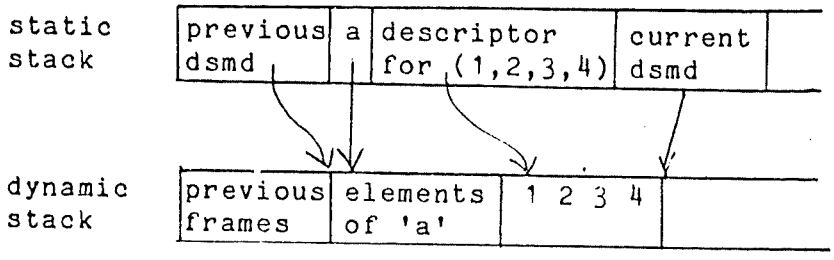
```
begin
   [1:10]int a;
   proc f = (int i)int:
      begin
         [1:10]int b;
         begin
            [1:10]int c;
            ( i <= 1 | 1 | i * f(i-1) )
         end
      end;
   print( f(1) )
end
```

The stacks after declaring 'c' would be:



One serious problem remains in our description of the stacks - this is the yielding of a result from a block or a procedure. The difficulty is that the result is constructed on stack frames, inside the block or procedure, which are about to be relinquished. For example:

```
begin
   [1:1000]int a;
   .
   .
   .
   (1,2,3,4)
end
```



There are basically two possibilities: either copy the value onto the outer SWOST and DWOST, or delay relinquishing the frames. However, copying can be very expensive (and sometimes very difficult), while delaying relinquishment can waste vast amounts of storage. An extensive analysis of this problem has been given by Branquart, and an algorithm which assists in copying has been given by Meertens. It is certainly best to delay the decision as to whether to copy or to avoid relinquishing, until as late as possible. At present ALGOL68C does not recover the storage in this situation - this is hardly satisfactory. With sufficient care, it is possible to achieve satisfactory results even in extreme examples such as:

```
op * = ([]int x,y)[]int: ... ,
   + = ([]int p,q)[]int: ... ;
[1:100]int a,b,c;

1: a * b + ( ... | c | goto 1 );
```

In particular, the compiler can treat some constructs as if they were dranges (though not actually ranges) to aid the recovery of dynamic stack.

3 Summary

The power and flexibility of the constructs available in ALGOL68 lead to considerable complexity in the objects being manipulated and in the management of the storage for them. By dividing the stack into two independent stacks we greatly simplify these problems, although on an unsegmented machine the need for three storage areas (the stacks and the heap) presents extra difficulties. An alternative solution, often adopted, is to place dynamic parts on the heap in times of difficulty - this we still must do when assigning objects of modes such as union([]int,[]real) - but this approach was discarded, because it is expensive and because it uses the heap behind the programmer's back. A full discussion of the problems of result passing would be outwith the scope of this paper, as are the techniques available for flex.

4 Acknowledgements

The ALGOL68C compiler was developed in Cambridge by a team led, until January 1975, by S.R.Bourne. Since then it has been maintained and further developed by C.J.Cheney for the University of Cambridge Computing Service. Throughout the development of the compiler much advice and much work has been given by M.J.T.Guy, I.Walker, and myself. Much of the work has been funded by the Science Research Council, and the maintenance is now supported by the Computer Board. Help has been given by our various users and by I.Wand of York University Much of our terminology, and some of the ideas, are based on those of P.Branquart.

5 References

[1] A. van Wijngaarden, et al, "Revised Report on the Algorithmic Language ALGOL 68", Acta Informatica, Vol. 5, pts 1,2,3, (1975).

[2] S.R.Bourne, A.D.Birrell, I.Walker. "ALGOL68C Reference Manual", Cambridge University Computer Laboratory, (1975).

[3] P.Branquart, et al, "An Optimized Translation Process and its Application to ALGOL 68", Report R204, M.B.L.E., Brussels, (1974).

[4] P.Knueven, "The Foundation of a Flexible Run-time System for ALGOL 68S", in "Experience with ALGOL 68", Proceedings of the Liverpool University Conference, April 1975, Ed. C.C.Charlton and P.H.Lang.

[5] L.G.L.T.Meertens, "A Space-saving Technique for Assigning ALGOL 68 Multiple Values", Mathematisch Centrum, Amsterdam, (1976).