1995

An Ada-like language to facilitate reliable coding of low cost embedded systems

Michael Collins
Edith Cowan University

Edith Cowan University

Copyright Warning

"An Ada-like Language to Facilitate Reliable Coding of Low Cost Embedded Systems"

by: Michael Collins

A dissertation to be submitted in partial fulfilment of the requirements for the degree of

Bachelor of Applied Science (Information Science) Honours.

Department of Computer Science,
School of Information Technology and Mathematics,
Edith Cowan University,
Perth, Western Australia.

Supervisor: Dr T. J. O'Neill.

February 1995

Michael Collins.
Student Number 0912143

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

Table of Contents

Abstract

Due to a lack of operating system (O/S) support, it is more difficult to develop programs for embedded systems than for workstations. For those developing on a low budget, the problem is often further compounded by the necessity of using inappropriate, O/S dependent, compilers.

This study attempts to ascertain those elements of a High Level Language (HLL) which are absolutely necessary and implementable to produce reliable, efficient, embedded programs without the benefit of a large budget. The study is based upon the Ada philosophy as the Ada language incorporates many desirable features for modelling real-world problems in terms of embedded solutions.

By implication, the research provides a small step towards an increased availability of low cost tools to assist in the development of reliable and efficient code for use in medium performance embedded systems.

Declaration

I declare that this thesis does not incorporate without acknowledgment any material previously submitted for a degree in any institution of higher education, and that, to the best of my knowledge and belief, it does not contain any material previously published or written by any other person except where due acknowledgment is made.

Signature:

Date: 16th June 1995

Acknowledgments

No significant project can be completed by one person. Accordingly, I would like to thank all those people who helped and encouraged me with this project. However, there are some special people to whom I wish to express particular gratitude.

Thank you Sue, Ben and Beccy, for being there when I was here.

Thank you Dr Thomas O'Neill, for you can inspire the transformation of a mediocre passage of text into something that is worth writing.

Thank you C.C.S. Communications Pty. Ltd., for supplying the code-locator that is used in this project.

1 Introduction

This chapter contains an overview of embedded systems in general, and the problems of programming for medium performance embedded systems in particular. The aims of the project are described and a synopsis of the remainder of the project is given.

An embedded computer system, suggests Zave (1982), is surrounded by a larger non-computer system and is frequently subject to operational constraints on time and space. Embedded systems are used to control many aspects of everyday life: from automobile engines to those in fighter-planes; from microwave ovens to production lines; from domestic reticulation systems to agricultural crop spraying; and, from model train sets to national railway signalling systems. Their computing power ranges from that of a simple eight bit microcontroller capable of simple arithmetic and logical evaluations right up to state of the art thirty-two bit parallel processors able to perform the fastest calculations. In every case, the embedded systems are expected to perform to extremely high standards of reliability, often under adverse operating conditions and within exacting response times. Safety requirements may at times appear conflicting with military weapon systems being "designed to be very safe in peacetime, but lethal in time of war!" (Cullyer, 1993).

For software engineers active at the high end of the embedded arena (for example, within the relatively large budgets of the aerospace industry) Mann (1992) suggests that development environments are available which are every bit as sophisticated and productive as those for developing workstation hosted software. Where budgets are more limited, however, the choice and availability of development tools are often restrictive to the implementor. This can result in inefficiencies within system development and maintenance, the consequences of which can be at best expensive, or at worst fatal to the system or its users.

There is available a range of general purpose, medium performance, microcontroller units based upon the standard sixteen bit microprocessor architectures that are found within personal computers (PCs). These microcontrollers lend themselves well to embedded development on a tight budget due to their significant advantage of being able to execute code generated by standard, relatively cheap, PC compilers (e.g. Microsoft C). However,

embedded systems often exhibit a lack of O/S and/or concurrency support for such code and this disadvantages the developmental process. It may also be difficult, and/or expensive, to obtain program debuggers which will operate meaningfully within a real-time embedded environment, thus prolonging test and verification phases.

Ada is a concurrent, strongly typed, language which was created chiefly for use in large scale embedded systems; yet for efficiency, familiarity and availability reasons, the C language remains the primary development language of medium performance embedded systems. However, C includes no natural expression of concurrency and exhibits weak typing which can let many program errors persist until run time, when they become more difficult to remedy.

Sommerville (1990) suggests in his discussion on debugging that the later a mistake is corrected in a software development cycle, the more expensive the correction becomes. Strong type-checking, as found in Ada, can detect many errors before run-time.

The study examines current developmental practice within the context of medium performance embedded systems. It aims to establish that the existing programming tools and language elements of Ada and C may be combined effectively to reduce the incidence of error entry into embedded systems, thus increasing reliability. This is accomplished by establishing a subset of C (termed "Safe C" throughout this document) which, requiring no O/S support, may be used to safely express embedded programs written in an Ada-like syntax.

Chapter two elaborates the problem. The background of the study is discussed: the embedded system is defined and examples provided. The type of embedded system targeted in this study is explained and identified by its prominent features and abilities. Problems faced by software engineers in providing executable code for such systems are outlined in terms of complexity and economics, with reference to conventional languages and tools. Finally the significance of this study in the real world, and its framework, are outlined.

Chapter three takes the form of a review of the relevant literature. The work of others, illustrated in the firm foundation of text books and augmented by the documented research and experiences of others in papers and articles, forms a basis of guidance and justification for the approach taken and substance of this project. A list of desirable features for Real-Time languages is discussed, together with the approaches of different multi-tasking models and the use of C as an intermediate target.

Chapter four builds upon the foundations outlined in the Literature Review (chapter three), combining these with the needs identified in the Study Background (chapter two) to form the basis for the pilot project design. The design of the pilot study is developed, together with its methods and verifiable outcomes. The discussion addresses the methods in terms of the list of desirable features discussed in chapter three.

Chapter five presents the findings and results of the pilot implementation. Selection criteria for a suitable target system are discussed and an overview of the features offered by the selected target is given. The test program, which incorporates a variation of the producer-consumer relationship designed to demonstrate treatment of the desirable features, is examined. Suitable extracts from the Ada-like source for the test program are presented, together with their "Safe C" productions. These are used to provide answers to the research questions.

Chapter six concludes the project. A summary is given of its beginnings and initial aims; the manner in which the project framework was arrived at; the design criteria for the pilot implementation; and, the resultant product of the Ada-like language to "Safe C" translator. Finally, implications are discussed for current practice and future research.

This document concludes with several appendices, which are used to provide clarification and amplification of significant areas of the study, and a section where end text references for documents used to support th e study are given: namely

Appendix A consists of a comprehensive glossary of terms used in this document;

Appendix B presents a description of an Ada-like Language;

Appendix C lists those elements of Microsoft C Version 6 which have been found to satisfy the requirements for "Safe C" as used in this project;

Appendix D demonstrates how Ada's generic packages facilitated the re-use of major components in the translator;

Appendix E illustrates how pre-processor macros were used to achieve inter-compiler compatibility;

Appendix F presents the Ada-like language test program, and its translated productions in "Safe C"; and

Appendix G is a listing of the startup code used to support the test program.

2 The Problem

2.1 Background to the Study

It is generally more difficult, and hence more costly, to engineer programs for embedded systems than for applications of similar complexity on workstations. The extra difficulty is, as Mann (1992, p58) and Aucsmith (1988) suggest, because the embedded program cannot usually take advantage of underlying O/S support. Where a tight budget dictates the use of inappropriate O/S dependent compilers, developers of embedded systems face further complications.

An embedded system is defined as "a computer system which forms part of a larger non-computer system" (Zave 1982). Booch (1987, p15) suggests that although embedded applications are varied, they share a set of common characteristics, which tend to :-

- be long-lived and subject to continuously changing requirements;
- be subject to constraints of time and space, especially for real-time response;
- be highly reliable and fault tolerant, requiring automatic error-recovery (Aucsmith, 1988);
- be "extraordinarily hard to test" (Zave & Yeh 1981);
- exhibit "asynchronous parallelism" (Zave 1982).

The processing power available for embedded systems ranges from simple eight bit microcontrollers (Mazur, 1992), such as could be found controlling a microwave oven, through to thirty-two bit computers (Bhansali, Pflug, Taylor & Wooley, 1991), as used in the avionics industry. Within this range exists a set of low-cost medium-performance embedded systems with which this study is concerned. These are built around a general purpose Central Processing Unit (CPU) core capable of addressing a Megabyte of memory, employing sixteen and thirty-two bit registers and sustaining clock speeds of five to twenty megahertz. Current examples of such CPUs are the Motorola 68000, Intel 80186 and NEC V25, V35, V40 and V50 range. Their general purpose CPU core allows the execution of code normally intended for PCs.

Embedded system programmers are faced with large up-front costs of purpose built Cross-Compilers and In-Circuit Emulators. As an economical alternative they may choose to relocate the code produced by efficient proprietary compilers (such as Microsoft or Borland C), and subsequently use monitor programs to test the resultant code in the target system, as described by Phillips & Rowett (1991, p85). This technique introduces fresh problems, as code produced by such compilers naturally expects to find facilities provided by an O/S. Whilst it is possible to embed an O/S with the application, this imposes additional costs of necessary hardware, software, and distribution licenses and/or royalties; and, like all software, is liable to contain errors. Not all O/Ss may be suitable for the hosting of multitasking applications as portions of their code do not support re-entrancy.

Some embedded systems are still coded, and maintained, in assembly language. Many developers have migrated to the use of C (Mazur 1992). This provides relative improvements in portability and abstraction while still permitting low level control and flexibility. Its portability is demonstrated by recent commercial studies which have concluded that "C in its standard form was the only language fully supported for a range of real-time embedded operating systems" (Velastin, 1991). However, C's freedom is gained from weak typing and the use of pointers, both of which may be dangerous. As Walraet (1989, p140-144) explains, C provides many semantic ambiguities which allow "undisciplined manipulations of data". There is no mandate for exception handling, as returned error values can be freely ignored throughout C programs. Furthermore, C gives no natural expression of concurrency, which is an essential characteristic of real-world problems. Summarised by Velastin (1991), "C is an inherently unsafe language which has inevitably resulted in unsafe practices".

With its many function libraries, C is a capable yet simple language, but notably one which requires an extremely strict programming discipline. C++, as suggested by Phillips et al. (1991, p76) and Stapfer (1992, p72), is also beginning to be employed for embedded system programming. Whilst performing stronger type checking, C++ allows the free inclusion of C syntax and thus requires the same strength of discipline. This can really only be imposed by rigidly following a tight specification. However, a lower level language such as C, with its preoccupation for fine detail and lack of natural concurrency, does not

relate well to specification languages. These need to be "high level enough to be understandable, yet precise enough to define completely a particular class of behaviour" (Swartout & Balzer 1982, p438).

Ada is a language which was developed primarily for large-scale embedded systems (Booch 1987). Its strong typing insists that data types, and operations on them, are known up front, enforcing a rigid programming discipline. It features a rich set of constructs, including concurrency. This facilitates expression at both high and low levels, and consequently eases the translation from a specification mechanism, such as Jackson Structured Development (JSD), to implementation language (Topping & Yeung, 1990). Ada's suitability in this case is further demonstrated by Jackson's release of a JSD to Ada code generation tool (Topping & Yeung, 1991). Further, Ada provides almost unprecedented portability, good maintainability and is now being widely taught at colleges and universities.

In spite of these obvious advantages there is a down side. Ada's concurrency mechanism, called tasking, has been widely criticised (Sims, 1991; Struble & Wagner, 1989; Baker, 1988) for real-time applications. Compilers are expensive, relative to those of C, and their output is seen to be resource hungry and inefficient in terms of executable speed and interrupt response time, as found by Dobler (1992) and Harp (1988). Faced with these, some developers view Ada with reservations (Struble & Wagner 1989).

2.2 Significance of the Study

From the background discussion, the following is apparent. On the one hand there is the C language, with inexpensive and readily available compilers generating efficient relocatable code. However, using C creates a difficulty of verification and any resulting errors may lead to time wasted in testing and debugging. On the other hand, there exists the rich strongly-typed language of Ada, compilers of which can be expensive and can produce code that may be inefficient on medium performance machines.

The study demonstrates the possibility of harnessing the descriptive power, strong typing and concurrency, found in the Ada language and translating this to "Safe C" so that a C compiler can be used to generate efficient verifiable executable code. This affords the practicing embedded-systems engineer a number of advantages :-

- concurrency may be expressed formally;
- strong typing reduces the amount of coding errors which actually get as far as run time;
- the proper use of an Ada-like syntax increases the readability and hence the maintainability of the program;
- extensibility with assembler and C modules will still be available;
- compilation and linking tools will probably already be familiar items.

2.3 Statement of the problem

The study investigates the possibility of using an Ada-like syntax for source code entry and the subsequent translation of the entry to an efficient C format which attempts to suppress coding and logic errors reaching run-time, and which provides a natural expression of concurrency in low-budget medium-performance embedded-system development.

2.4 Research questions to be answered

The main question:-

"Can the desirable features for low cost, medium performance, embedded systems programming be provided in an Ada-like language, and then translated to "Safe C" to achieve reliable run-time efficiency? "

The major components of the above question are :-

a) "What elements of computer languages are desirable to express solutions as low cost, medium performance, embedded systems?"
b) "Can the elements noted in *(a)* be expressed in an Ada-like language?"
c) "Can those language elements be implemented using only verifiable elements of the C language?"

2.5 Summary

Where severe budget constraints apply, development of embedded systems represents an area where verification and expression are difficult using traditional, but widely available, C compilers. The Ada language has been designed to solve problems experienced with large-scale embedded systems, but due to execution inefficiencies, and resource hungry run-time demands, it is still seen as being unsuitable for medium performance embedded systems. This study ascertains that "Safe C" may be used as an efficient intermediate step to represent models expressed using the safety-net of an Ada-like stronger typing.

3 Review of the Literature

3.1 General Literature

From Booch's (1987, p15) definition of an embedded system, outlined in Section 2.1 of this document, we know that embedded systems must frequently work within strict physical and temporal (real-time) constraints. This differs from normal processing requirements, and the needs for this must be serviced accordingly by languages and their compilers. Laplante (1993, pp64-88) provides an excellent discussion of Language Issues for Real-Time systems. In the discussion he identifies run-time memory requirements and speeds of execution and response as being the principal constraints on such a language. He further suggests the following desirable features in a Real-Time language:-

- strong typing;
- flexible parameter passing mechanisms;
- recursion;
- exception handling;
- concurrency;
- elimination of constructs with indeterminate execution times.
- interrupt types (this implies re-entrancy);
- modularity;
- low-level control;

(Cotigny & Ple (1991), based upon their practical experience in CNC machine design, echoed Laplante's list of desirable features, but additionally emphasised the necessity of

- portability

in their criteria of language choice.

Using the extended list of desirable features as a framework, each item will be inspected with respect to relevance to this project:

- **Strong typing**

  Brosgol (1990) states the importance of matching the type to the data, so that the program may realistically model the data of the real world. Where types may be defined in this manner, the trapping of variable mismatches at compile time by the use of strong typing is "widely acknowledged as being a good idea" Sebesta (1989, p122).

Sebesta further suggests that strong typing may be described as the situation where each name in a program is associated with a single type, and that the type is known at compile time.

The general idea of abstraction is seen by Tucker-Taft (1993, p127) to be extended by the object-oriented paradigm so that a given abstraction may effectively have multiple implementations. This is the basis of object-orientation, while still retaining strong typing, as used in Ada-9X.

- **Flexible parameter passing mechanisms**

  If we consider a subprogram frame to be the scope where all of its variables can be manipulated, then a subprogram communicates with the world outside that frame in two main ways: either by changing values globally available to it, or through the passing of parameterised values. An excellent discussion of parameter passing exists in Sebesta (1989, pp264-265), a synopsis of which might be the following:

  Four implementation models are generally available:

  (a) Pass (or Call) by Value (IN-mode): where the value of the actual parameter is copied on the stack to a subprogram. This method provides for inward only communication.

  (b) Pass by Result (OUT mode): no value is taken from the parameter on entry to a subprogram, but just before control is passed back to the subprogram's caller, the variable's value is copied back to a supplied variable on the stack. This method provides for outward only communication.

  (c) Pass by Value-Result (IN-OUT mode): a copy is made on the stack of the variable's inward value to the subprogram and, at subprogram termination, the final value is copied back out to the caller. This method provides for two-way communication but, as with the other two methods, excessive stack space can be used with large variables.

  (d) Pass by Reference (access or pointer method): the address of the variable is passed to the subprogram, which subsequently may manipulate the contents via that address. This method can lead to undesirable side-effects, but serves to minimise stack space.

Note that methods (a), (b), and (c) fall into a general category of "Call-by-value", as actual copies of the value are transferred to/from the called subprogram, and that (d) falls into the category of Call-by-Reference, where "Reference" indicates that the address of the variable is transferred. Excessive copying of entire variables to the stack can adversely affect run-time performance, limiting the ability of an embedded system to respond.

On a lighter note: Niklaus Wirth, founder of Pascal, Modula-2 and Oberon, when asked how he pronounced his name was reputed to have replied "If you call me by name it's *veert*, if you call me by value it's *worth*" (Laplante, 1993, p66).

- **Recursion**

  A recursive subprogram can be self-referential, i.e. it can call itself. Whilst recursion is elegant, Laplante (1993, p68) points out that its use of stack for parameters and local variables can reduce run-time efficiency to the point of endangering performance. He suggests that compilers could combat this by recognising the recursion and then mapping it into an iterative form to increase execution speed and reduce stack dependency, though he knows of none that do. As an alternative, Real-Time Euclid programs prohibit recursion to ensure compile-time knowledge of execution times and storage requirements (Stoyenko & Kligerman, 1986, p943).

- **Exception handling**

  Where a handler is provided to cater for exceptional or abnormal run-time conditions, this permits the program to handle the event and recover gracefully. Recognising that exceptions will occur, however, is a small part of the problem, as the machine must be restored to a stable state before normal execution can occur. The binding, between an exception and its handler, which restores control to a level of normality, can be achieved in two main ways, (Sebesta, 1989, p382) each having its own attendant arguments:

  ◊ If we handle the exception local to its occurrence, then it may be the case that many handlers are required to ensure the correct trapping of all exceptional conditions. This complicates the code, increasing the complexity of the program.

  ◊ If we allow the exception to propagate, for example to the calling environment, then we can reasonably trap several possibilities for exception with one handler. This can result in less complex, more easily validatable code.

  After the exception has occurred and been trapped, the question of to where control is transferred must be addressed. The easiest solution, Sebesta says (1989, p382), is controlled termination of the program. However, an embedded system may not be allowed the privilege of simply terminating, and must be expected to resume program execution reliably. Hoogeboom and Halang (1991), suggest that such behaviour is essential for robustness and that there is "a requirement that the system remains in a predictable state even if the environment does not correspond to the specifications".

  Notable, and well documented, exception handling facilities are exemplified in PL/I (which pioneered the concept), CLU, and Ada. C++ has recently added exceptions to its latest standard, though implementations providing it are difficult to find (Eckel, 1993, p716).

The scope of PL/I's handlers are dynamically bound to the exception, and are therefore not necessarily known at compile-time. Such binding represents an area where program control is non-deterministic. The designers of CLU, noting this, worked according to two major decisions (Liskov & Snyder, 1979):

◊ handlers are statically associated with invocations; and

◊ for simplicity, handlers may be attached only to program statements and not to expressions.

Subprograms in CLU can return in one of two ways: namely, on normal termination by executing a return, and on abnormal termination by signalling an exception. In this way the subprogram is able to return differing types and objects according to its termination condition.

Ada's mechanism partially builds upon that of CLU (Sebesta, 1989, p 392) in that handlers are statically bound, known at compile-time, and may be sited locally to the code where the exception occurred or, by their absence there, allow the exception to propagate and be handled by a more general exception handler in the calling chain. Additionally, Ada provides for five pre-defined exceptions. C++, a relative newcomer to the world of exception handling, has modelled its version, CLU like, as "an alternative return mechanism from functions" (Eckel, 1993, p 717), differing only by the fact that under exception conditions the function can return types other than those declared in the function declaration. C++ functions are provided with destructors which implicitly remove (i.e. de-allocate memory for) objects suitably created during the function execution. Harnessing an exceptional return to the mechanism provided for a normal return ensures that only those destructors necessary to remove objects created up to the exception point are called. In similar fashion to Ada, exceptions can propagate along the calling chain until, if no handler is provided, controlled termination of the C++ program will occur.

- **Concurrency & elimination of constructs with indeterminate execution times**

  Topping and Yeung(1990, p25) place great importance on the ability to handle concurrent activities that occur in embedded environments. It is important, however, to recognise that the presence of concurrency in a language does not signify that Real-Time requirements, in particular determinism, will be met. Gligor & Luckenbaugh (1983) highlight the differences between the requirements for concurrent languages and those for Real-Time languages. These may be summarised as:-

  - Real-Time
    - A Real-Time program must
      - produce the correct results; and
      - produce those results within the allotted time (i.e. be deterministic).
    - Real-Time languages must facilitate, to the programmer, adequate control of both the timing and the sequencing of operations. However, Laplante (1993) states that constructs which do not automatically relinquish control of the processor should be discouraged by the compiler. An example of such a construct is a loop which can only be exited following an external event (e.g. when waiting for a key-press). The language Real-Time Euclid (Stoyenko & Kligerman, 1986) actually mandates the inclusion of some Maximum Time-Out at which loops must end in order to preserve temporal determinism.
  - Concurrent programming
    - Concurrent programming has been developed primarily as
      - a means for using the underlying computer hardware more efficiently; and
      - a methodology for the decomposition of large programming systems.
    - In concurrent programming the goal is frequently to hide the precise details of sequencing from the programmer.

The issues of providing support for concurrency have provided the subject for much research over the last three decades (Arjomandi, O'Farrell & Kalas, 1994) and have typically presented two approaches. In the first approach, language extensions are produced with concurrency constructs emerging as part of the (new) language; and, in the second, the lower level details of concurrency are kept outside of the language and encapsulated in libraries. Arjomandi et al., in presenting an overview of concurrency

support, found that the former is the more desirable and less restrictive approach, due chiefly to the observation that "users of concurrent libraries must observe certain protocols with varying degrees of severity".

- **Interrupt types**

  Best described by their features, the following points have been noted from the Turbo C++ Programmers Guide(1990). An interrupt type denotes a special kind of subprogram which, being activated by an event asynchronous to normal program flow (e.g. a hardware derived interrupt), is characterised by the following:

  ◊ it is void of parameters;

  ◊ it implicitly saves and restores machine registers as prologue and epilogue respectively;

  ◊ it terminates with an end-of-interrupt instruction (c.f. a subprogram return);

  ◊ it is typically associated with a vector location in memory which contains the interrupt-service subprogram's start address;

  ◊ it provides re-entrant code so that multiple instances of each may occur harmlessly.

  Variables accessed by, but external to, Interrupt-Driven subprograms need special consideration due to their asynchronous operations. Some C compilers provide the "volatile" keyword to prevent any compiler optimisation on such variables. Ada uses "pragma shared" to accomplish similar variable protection (Duhaut, Bidaud & Fontaine, 1992, p 299).

- **Modularity**

  Parnas and Clements (1972) give a set of criteria which provides for the partitioning of software modules with clearly defined intermodule communication but which avoids unwanted intermodule interference. The correct application of modularity can enhance maintainability. Sommerville (1990, p198) points out: "Maximising cohesion in a component and minimising coupling between components is likely to lead to a maintainable design". Such a modularisation is acknowledged to be desirable from many perspectives, but chiefly from that of information hiding, which suppresses how an object or operation is implemented and "focuses our attention on the higher

abstraction" (Booch, 1987, p33). However, with respect to aspects of Real-Time performance, Laplante (1993, p76) warns of an increased overhead with subprogram calls and their associated stack usage, that accompanies excessive modularisation.

- **Low-level control**

  Embedded systems frequently revolve around hardware control, and as such must be able to access intimately the low level architecture of the machine. A language used for such systems must provide realistic constructs which will effectively facilitate data movement between the CPU and Programmed I/O and/or Memory-Mapped I/O (Laplante, 1993, p48).

- **Portability**

  This stems from the emphasis placed by Cotigny & Ple (1991) upon the necessity of using a "normalised language with standard libraries" for portability. It is important to recognise that once a program has been written and tested, the real-world continues to evolve. Needs change, and computing hardware continually improves. Aucsmith (1988) comments that due to enhancements and enlargements, embedded systems tend to be "long lived and have many iterations". Where a processor is upgraded, portability becomes a prime consideration. Emery and Nyberg(1989, p 245) define Software Portability as simply "the effort required to get a piece of software running on one host to run on another". Their findings conclude that the capabilities and facilities provided by Ada assist in developing systems that are easy to port.

This concludes the discussion of the list of desirable features. The presence / absence of these features in commonly available languages is now discussed.

Figure 3.1 emphasises the poor support for desirable Real-Time features. It presents a comparison of features found in those languages which may be considered candidates for Real-Time embedded-system programming. Not included in the comparison are "hybrid" languages such as Concurrent Pascal, Concurrent C/ C++ (although some of their features are reviewed later), and Real-Time ones such as Pearl and Euclid. This is chiefly due to their lack of widespread availability or standardisation (de-facto or otherwise) for low cost

development environments. Also missing is the feature of dynamic memory support which, in most instances, is translated by the compiler into O/S calls and this study concerns itself with non-O/S hosted programs. Even if this were not the case, Stoyenko and Kligerman (1986, p942), point out that allocation and de-allocation of dynamic structures makes it very hard to ensure the temporal and resource determinism necessary for real-time determinism.

Language	Call-by-value-Parameter	Call-by-reference-Parameter	Strong-typing	Interrupt-type	Exception-Handling	Enum-erated-type	Modularity	Re-entrant code	Low-Level Control	Con currency
Ada	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y#
BASIC	N	N	N	N	N	N	N	N	N	N
C	Y	Y*	N	Y**	N	Y	Y	Y	Y**	N
C++	Y	Y	N	Y**	Y**	Y	Y	Y	Y**	N
FORTRAN	N	Y	N	N	N	N	N	Y**	N	N
Modula-2	Y	Y	Y##	Y	N	Y	Y	Y	Y	Y
Pascal	Y	Y	Y##	N	N	Y	Y	Y	N	N

*Simulated via pointers; **Compiler / Version Dependent; #Non-deterministic; ##except variant records

Figure 3.1 Feature Comparison Table (substantially augmented from an original in Laplante(1993, p79)).

Figure 3.1 illustrates that Ada provides a more complete set of the desirable characteristics for Real-Time embedded development. Ada has now been sufficiently used for comparative studies to have found that its use for non-trivial programs (greater than 1000 lines of code) provides "a significant advantage over other high-order languages (such as FORTRAN & Pascal), even for the first project." (Bhansali et al., 1991, p26). Bhansali et al.'s work suggests that the advantage increases with increased program size or when software components are re-used, for which Ada is well suited.

The facilities provided by Ada frequently come at the expense of run-time efficiency. Aucsmith (1988) summarises this as "The (Ada) language provides many features which are not present in any other single language. The problem with using Ada for embedded systems is more of a problem of efficient implementation of its features." Aucsmith is not alone in acknowledging Ada's limitations of implementation. Several have felt inclined to address Ada's inefficiency of executable code by exploring, or describing, run-time system

alternatives or enhancements (Dobler,1992; Kamrad, 1992; Sims,1991; Powers & Roark, 1990; Aucsmith,1988; Colton,1988; Baker,1988). Not all, by adopting a "strictly Ada" approach, met with immediate success in terms of speed (Baker, 1988), which gives an indication of the difficulty of altering such a highly integrated system. Struble & Wagner (1989) offer a quantitative evaluation of Ada's tasking model, particularly with respect to interrupt handling. Of note in their findings is that Ada compiler vendors frequently offer two interrupt handling schemes: one to comply with the Ada validation suite; the other non-standard but faster, to cope with Real-Time response! Interestingly, the model used for providing pre-emptive or serial multi-task control in Ada is implementation dependent. As Booch (1987, p281) notes "Ada semantics do not require that task scheduling applies a time slicing algorithm".

This study addresses the run-time efficiency problem by the approach of using C as an implementation vehicle. It addresses the lack of determinism and interrupt response by adopting a slight departure from the Ada tasking model.

3.2 Specific Studies Similar to the Current Study

Snow (1992, p9) acknowledges that a system allowing concurrency will almost always require more processes than the number of physical processors provided by the hardware. The abstraction of concurrency in these circumstances, which are directly applicable to this project, necessitates the use of a *pseudo-concurrency* where a processor's resources are shared in some way. This sharing is achieved via multiplexing the processor with respect to time, either on a pre-emptive or a co-operative basis. Snow (1992, p11) reports that even in the early 1960s the idea of multiple threads of control was being pursued, together with attendant problems such as memory sharing. By 1965, the idea of explicitly declaring areas of parallelism, using programming constructs such as *parbegin* and *parend* (Dijkstra, 1965), had been proposed. Dijkstra had formalised at a high level many of the problems of synchronising, data sharing and concurrency in his classic article "Co-operating Sequential Processes". Principally, he identified the critical section problem, when a process is accessing shared data, with the use of semaphores proposed as a solution. His secondary identification was the problem of deadlock, with a deadlock detection algorithm proposed. Dijkstra's semaphores gave way to monitors which encapsulated both the data and the necessary operators to permit serial re-usability. These were eventually refined by Hoare (1974).

The issues of data-sharing, or communication, between concurrently executing processes have produced three main models (Gehani & Roome, 1993, p 154):

a) the shared memory model;
b) the asynchronous (non-blocking) message passing model; and
c) the synchronous (blocking) message passing model.

A fourth sub-model has recently emerged, as exemplified in Linda (Appleby, 1991, p214), where a shared data space between, but owned by none of, the processes allows them to create, remove and edit data.

Since Dijkstra's first efforts, there has been much work in the problem areas of concurrency. The problem of providing a realistic, yet deterministic concurrency model is perhaps the single biggest problem facing the designer of a language for Real-Time embedded-system use. Instances of concurrency models which have provided lessons for this project will be reviewed now.

- Concurrent Pascal;
- Concurrent C;
- Oberon;
- Microsoft Windows;
- Smalltalk V;
- co-routines.

Concurrent Pascal is an instance of using an existing, already successful, language of the time as a vehicle for the description of concurrency using a shared memory model. Two major additions were made to Standard Pascal in order to achieve the concurrency required for "structured programming of computer operating systems" (Brinch-Hansen, 1975, p264):

- the **Process** : consisting of a *private data* structure and a *sequential program* to operate on that data, which may not be accessed by another process. A process functions independently of other running processes.
- the **Monitor** : consisting of a *shared data* structure, which may be initialised, and the *operations* that processes can use to access it. Processes call these operations which then exclusively access the shared data; thus, the operations have a synchronising effect. Monitors may be used to pass data between processes, or to system components such as disk drives, and may call other monitors. Deadlock is prevented by barring monitors from making recursive calls. Communications using monitors appear to be non-deterministic.

Concurrent Pascal compilers produce code (called C-code) which relies upon an underlying virtual machine (called a C-machine) in order to execute. The virtual machine provides a "multiprogramming kernel and interpreter" (Mortensen, 1984, p 155) and it is responsible for interpreting, at run-time, the C-code to executable machine code. All

machine specific functions and low level features such as Register-access, Addressing and Interrupts are excluded from the Concurrent Pascal language, being handled by the virtual machine.

Concurrent C (Gehani & Roome, 1993) is an instance of providing extensions to an existing language, C, to handle the issues of concurrency. Although Concurrent C uses predominantly message passing models, as with Concurrent Pascal two major additions were made to the base language:

- the **Process with parameters**: In Concurrent C the process consists of two parts: a declaration of the process name and its parameter types, and a body which additionally contains statements to execute. More than one instance of the process may be created, each one having its own set of parameters in order to pass information to the process when it is created. Each process instance may be assigned a priority at its time of creation and, if there are multiple processors available a process may be assigned to a specific processor. In uni-processor implementations, a library provides for pre-emptive multitasking.
- the **Transaction**: A caller process may interact or communicate with a receiver process via transactions, in a similar relationship to that between a client and a server. Two types of transactions are possible: synchronous and asynchronous. In the synchronous case, the caller sends data to the receiver and waits for the receiver to accept the transaction. In due course this happens, whereupon the receiver does whatever processing is necessary and returns some value to the caller, who now resumes execution. This allows for two way message passing, is also known as the *extended rendezvous* model (Gehani & Roome, 1993, p154), and is similar to the Ada tasking model. This model is non-deterministic. The asynchronous transaction allows the caller to send information to the receiver and simply continue processing. It waits neither for the receiver to accept the information nor for any return information, and it is suitable for uni-directional information exchange, and for connection to hardware interrupts. However, it has the considerable advantage of decoupling the caller from the receiver, but implies the use of some underlying pipe-lining of the messages. No attempt is made to avoid deadlock, this being left to the programmer.

Concurrent C is effectively a superset of the C language, which provides for the uninhibited use of shared memory at the simplest level (i.e. accessing objects via common pointers or global objects). It permits hardware interrupts to be attached to subprograms dynamically under programmer control, via a provided library function call to effect the association (Gehani & Roome, 1986, p821). Its compilers produce C code, i.e. it uses C as an intermediate target.

Oberon is the single-minded progress in computer languages pursued by Niklaus Wirth, starting with Pascal, evolving through Modula-2, and culminating in the integrated language and operating system of Oberon. This "is a single-user, single process, multi-tasking system designed for a workstation."(Wirth, 1989, p10). The Oberon system presents a multi-tasking model which is described (Wirth & Gutknecht, 1992, p 13) as allowing "the user to pursue several different tasks concurrently" but which does not depend upon pre-emption and, in fact, Wirth & Gutknecht (1992, p13) "classify Oberon as a single-process system". The Oberon system revolves around a central loop, which resides in module Oberon, this being the heart of the O/S. That central loop involves the processor in continually polling event sources whilst not involved in the interpretation of a user entered command. Synchronisation problems (described in Sebesta, 1989, p356) existed with the programmer controlled, implementation dependent and hence non-portable, multi-tasking model of Modula-2. In recognition of this, Wirth removed all multi-tasking access from the Oberon language to its co-existent operating system component.

Tasks within Oberon are styled according to perceived priority. *Interactive tasks* are fairly localised, being bound to local regions on the display screen and generally have a high priority and quick execution, whilst *background tasks* are described as being global in nature and are polled with low priority (Wirth & Gutknecht, 1992, p 26). Once tasks are created, they exist in a state of suspension until they are activated by the task scheduler passing a message to them. The synchronous mechanism for message passing here is analogous to passing parameters to a subprogram call. When the task has finished processing for that message it returns control to the task scheduler. The scheduler then proceeds to the next task which has a message to reactivate it, transferring control as an

ordinary subprogram call and awaiting the normal subprogram return. Asynchronous events are dealt with by interrupt service routines within special drivers that are accessories to the O/S. These routines buffer the result for later collection by the central loop.

In summary, Wirth presents a tasking model peculiarly suited to the single user, who may use the system to flip between tasks at will. To achieve this model, pre-emption was found to be unnecessary. This served to eliminate any context switching and removed the need for data protection such as locking of common resources. Deadlocks, therefore, are never a threat.

Microsoft Windows currently runs on top of the single tasking and non-reentrant MS-DOS, and accordingly employs a non-pre-emptive tasking model (Petzold, 1992, p8). Windows programs are not interrupted by the operating system: instead, each program voluntarily interrupts its own operation to let any other programs run. Coupled with this is a "message delivery system" (Norton & Yao, 1992, p15), which takes the form of a queue. Interrupt driven subprograms within device drivers buffer asynchronous events, e.g. keyboard entry, into the hardware event queue for subsequent collection and processing by the active Windows program. When the Windows program has finished processing the message it will return control to the Windows scheduler and request any new message. In the event of lengthy processing, the programmer can temporarily relinquish control, using pre-defined system calls, to the scheduler at suitably placed synchronisation points. Like Oberon, Windows also has foreground (e.g. character entry) and background processes (e.g. printing).

The Windows model is similar in concept to Oberon's, except that the programs themselves initiate a temporary transfer of control to a central scheduling system at programmer controlled synchronisation points, instead of initiation by the user. A program in this instance can be seen as a process running under the Windows scheduling system, which also maintains queues for interprocess communication.

Smalltalk V, being a pure object-oriented implementation, involves "communicating objects which send messages to each other so as to perform useful work"(Digitalk, 1992, p251). Smalltalk V allows for pseudo-concurrency by maintaining multiple stacks of incomplete message sends. Each stack is represented by a separate object of class Process. At any time a single process is executing. Processes, at creation time, are prioritised which means that higher priority processes are scheduled before those of lower priority. Processes either terminate or, if incomplete, relinquish control with the wait construct to the scheduler, which will allocate the processor to the next task based upon priority. In the event of competing equal priority tasks, length of time spent "ready-waiting" will determine allocation. The Semaphore is used to synchronise multiple processes, where a process will send to the semaphore the message *wait*, meaning to attend an event, or *signal*, meaning to indicate an event has occurred. Smalltalk V also supports Interrupt Processes.

The co-routine, exemplified in the Simula-67 and Modula-2 languages, is described by Sebesta (1989, p347) as "a subprogram that has multiple entry points, which are controlled by the co-routine itself, and the means to maintain its status between activations". Co-routines provide a relatively simple means to achieve co-operative multi-tasking since they can relinquish control to the processor and, when granted the processor's resources again, resume execution just after the last active statement. Knowing the points of resumption and relinquishment of control, and not being dependent upon external timing, co-routines can be seen to be deterministic. Rather than acting in a master-slave relationship, co-routines co-exist as peers (Appleby, 1991), passing control co-operatively. Akerbaek (1993) suggested extensions to C++ so as to implement co-routines. In Akerbaek's scheme each co-routine has its own stack. This becomes the current machine stack when that co-routine is activated by being called as a normal function from the main program. When the co-routine wishes to relinquish control it saves its current stack status before returning to the main program as if it were a normal function. Similar work, at the University of Brighton, by English is described in Volkman (1993, November). English, attracted by the lack of elaborate interprocess synchronisation, presented co-routine classes for Borland C++ under MS-DOS.

In summary, the multi-tasking models explored can be perceived as being according to two models: pre-emptive and co-operative. Data sharing is less complex with the co-operative model, needing less protection against critical sections and deadlock. The Smalltalk model showed that prioritisation can be accommodated in a co-operative tasking model, and the elusive temporal determinism can be achieved with co-routines. Data can be shared between processes by using areas of shared data or the use of message queues, the latter engaging processes synchronously (wait for a reply) or asynchronously (deliver the message and get back to work).

The issue of writing a compiler to produce code to run on a bare machine, i.e. one without an O/S, was found by Colton (1988) to present some interesting problems. His solution involved incrementally crafting a mini-operating system, predominantly in Assembler, to perform much of the run-time support needed for the compiler. The compiler's output code, then, is situated one layer removed from the machine. This, however, raises the prospect of the run-time system having to be re-written for each target processor. Bentley (1986) suggests that if an intermediate language can be found, this "circumvents much complexity". Portability across different target machines becomes a matter of using an available "back-end translator", provided by compilers of the chosen intermediate language. Diagrammatically this may be represented as follows:

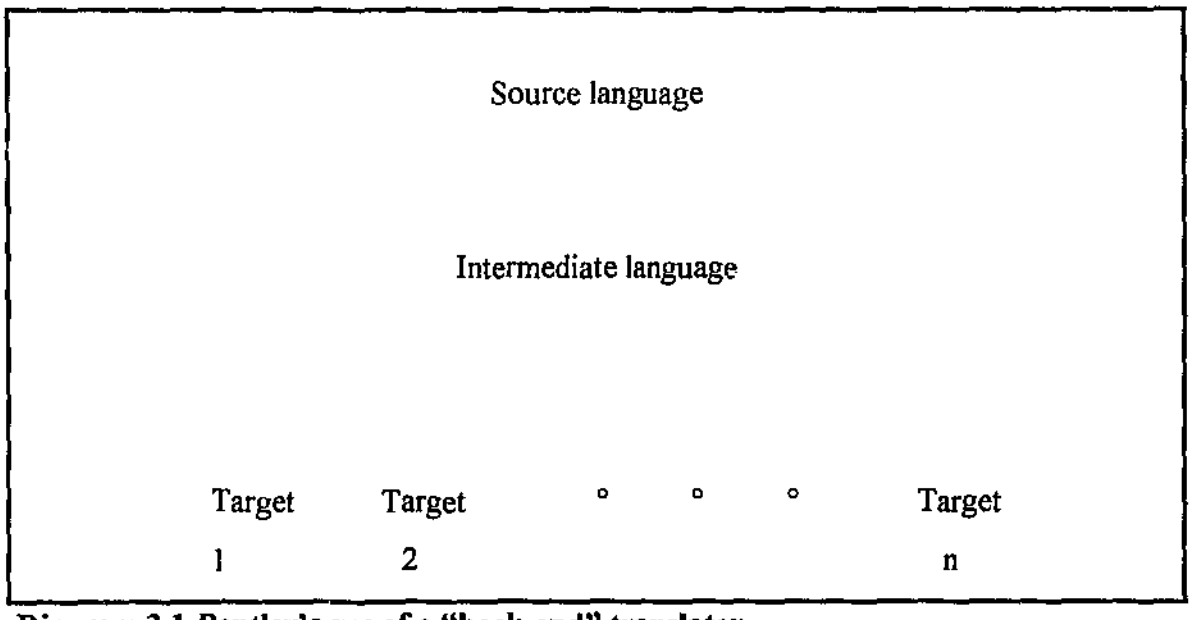

Diagram 3.1 Bentley's use of a "back-end" translator

The concept of using C as such an intermediate target is no longer regarded as a novel approach, and is often adopted for very practical reasons. As Eckel (1993, p40) describes, Bjarne Stroustrup recognised when creating C++ that the key to acceptability is availability. To hasten this availability, he used a C-code generator, *cfront*, that may be quickly ported to any platform which boasts a C compiler. The giant A.T.&T.(trademark) put their resources behind the creation of *cfront*, which translates C++ code into C, a technique which is still commonly used on Unix C++ implementations(Cullens 1994).

Eiffel, an object oriented language which performs strict static type checking, uses C as its target. Meyer (1988, p487) explains, "As a language, Eiffel is in no way an extension of C, but the use of a widely available assembly language such as C as intermediate code has obvious portability advantages.". Similarly, the reactive Real-Time language for workstations, Esterel, (Boussinot & De Simone, 1991) can be translated into one of several target languages, (Ada, C or Lisp) which, for portability reasons, is C by default.

The practice is not limited to being a tool for the code generation for programming languages. Odette (1991) describes CLIPS, a NASA developed language tool-kit for use in Artificial Intelligence, which successfully uses C as its implementation language, and also describes the use of C to simulate a Prolog machine.

There are other advantages. The use of C, in this project, allowed the study to concentrate on the higher level issues of the presentation and workings of the translation process. It facilitates an implementation without major concern for the separate problems of target code generation and linkage.

3.3 Other Literature of Significance to this Study

Usually, where executable code is generated for an O/S hosted target, that code will be relocatable by the O/S using well-established location techniques. Recognising this makes it possible to envisage the possibility of locating such code so as to run on an absolute address target. The techniques of re-locating operating system targeted files are documented by Allison (1994) and Pillay (1990), the latter of whom presents a tutorial on the subject, using the 80x86 processor as an example target. Diagrammatically, the mechanism can be represented as follows:-

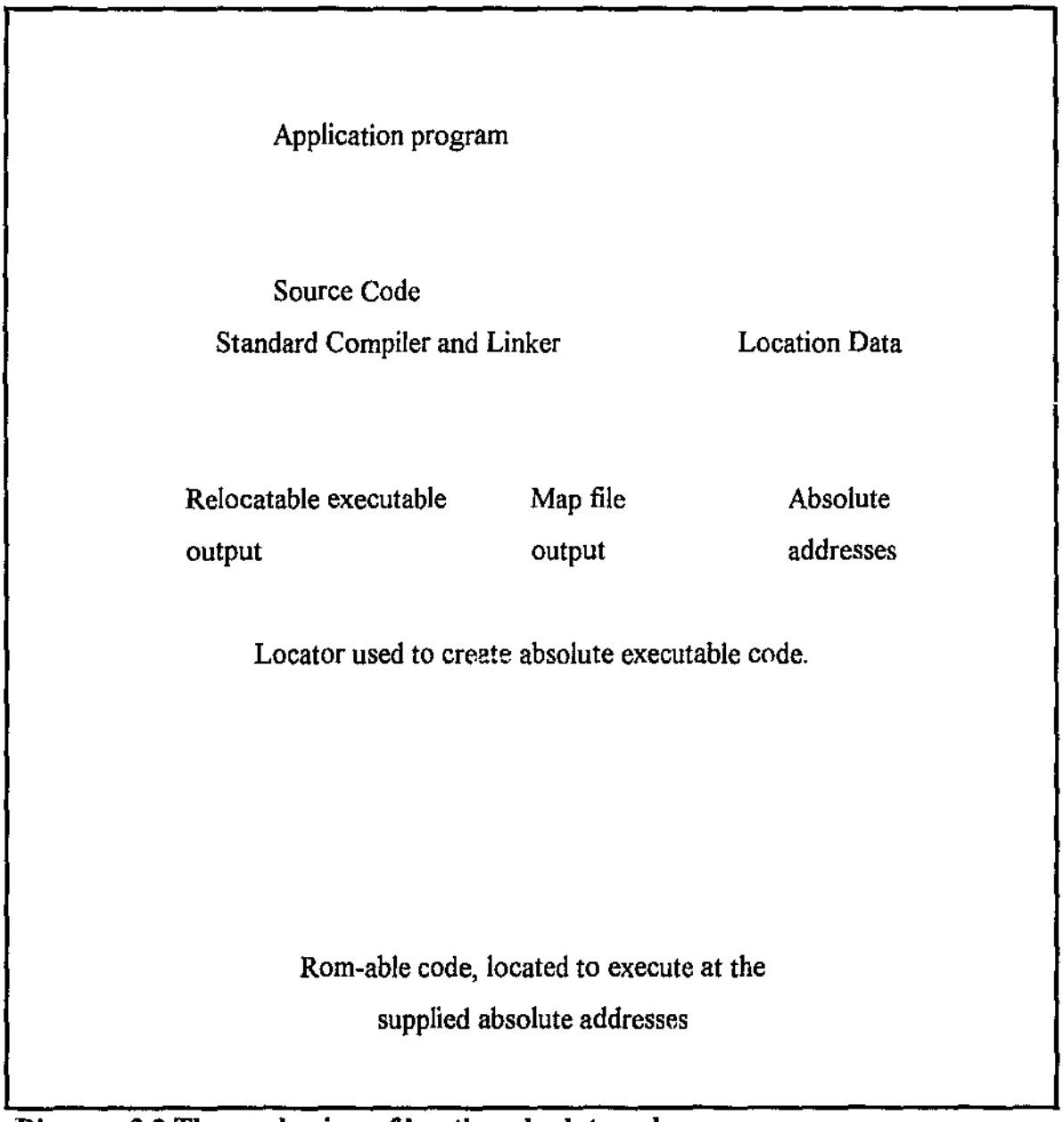

Diagram 3.2 The mechanism of locating absolute code

Both Pillay's and Allison's works describe how the MS-DOS standard .EXE file header format is combined with .MAP file information provided by MS-DOS linkers to achieve location at the specified Absolute addresses. Pillay's article gives a comprehensive analysis of that .EXE header format. Allison's article additionally describes the necessity of providing appropriate start-up code so that the Application main program is launched correctly. This, at least, consists of:

- setting up any segment access registers for program code, data and stack segments;
- establishing a program stack area;
- transferring any initialised variable data from ROM to RAM;
- setting un-initialised variables to zero;
- setting-up any error (e.g. divide by zero, null access) vectors; and
- calling the application main program.

It should be noted that, since commencement of this study, several commercial programs for the location of executable code, produced by proprietary C compilers, have become available.

Welsh (1993) describes a method of using a language based on an O/S hosted language to achieve development of critical but platform independent components of a project, citing his experience with an embedded data compression project. This is so that O/S development environments can be used to ease the initial coding phase. Welsh's compression code was initially written in a subset of VAX hosted FORTRAN which was sympathetic to limitations of the eventual target language, Intel's proprietary PL/M. When this phase was completed, the source was successfully transliterated to PL/M so that it could be compiled for the target 80386 micro-processor.

The code produced by proprietary compilers, as supplied by Microsoft or Borland, quite reasonably expects to find operating system (O/S) support. Phillips et al. (1991) advocate caution when using procedures from the C standard library (through which C performs much of its Input/Output), and suggest three options for determining and supplying the program's needs for O/S support:-

- buy the vendor's library source, then modify it for your own purposes; or
- implement your own run time system; or
- inspect your disassembled code for calls to known O/S function entry points.

This study is based on the concept that a reasonable and far safer alternative is to produce code that *never* makes O/S function calls. To achieve this, the Ada-like language, provided for scripting of the original source, is automatically transliterated into a subset of C. This subset of C, known here as "Safe C", has been tested for O/S calls and will behave in a manner sympathetic to the limited start-up procedures provided for the application program to execute successfully.

3.4 Summary

The evolution of programming languages, and indeed languages in general, is a steady process, with each advance being in response to fresh and re-emergent needs, or the disappearance of previous needs. What one person sees as an advantage, another may clearly view as an unnecessary handicap. Clearly it is necessary for any language designer, or one proposing change to an existing language, to define needs and hence responses in an organised and structured manner and base these upon the firm foundation of fact. Hoare's dictum that the language designer's task is one of "consolidation, not innovation" (Hoare, 1973) is as valid now as it was when first stated.

Using facts and documented experience, the literature has been used to provide a basis for a discussion about identifiable desirable features that cater for the special needs of Real-Time embedded systems. Significant models of pseudo-concurrency have been discussed, with an emergent idea that co-operating multi-tasking processes can achieve temporal determinism using less elaborate synchronisation than that needed for pre-emptive tasking models. Also reviewed were the reasons for, and virtues of, using a portable intermediate language, or a safe subset of one, during the translation process from source to executable code for the target.

4 Research Design

4.1 General Method

In order to produce an effective pilot implementation, it is necessary to realise three distinct components:

- the determination of "Safe C": that is, the subset of the intermediate language C which, when compiled, makes no O/S calls and is therefore safe for use on O/S-less embedded systems;
- a translator which accepts an Ada-like language as its source code for translation into Safe C; and
- the provision of start-up code necessary to organise the target microprocessor so as to support the compiled program.

As noted in chapter three, commercial software is available to convert relocatable code to absolute programs which may be executed from ROM. In the interests of expedience, such a product is used, for this purpose, in this project.

Determination of Safe C

As Phillips et al. (1991) suggest, one way of determining whether or not the functions in a program exhibit any O/S dependency is to inspect the program's disassembled code for O/S related calls. In the case of this project, the "Safe C" subset of the C language is determinable by compiling and linking a C program containing a selected usage of each function under test. The resultant executable may then be disassembled and inspected for software interrupts, by which all communications to the O/S are made. Any such O/S calls made by a function may be used as a criterion for eliminating that function from inclusion in "Safe C".

The C language, by itself, is not a large language and much of its power comes from the function libraries and pre-processor which accompany a C compiler. The task of testing functions within those libraries is made easy because the functions tend to be grouped within the libraries. Furthermore, within these groups, functions exhibit data or functional similarities; thus, the task of testing is straightforward.

The design of the translator, from the Ada-like language to Safe C, relies heavily upon the application of strong typing within the source language. In a strongly typed language there may be found three distinct areas/phases within each submitted program:

- A Global Specification area which contains symbols of:
  - ◊ a declaration of each subprogram, specifying a name, a type and any formal parameter list to identify the subprogram for use throughout the program;
  - ◊ all globally visible constants, specifying their names, types and actual values;
  - ◊ all global variables for sharing data between the subprograms, specifying their names and types.

  In the case that all global symbols are declared in this manner at the beginning of the program, it is possible to complete the translation exercise in a single pass. This is seen by Fischer and Leblanc (1991, p17) as having a "positive contribution to compiler efficiency".

- For each subprogram, there exists a Local Specification area which consists of:
  - ◊ a re-iteration of the subprogram declaration, with the name, the type and the formal parameter list as specified within the global specification area;
  - ◊ all local constants, specifying their names, types and actual values;
  - ◊ all local variables, specifying their names and types.

- Each subprogram, of necessity, also contains an area where the subprogram's work is achieved. Typically, this is delimited by an identifiable beginning and ending, and contains:
  - ◊ assignments to variables, where a left-hand side variable is used to receive an evaluation of identically typed variables, functions or constants resident on the right-hand side. Between the left hand side and right hand side must exist an assignment operator (e.g. :=);
  - ◊ subprogram calls which return no value, but which may operate upon currently visible variables and constants.

The translator initially finds itself in the Global Specification area, thereafter it must exist exclusively in one of the other two areas at any point in time, and can therefore respond to statements encountered within the submitted source code according to the particular rules of that phase of the translation.

In order to simplify the design of the pilot implementation there is no provision for the nesting of subprograms, thus visibility is restricted to the following two levels:

- global variables, constants and subprogram specifications are visible and accessible from within all subprograms and endure for the life of the program; and
- local variables and constants may be referenced solely from within the currently executing subprogram and endure for its life only. Note that local variables and constants may obscure global entities which share the same name.

Provision of start-up code

The work of Pillay (1990) provided an ideal basis for the provision of suitable start-up code, although further experimentation became necessary in order to accommodate executable code generated by the Microsoft C compiler. Preparing such start-up code at its simplest level entails:

- turning off all maskable interrupts;
- setting up the Code, Data and Stack registers so they may access available RAM;
- declaring the "main" function, present in all C programs, as an external callable label;
- for Microsoft C specifically, setting the global label "_acrtused" to a non-zero value to prevent the Microsoft linker from including the default O/S dependant start-up code;
- copying initialised data from ROM into RAM;
- calling the compiled C program via its "main" function.

4.2 Specific Procedures

The derivation of the "Safe C" subset, for the Microsoft C compiler version 6, relies upon the detection of calls made to O/S services. These services will not be present in an O/S-less system, and calls to them will result in unpredictable and probably fatal results. The Microsoft C compiler is capable of generating programs to run under the operating systems of MS-DOS, O/S-2, and Microsoft Windows, according to configuration details supplied to the compiler. With the compiler configured for its default production of MS-DOS based programs, we may concentrate solely upon MS-DOS O/S service calls.

MS-DOS O/S services are invoked by "using the **int** instruction and specifying Interrupt 21h" (Microsoft Corporation, 1991, p11), rendering their detection relatively straightforward. The method is as follows:

- assemble start-up code so as to prevent linkage with the MS-DOS library;
- submit source code to the C compiler, compiling to object format and linking with the assembled start-up code to produce a .EXE file;
- disassemble the .EXE file to produce an assembly listing;
- load this assembly listing into a text-editor, and search for the string '**int** 21'.

In the absence of '**int** 21' statements, the functions (properly) used within the source code may be deemed safe for use in an O/S-less environment.

Tools used in the above procedure are:

- the Microsoft C compiler;
- the Microsoft Assembler or Borland's Turbo Assembler;
- a Dis-Assembler (e.g. the ShareWare program "Sourcer"); and
- a text editor capable of performing text based searches (e.g. Microsoft's EDIT).

The construction of the translator from the Ada-like language source code to "Safe C" focuses upon the desirable features identified in chapter three. The decision to craft a translator from scratch, rather than combining the use of existing tools like Lex and Yacc is based upon the following observations:

- Yacc, when used for any non-trivial exercise, produces an output which must be submitted to a C compiler before it may be executed (Mark Williams Company, 1992,

p1185). The C source code produced is effectively unalterable by its lack of readability. This represents a limitation of control over the final product which was felt by the author to be unacceptable for this project.

- Yacc has problems dealing with any ambiguities which may be present in a language, and which may not be resolved by the use of precedence rules executed (Mark Williams Company, 1992, p1185). In keeping the syntax of the Ada-like language closely aligned to that of Ada itself, such ambiguities exist principally in the representation by names of machine specific features: for example,
  - ◊ the hardware mapping of tasks used as interrupt service routines, as distinct from multi-tasking tasks; and
  - ◊ the location of I/O mapped and Memory mapped hardware register variables as distinct from normal program variables.

  Other ambiguities exist in the automatic generation of co-routine management code.
- The effective use of Yacc appeared to the author to require a substantial learning investment which, it was felt, could not be guaranteed to produce the desired translator in the time available.

  Ada, the language chosen for the crafting of the translator, permits the rapid construction and easy instantiation, in various guises, of generic components such as trees, lists, and binary searches. These facilities assist greatly in the construction of major components of the translator. Illustrations are given in Appendix D of the useage of Ada's generics in this implementation for the effective construction of the symbol trees, which are necessary for rapid symbol location.

The translator addresses the list of desirable features for Real-Time Languages, identified in chapter three in the manner described below.

- **Strong Typing**

  Recall, from chapter three, that Sebesta (1989) suggests the idea of strong typing is where each name in a program is associated with a single type, which is known at compile time. Using this concept, the pilot implementation checks all declarations and subsequent occurrences of named constants, variables, subprograms and parameters with respect to consistency of type. Constants need to be assigned values which are

consistent with their declared type. References to symbols used in left and right sides of assignments need also to be checked for such consistency.

The relationship between declaration of a name and its subsequent reference for use in a subprogram is that of one to many. Due to this relationship it becomes apparent that the translator will spend some of its time inserting a declared name into some holding structure, but much time will involve searching for that name. This mandates that the translator employ a rapid method of searching for the declared names, for Booch (1987a, p501) quoting Knuth states "the substitution of a good search for a bad one often leads to a substantial increase in speed". To this end binary trees are employed for named subprograms, variables and constants as they exhibit an order of efficiency of insertion and searching which tends towards $n\log_2 n$ (where there are n items in the tree). Parameters, which tend to be small in number, are stored in lists as Booch (1987a, p 464) quoting Aho, Hopcroft and Ullman states "the more sophisticated algorithms are generally a waste of effort for n less than one hundred". A generic binary tree may be easily instantiated to facilitate each case of rapid storage of, and access, to essential details of:

- global constants, whose named references are replaced throughout the translation by their values, after successful type-checking with their destination;
- global variables, whose use is permitted after successful type-checking with any source or destination during assignments or operations;
- subprograms, whose use is permitted after type-checking of any return value and constistency of connection between declared formal parameters and actual parameters supplied in subprogram statements; and
- local constants, which are treated in similar manner to global constants in terms of replacing names with values.

This represents a complete, and effective, type-checking mechanism for all necessary components of the Ada-like language. However, to cater for occasions when it is expedient to change the type of a named entity, explicit type casting is permitted. An example of this is when a variable of type CHARACTER needs to be output to a serial port transmission register which is of type BYTE.

- **Flexible parameter passing mechanisms**

  Parameter passing methods of IN, IN OUT and OUT parameters are facilitated for all provided data types within the Ada-like language, there being no upper limit to the quantity of parameters passed to a subprogram in keeping with the consistency of its declared formal parameters.

  Simple data types provided in the Ada-like language are as follows:

*	INTEGER	-- a 16 bit signed value,	e.g. -1968;
*	WORD	-- a 16 bit unsigned value	e.g. 43764;
*	CHARACTER	-- a 7 bit value	e.g. 'a';
*	BYTE	-- an 8 bit unsigned value	e.g. 255
*	ADDRESS	-- a 20 bit unsigned value	e.g. 655536
*	FLOAT	-- a floating point number	e.g. -197.64

  The decision to service only simple data types is taken to keep the pilot implementation manageable in size, yet still demonstrate effective addressing of the desirable features identified in chapter three. Complex and user defined data types are essentially mathematical extensions of the simple types to be found in a language, and operations on these have been deferred to a later study.

- **Recursion**

  The decision of whether to prohibit or permit recursion within a language is a classic illustration of the type of calculated compromise which must be borne by a computer language designer. Inevitably the language must be aimed at a certain type of programmer, and Sebesta (1989) suggests that the designer must consider a balance between freedom of expression within the language and protection of the programmer from their own mistakes. It is the author's experience that embedded systems engineers are often instrumental in designing both hardware and software for their system. They may be assumed to be aware of any limitations of memory space which would be affected by excessive recursive stack use. Recursion, involving self-referential subprograms, therefore is included in the Ada-like language, subject to the general strict type checking rules for subprogram calls.

- **Exception handling**

  From chapter three we have seen the immense value of potential protection offered to embedded programs by the facility of exception handling. However, the sheer size of intercepting and coping with all possible sources of unexpected or erroneous program termination renders it beyond the practical scope of this project. Consequently, the subject of exception handling is deferred to a later study.

- **Concurrency**

  An emergent idea was developed in chapter three of using co-routines to achieve temporal determinism within a pseudo-multitasking program. Ada provides the keyword "task" to denote a subprogram which is involved in some kind of background processing.

  Tasks which are used as an expression of concurrency will not be associated with a specific memory location, as is the case with interrupt service routines. In order to function correctly, co-routines must have two essential components:

  ◊ synchronisation points that, when encountered, smoothly relinquish control to the next waiting process; and

  ◊ management code to resume the execution successfully from the point where it was last relinquished from within the co-routine.

  Once again noting that embedded systems engineers are, in general, intimately familiar with the requirements and limitations of their system, the placing of the synchronisation points is reasonably placed under programmer control by provision of the keyword "**synch**" (see Appendix B). However, during the translation process when the translator will be aware of the location of the synchronisation points, it is then a relatively simple matter for the translator to generate code to manage invisibly the relinquishment and subsequent resumption of control.

  For co-routines to perform any worthwhile work, they must be able to communicate with each other. Recalling that co-operating processes such as co-routines are never able to compete simultaneously for data, the "shared data" model is used. In the pilot

implementation instances of the shared data objects are, in fact, globally declared variables. Co-routine tasks in the Ada-like language communicate solely through these objects and, therefore, are parameter-less subprograms.

- **Elimination of constructs with indeterminate execution times**

  I/O ports may change their state asynchronously with respect to the main flow of the program and it is desirable that these should not be relied upon to terminate a loop construct because in the worst case, the loop may never terminate. The translator prohibits the inclusion of variables mapped to hardware locations with representation clauses in the resolution of loop control. Whilst this does not trap all possibilities of a program becoming "hung" while waiting for a port to change state, it goes some way to focus the programmer on this possibility, by trapping obvious places where this may occur.

- **Interrupt types**

  Hardware interrupts are an essential part of most embedded systems, for they permit the rapid processing of asynchronous events as close to the time of their occurrence as possible. Few languages permit the automatic association between a hardware interrupt vector and the code responsible for servicing the interrupt. Ada permits this by use of representation clauses, where the language lets us refer to the interrupt vector by the name of the subprogram with which we wish it to be associated. In similar manner, the Ada like language permits such association, and the translator necessarily supplies all code needed to effect the association at run-time.

  It is necessary to protect any globally declared variable from the asynchronous effects of being accessed during interrupt servicing. This is achieved by the translator declaring all globally declared variables to be "volatile", indicating such protection to the C compiler.

- **Modularity**

  In this pilot implementation modularity is provided at subprogram level only. This permits the programmer the opportunity to maximise cohesion at a functional level, and to minimise coupling by the choice of parameters used to communicate with a subprogram. Package level modularity is deferred for a later study.

- **Low-level control**

  Intimate machine access to allow the manipulation of I/O ports is a vital requirement of embedded systems control. The Ada-like language permits such control in two aspects:

  ◊ permitting named port variables to be associated with hardware locations by means of representation clauses. Such ports come in two "flavours":

  * I/O mapped (denoted by a representation clause containing the *map_at* keyword); and
  * memory mapped (denoted by a representation clause containing the *use_at* keyword).

  From a programmer's point of view there is little difference between these, and the Ada-like language permits identical operations to be performed on both types. The translator must, however supply code for the machine to access each type in prescribed fashion. This differs from most current language implementations (e.g. Turbo Pascal, Borland C, Microsoft C, Meridian Ada), which force the programmer to use procedures to write to I/O ports and functions to read their value. This has a potential portability benefit, since porting I/O mapped variables to a memory mapped machine will only necessitate a change of the representation clause from MAP_AT to USE_AT.

  ◊ the facility to perform bitwise operations such as bitwise AND-ing and OR-ing of variables and ports. Again, from the programmer's point of view there is little difference between the expression of Boolean logical operations and bitwise logical operations and it is logical that identical keywords are used. The translator determines the particular operation from the type of variables being operated upon, and it supplies appropriate code to the C compiler. In this implementation bit-

shifting operations (e.g. Shift Left and Shift Right) have not been supplied, chiefly for reasons of time; and

◊ the provision of statements to turn on and turn off machine interrupts.

- **Portability**

  Portability is achieved by using C as the intermediate language in the manner of Bentley's "back-end" translator (see Diagram 3.1). However, although an ANSI standard exists for the C language, all C compilers are not the same! Fortunately, the C pre-processor permits the use of macros to align C source code statements in such a manner as to be acceptable to various compilers. This pilot project is directed toward the Intel 8086 instruction set computers. Accordingly, macros are supplied to satisfy compilation by both Microsoft and Borland C compilers. An illustration of such code is given in Appendix E.

4.3 Verification of the translation software

Sommerville (1990), in his discussion on formal specification and his discussion on formal verification, suggests that an implementation be verified for correctness against a known precise specification. In the case of a computer language, a precise and formal specification may be expressed using the Extended Backus-Naur Format (EBNF). Such a specification is included for the Ada-like language in Appendix B.

This pilot implementation must be able to translate correctly (in readiness for compilation by the C compiler and subsequent execution) a program written in accordance with the formal EBNF language specification. It will be a measure of portability that the code be found acceptable by the two major C compilers for the 8086 family of processors: namely, those of Microsoft and Borland. Material presented in Appendix E illustrates the methods by which compatibility between differing C compilers is achieved. Differing machine specific instructions, such as those for addressing I/O mapped ports preclude any attempt at a generalised compatibility for C compilers.

Finally, on a more general note, the translator makes no attempt at error-repairing. That is, when an error is encountered in the submitted source code, the translator will terminate, and, also inform the user of the error-type, the token at the point of failure, and the line number where the error may be found in the source code.

4.4 Summary

The major components of design for the pilot implementation of an Ada-like language have been described. In the course of chapter three, a list of desirable features for Real-Time languages was developed. In this chapter a response has been made to each element on that list. Finally the broad criteria by which the implementation may be judged in terms of success or failure have been introduced.

5 Findings

5.1 The Implementation of the Study

As stated in chapter four, the implementation required three major components to be produced:

- the determination of "Safe C". This involved the selected use of groups of functions. These were compiled, disassembled and inspected for the tell-tale "**int** 21h" instruction which, when present, revealed a dependency upon the O/S. Having established the procedure, this component became a straightforward process and its tabulated results are given in Appendix C of this document.

- the translator which puts into practice, to the extent described in chapter four, the desirable features for real-time languages espoused in chapter three. This pilot implementation has been accomplished using the Ada language, which lends itself well to the fabrication, and facile re-use, of substantial portions of code. Notwithstanding this, the project proved to be a substantial undertaking and required approximately 5500 lines of source code to complete the working model.

- the provision of start-up code to support the compiled program. This, essentially represented a refinement of Pillay's (1990) work, chiefly in the area of the specific details required by the chosen target microprocessor.

Having completed these three major components, it now became necessary to prove the effectiveness of the pilot implementation. This involved the selection and production of two further key elements:

- a suitable target system, adequately equipped to demonstrate a working test program expressed using the Ada-like language detailed in Appendix B; and
- such a working test program.

The target system

Embedded-systems engineers, when selecting hardware, must include such criteria as suitability and availability. The question of suitability looms large because, above all, the hardware must be capable of doing the required processing in a timely reliable manner. Availability must also be considered carefully, as considerable investment will occur before production models may be released, at which point the engineering investment may begin to be recouped. The hardware components of the production model must continue to be available in economically viable quantities until a satisfactory return on that investment has been achieved.

Based upon the aforementioned criteria, the choice for the target system centres upon the NEC V25 microprocessor which exemplifies a medium-performance low-cost processing device. Together with its accompanying hardware (designed and produced by Sturt Technology of Adelaide, South Australia) it is eminently suitable for demonstration of the features of this project. In terms of availability the target system is priced at a level which makes it attractive for incorporation in embedded systems requiring large or small production quantities. In short, it is representative of the kind of system upon which an embedded-systems engineer, such as the author, could be expected to rely in order to develop a product.

The V25 processor chip is a complete microcomputer subsystem whose core CPU is software compatible with the Intel 8086 range of processors: consequently, it can support code generated by PC based C compilers (e.g. Microsoft C). On chip, there exist two-hundred and fifty-six programmable byte-wide memory-mapped register ports to provide facilities which include:

- two full duplex asynchronous serial ports;
- twenty-four parallel I/O lines;
- two sixteen-bit timers;
- a twenty-bit time base counter;
- a programmable interrupt controller; and
- a two-channel Direct Memory Access (DMA) controller.

On power-up, these memory mapped register ports are relocatable within a wide range of the total address space. Off chip, the V25 addresses a potential one Mbyte memory and provides a sixty-four kbyte I/O mapped port address space. In this I/O mapped port address space is a Real-Time clock calendar chip/integrated circuit (I.C.). An on-board EPROM of up to sixty-four kbyte, and two thirty-two kbyte RAM I.C.s are provided. Further memory and I/O mapped devices may be accommodated externally (off-board) via control, address and data bus lines for which connectors are provided. However, it should be noted that in order to achieve any real functionality with the V25 on chip devices, substantial initialisation of both their registers and the CPU interrupt registers is necessary.

From the programmer's perspective, the I/O and memory maps appear as in Diagrams 5.1 and 5.2, respectively.

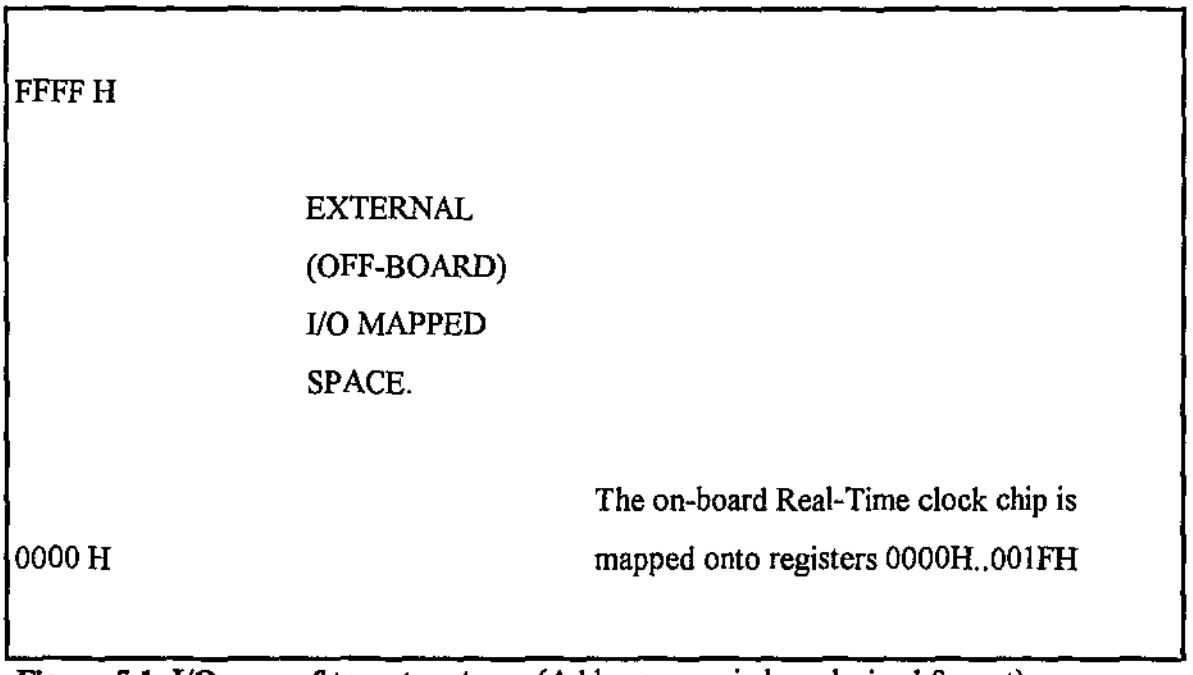

Figure 5.1. I/O map of target system. (Addresses are in hexadecimal format)

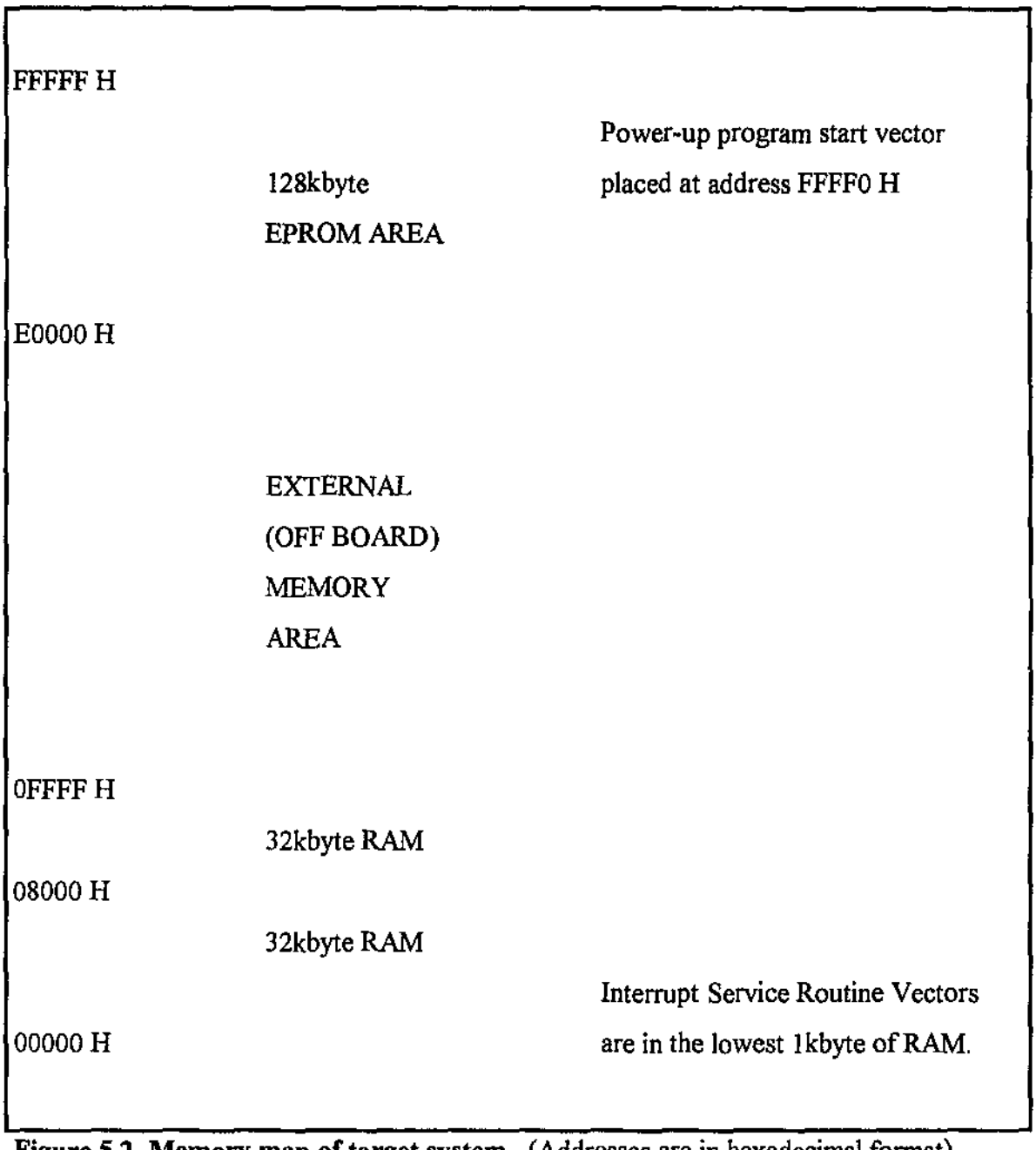

Figure 5.2. Memory map of target system. (Addresses are in hexadecimal format)

The test program

The test program was designed to demonstrate the language features developed in this project and, also, to represent the sort of program which an embedded system could reasonably be expected to achieve: e.g. that of protocol conversion and periodic message transmission. The target hardware includes two asynchronous serial ports and a Real-Time clock chip. Under the test program, one serial port is configured to support the RS232 interface at ninety-six-hundred bits per second, the other to support the RS422 interface at forty-eight-hundred bits per second. Additionally, the output of the Real-Time Clock "seconds" port is transmitted once per second.

One of the classical demonstrations of concurrency is the *producer - consumer relationship* as described in Deitel (1990, p90) and Gehani (1989, p125). The *producer* is engaged in generating data that a second process, the *consumer*, uses when available. Importantly, the producer is decoupled from the consumer such that the consumer cannot slow down or otherwise regulate the producer, which is free to generate data within reasonable limits as rapidly as it needs. Varying this relationship, the test program's role incorporates the following considerations:

- data may be forthcoming from three sources for collection by the consumer:
  - ◊ the RS232 serial port: whereupon an interrupt service routine (ISR) task buffers the data and raises a flag to indicate its presence;
  - ◊ the RS422 serial port: whereupon a second ISR task performs similar buffering and flag-raising; and
  - ◊ a producer task, which continuously polls the Real-Time Clock chip to detect the presence of fresh "seconds" upon which that data is buffered and flagged;

- the consumer will perform as follows:
  - ◊ when data is available from either of the serial ports, it will send that data to the other serial port; and
  - ◊ when data is available from the fresh "seconds" producer, it will send the latest "seconds" in character form to the RS232 serial port;

- both producer and consumer demonstrate the ability to communicate meaningfully with each other and with machine hardware; and

- asynchronous events are handled by interrupt service routines and buffered for later use by the consumer.

Complete listings of both the source test program in our Ada-like language, and the intermediate "Safe C" translation are given in Appendix F. Note that while the use of mixed case in the source listings is arguably unattractive, its use demonstrates the case-independence of the translator.

Annotated extracts from the test program are now examined, together with their translated "Safe C". The purpose of this examination is twofold: first to demonstrate the "behind the scenes" work done by the translator; and, second to demonstrate that C makes an excellent intermediate language.

Every program must have only one driver subprogram (Fig. 5.3a). From the programmer's perspective, this identifies the main entry point to the program.

```
driver TEST_PROG;
```

Figure 5.3a. Syntax of a driver subprogram specification.

This distinction cannot be carried through to the C compiler, so it is translated as a subprogram with no return value, which in C is of type *void* (see Fig. 5.3b). Note also that the first statement in the translated code enables the inclusion of the macros written to align the "Safe C" code to the chosen C compiler, thus increasing portability. This file, "standard.c" is presented, in its entirety, in Appendix E.

```
#include "standard.c"

void test_prog(void);
```

Figure 5.3b. Initial lines of the translated "Safe C" code.

Specifications for functions and procedures (exemplified in Fig. 5.4a), with and without formal parameter declarations are made as in the Ada language. These form the translator's foundation for flexible parameter passing and strong-typing and it is against these specifications that all type checking is done with respect to parameters passed and function return types.

```
function CHAR_LENGTH_CONVERT( THE_LENGTH   : integer  ) return byte;
procedure INIT_REGISTERS;
procedure SET_SERIAL_PORT (  THE_PORT                 : INTEGER;
                             THE_BAUD_RATE            : INTEGER;
                             THE_PARITY               : CHARACTER;
                             BITS_PER_CHARACTER       : INTEGER;
                             THE_STOP_BITS            : INTEGER );
```

Figure 5.4a. Subprogram specifications in the Ada-like language.

As may be seen from a comparison of Figs. 5.4a and 5.4b, the translation of subprogram specifications is a fairly literal process.

```
unsigned char char_length_convert(int the_length);
void init_registers(void);
void set_serial_port(      int the_port,int the_baud_rate,char the_parity,
                           int bits_per_character,int the_stop_bits);
```

Figure 5.4b. Translation of subprograms in Fig. 5.4a.

Next are specified some Interrupt Service Routines (ISRs). These tasks are invoked by a hardware interrupt mechanism which is triggered by a hardware event. They are tied into that hardware interrupt mechanism by having their code starting address placed/located at a dedicated place in memory. This place is called a vector and there is one such vector per hardware interrupt source. The clause which attaches the task name to the memory location, via the keyword *use_at*, is called a representation clause.

```
task SERIAL_PORT_0_ISR;                  --for on-chip serial port 0
for  SERIAL_PORT_0_ISR  use_at 52;       --located at memory address 52

task SERIAL_PORT_1_ISR;                  --for on-chip serial port 1
for  SERIAL_PORT_1_ISR  use_at 68;       --located at memory address 68
```

Figure 5.5a. Specification of ISR tasks.

The translated ISR tasks are distinguished by being declared of type *void interrupt*. The keyword *interrupt* tells the C compiler to protect the machine state by saving all CPU registers on the stack and, in addition, to terminate the subprogram by a "return from interrupt" instruction instead of a subprogram return. As C does not provide for automatic location of ISRs at a particular vector, the translator stores the location of the ISR vectors for future reference when producing the location code.

```
void interrupt serial_port_0_isr(void);
void interrupt serial_port_1_isr(void);
```

Figure 5.5b. ISRs in C are distinguished by the keyword *interrupt*.

The following co-routine tasks are distinguished from hardware event driven tasks (ISRs) by not being located at any memory address (Fig 5.6a).

```
task CONSUMER;          -- a co-routine task
task PRODUCER;          -- a second co-routine task
```

Figure 5.6a. Declaration of co-routine tasks.

Once again there is no way of informing the C compiler of these subprograms' special status, but the translator stores this for reference when translating the co-routine bodies. Their manner of returning is that of a normal function, and the C compiler's code generator determines which machine registers should be saved and restored in the subprogram's prologue and epilogue.

```
void consumer(void);
void producer(void);
```

Figure 5.6b. Co-routine task specifications translate to parameterless subprograms.

Constants, as seen in Fig. 5.7a, are declared in simular manner to those in Ada. However, their names are never used in the translated "Safe C" as every reference to them is type-checked, whereupon their value substituted in place of the name.

```
SERIAL_PORT_0 : constant INTEGER := 0;
SERIAL_PORT_1 : constant INTEGER := 1;
```

Figure 5.7a. Constant declarations in the Ada-like language.

Some global variables, of type byte (unsigned 8 bits [0..255]) and boolean ([True/False]) are declared in Fig5.8a. These may be used to pass data where co-routine tasks need to communicate.

```
RECEIVED_CHAR_0,
RECEIVED_CHAR_1     : byte;

CHAR_WAITING_AT_0,
CHAR_WAITING_AT_1 : boolean;
```

Figure 5.8a. Ada-like declaration of variables.

Where variables are global, the translator prefixes their type with the C keyword *volatile*. This prevents any attempt at optimisation by the C compiler where they would be vulnerable to becoming inadvertently overwritten. An instance of such optimisation is where variables are passed as parameters within CPU registers, instead of via the slower, but more secure, stack.

```
volatile unsigned char received_char_0;
volatile unsigned char received_char_1;

volatile BOOLEAN char_waiting_at_0;
volatile BOOLEAN char_waiting_at_1;
```

Figure 5.8b. Translated global variables in "Safe C" are declared as *volatile*.

Recall that for purposes of low-level control, the Ada-like language permits the representation of a hardware location by a name (e.g. in Fig. 5.9a). These located variable registers are hardware registers which must be configured, overwritten and read in order to make the machine work under low-level program control. Note that the names (e.g. BRG0) replicate those used throughout the V25 CPU documentation.

```
--The first examples are of memory-mapped I/O ports. The names are located at
--hardware locations through the use of the keyword use_at in their representation
-- clause.
BRG0,   --Baud Rate Generator 0
SCC0    --Serial Communication Control Register 0
        : byte;
for BRG0   use_at 1015658;
for SCC0   use_at 1015657;

--For I/O mapped ports, the use of the representation clause to attach a name to an I/O
--mapped address differs only in the use of the keyword map_at.
REAL_TIME_SECONDS_PORT : BYTE;
for REAL_TIME_SECONDS_PORT map_at 2;
```

Figure 5.9a. Ada-like declaration of hardware registers, both memory and I/O mapped.

There is no way of meaningfully passing this location data to the C compiler. Consequently, the translator declares these variables as normal globals, retaining the location data for substitution into C pre-processor macros when the located variables are referenced in sub-programs.

```
//first the two memory mapped I/O ports
volatile unsigned char brg0;
volatile unsigned char scc0;

//followed by the I/O mapped port
volatile unsigned char real_time_seconds_port;
```

Figure 5.9b. Hardware register declarations in C cannot be distinguished from normal variables.

This concludes the extracts from the global specification area and their "Safe C" translated counterparts. Extracts from the second and third areas: local specifications and subprogram bodies will now be examined in similar manner.

The function in Fig. 5.10a returns a value, ready for insertion into the on-chip Serial Communications Control Register, according to the Baud rate. It serves to demonstrate application of the case statement, variable assignments and use of recursion.

```
function SET_SCC( THE_BAUD_RATE : INTEGER) return BYTE is
RETURN_VAR  : BYTE;
begin
 case THE_BAUD_RATE is
   when 110      => RETURN_VAR := 8;
   when 150      => RETURN_VAR := 7;
   when 300      => RETURN_VAR := 6;
   when 600      => RETURN_VAR := 5;
   when 1200     => RETURN_VAR := 4;
   when 2400     => RETURN_VAR := 3;
   when 4800     => RETURN_VAR := 2;
   when 9600     => RETURN_VAR := 1;
   when 19200    => RETURN_VAR := 0;
   when others   => RETURN_VAR := SET_SCC( 9600);
 end_case;

 return RETURN_VAR;
end SET_SCC;
```

Figure 5.10a. An Ada-like subprogram body.

Function SET_SCC is translated to C in a fairly literal manner, showing how effective the C language is when used as an assembly language for a High Level Language. As the translated C code has been automatically generated, nested indentation levels, used to aid readibility, were not practical. To compensate for this, comments in the C source code are automatically inserted by the translator (e.g. at the end of all constructs) to aid in its legibility. This proved very useful to the author during the crafting of the translator when inspecting the translated code for correctness! However, for readability purposes within this section, indentation is employed.

```
unsigned char set_scc(int the_baud_rate){

unsigned char return_var;
  switch( the_baud_rate ){
    case 110:       return_var= 8; break;
    case 150:       return_var= 7; break;
    case 300:       return_var= 6; break;
    case 600:       return_var= 5; break;
    case 1200:      return_var= 4; break;
    case 2400:      return_var= 3; break;
    case 4800:      return_var= 2; break;
    case 9600:      return_var= 1: break;
    case 19200:     return_var= 0; break;
    default :       return_var=  set_scc(9600); break;
  } /*switch*/
  return return_var;
}/* end of set_scc*/
```

Figure 5.10b. "Safe C" translation of the SET_SCC function in Fig 5.10a.

The on-chip registers must be properly set in order for the hardware to perform correctly. This subprogram, Set_Serial_Port, demonstrates how much setting is necessary to coax a serial port into life. It also serves to demonstrate the flexible parameter passing mechanism, bitwise operations (and & or) on registers and data, and the use of constants to increase code readability. Note the use of the above subprogram (SET_SCC) to load the hardware registers SCC0 and SCC1.

```
procedure SET_SERIAL_PORT (  THE_PORT                  : INTEGER;
                             THE_BAUD_RATE             : INTEGER;
                             THE_PARITY                : CHARACTER;
                             BITS_PER_CHARACTER         INTEGER;
                             THE_STOP_BITS             : INTEGER ) is

ASYNCH                    : constant BYTE := 1;
TX_READY                  : constant BYTE := 128;
RX_ENABLE                 : constant BYTE := 64;
ERROR_INT_DISABLE_MASK    : constant BYTE := 71; --disable error Interrupts
TX_ENABLE                 : constant BYTE := 64; -- enable Tx generally
BIT_PARAMS : BYTE;
begin
 BIT_PARAMS := ASYNCH;
 BIT_PARAMS := BIT_PARAMS or TX_READY;
 BIT_PARAMS := BIT_PARAMS or RX_ENABLE;
 BIT_PARAMS := BIT_PARAMS or PARITY_CONVERT( THE_PARITY);
 BIT_PARAMS := BIT_PARAMS or CHAR_LENGTH_CONVERT( BITS_PER_CHARACTER);
 BIT_PARAMS := BIT_PARAMS or STOP_BITS_CONVERT(THE_STOP_BITS);
 case THE_PORT is
  when 0 =>
   BRG0  := SET_BRG (THE_BAUD_RATE);
   SCC0  := SET_SCC (THE_BAUD_RATE);
   SCM0  := BIT_PARAMS;
   SEIC0 := SEIC0 and ERROR_INT_DISABLE_MASK;
   SEIC0 := SEIC0 or 64;
   SRIC0 := SRIC0 and 7;
   STIC0 := STIC0 and 199;
   STIC0 := STIC0 or 64;
  when 1 =>
   BRG1  := SET_BRG (THE_BAUD_RATE);
   SCC1  := SET_SCC (THE_BAUD_RATE);
   SCM1  := BIT_PARAMS;
   SEIC1 := SEIC1 and ERROR_INT_DISABLE_MASK;
   SEIC1 := SEIC1 or 64;
   SRIC1 := SRIC1 and 7;
   STIC1 := STIC1 and 199;
   STIC1 := STIC1 or 64;
  when others => null;
 end_case;
end SET_SERIAL_PORT;
```

Figure 5.11a. Priming an on-chip serial port for action.

The "Safe C" production of SET_SERIAL_PORT is a fairly literal translation of the source text in that variables are assigned the prescribed values, and have operations performed upon them. It can be seen, however, that the translator is doing considerable "behind the scenes" work in providing code to handle the memory-mapped variable names, which have their assignments achieved via the mem_var_get and mem_var_put operators. These operators are, in fact, macros which the pre-processor will resolve at compile time. Due to the peculiar segmented addressing needs of the 8086 family of CPUs, the translator has calculated the segment and offset which determine the actual location within the CPU's twenty-bit address space.

```
void set_serial_port(       int the_port,int the_baud_rate,char the_parity,int bits_per_character,
                            int the_stop_bits){

unsigned char bit_params;
 bit_params= 1;
 bit_params= bit_params | 128;
 bit_params= bit_params | 64;
 bit_params= bit_params | parity_convert(the_parity);
 bit_params= bit_params | char_length_convert(bits_per_character);
 bit_params= bit_params | stop_bits_convert(the_stop_bits);
 switch( the_port ){
  case 0:
        mem_var_put( 61440, 32618, set_brg(the_baud_rate));
        mem_var_put( 61440, 32617, set_scc(the_baud_rate));
        mem_var_put( 61440, 32616, bit_params);
        mem_var_put( 61440, 32620, mem_var_get( 61440, 32620) &71);
        mem_var_put( 61440, 32620, mem_var_get( 61440, 32620) | 64);
        mem_var_put( 61440, 32621, mem_var_get( 61440, 32621) &7);
        mem_var_put( 61440, 32622, mem_var_get( 61440, 32622) &199);
        mem_var_put( 61440, 32622, mem_var_get( 61440, 32622) | 64);
        break;
  case 1:
        mem_var_put( 61440, 32634, set_brg(the_baud_rate));
        mem_var_put( 61440, 32633, set_scc(the_baud_rate));
        mem_var_put( 61440, 32632, bit_params);
        mem_var_put( 61440, 32636, mem_var_get( 61440, 32636) &71);
        mem_var_put( 61440, 32636, mem_var_get( 61440, 32636) | 64);
        mem_var_put( 61440, 32637, mem_var_get( 61440, 32637) &7);
        mem_var_put( 61440, 32638, mem_var_get( 61440, 32638) &199);
        mem_var_put( 61440, 32638, mem_var_get( 61440, 32638) | 64);
        break;
  default : ; /* null statement */
        break;
 } /*switch*/
}/* end of set_serial_port*/
```

Figure 5.11b. "Safe C" production of procedure SET_SERIAL_PORT.

Procedure Put_Char demonstrates an instance where constructs with indeterminate execution times can be eliminated. On this machine it is necessary to wait until the Transmit buffer indicates it is empty (by setting bit 7 of register STIC0) before a fresh character may be transmitted. If this were to be accomplished with a *while* loop (e.g. *while* (STIC0 < 128) *loop null; end_loop*;) then clearly, if the transmit buffer never emptied, the program could loop endlessly. Accordingly, the translator rejects any *while* loops whose termination is dependant upon a located variable. The *for* loop used in this instance is a preferable mechanism because it is deterministic.

```
procedure PUT_CHAR(   TO_THE_PORT : INTEGER;
                      THE_CHAR    : CHARACTER) is
LOOP_COUNTER : INTEGER;
begin
 if (TO_THE_PORT = 0) then
  for  LOOP_COUNTER in 1 .. 10000 loop
   if (STIC0 >= 128)                          --test if we can send the char
   then
    STIC0 := STIC0 and 127;                   -- remove buffer empty flag
    TXB0  := BYTE( THE_CHAR);                 -- demonstrate an explicit type-cast
                                              -- while putting the char into the
                                              -- transmission register
    exit;                                     -- then get out of the loop
   end_if;
  end_loop;
 else
  for  LOOP_COUNTER in 1 .. 10000  loop       --as above, but for Serial port 1
   if (STIC1 >= 128)  then                    -- can we send the char?
    STIC1 := STIC1 and 127;
    TXB1  := BYTE( THE_CHAR);
    exit;
   end_if;
  end_loop;

 end_if;

end PUT_CHAR;
```

Figure 5.12a. Sending a character from a serial port is an instance where indeterminate constructs can be eliminated.

Procedure PUT_CHAR demonstrates the code necessary to achieve an exit from within a loop. Whilst the C language does provide the keyword *break*, it is not possible to use this as an exit from within a loop in all situations. For example, C does not provide a natural way to break out of a loop from within a switch-case statement. A workable alternative is to always use a computed goto statement as demonstrated here. In order to achieve this, the translator has to create a unique label for every *end_loop*, in case an exit is required. When this occurs the exit statement is matched to the *end_loop*. The EXIT statement which appears in the C code below is in fact a macro that the pre-processor will convert to the *goto* keyword.

Again, note the translator provided comments (e.g. for an explicit type cast) inserted to make reading the "Safe C" translation easier, and the translator's automatic recognition and treatment of memory-mapped variables in Fig. 5.12b.

```
void put_char(int to_the_port,char the_char){

int loop_counter;
 if( to_the_port == 0 ){
  for( loop_counter=1; loop_counter<=10000; loop_counter++){/*(Loop Label=L1:)*/
   if( mem_var_get( 61440, 32622) >= 128 ){
    mem_var_put( 61440, 32622, mem_var_get( 61440, 32622) &127);
    mem_var_put( 61440, 32610, (BYTE)the_char/*Explicit Type Cast*/);
    EXIT L1;
   }/*end if*/
  } L1: ;
 } else {/*else part*/
  for( loop_counter=1; loop_counter<=10000; loop_counter++){/*(Loop Label=L2:)*/
   if( mem_var_get( 61440, 32638) >= 128 ){
    mem_var_put( 61440, 32638, mem_var_get( 61440, 32638) &127);
    mem_var_put( 61440, 32626, (BYTE)the_char/*Explicit Type Cast*/);
    EXIT L2;
   }/*end if*/
  } L2: ;
 }/*end if*/
}/* end of put_char*/
```

Figure 5.12b. The translation of PUT_CHAR, showing the loop EXIT mechanism.

Interrupt Service Routine (ISR) tasks differ from all other types of subprogram in that they are invoked, asynchronous to the main program flow, by a hardware mechanism. When task SERIAL_PORT_0_ISR was declared, it was attached to a hardware interrupt vector by means of a representation clause (see Fig. 5.5a). When a character arrives at serial port 0, the hardware interrupt mechanism associated with serial port 0 causes task SERIAL_PORT_O_ISR to be invoked. Its code causes the arrived character and any error condition to be read from appropriate registers (RXB0 and SCE0 respectively). Both items are placed in a Global variable for subsequent collection by the consumer co-routine. It then remains for the interrupt control register (SRIC0) and error control register (SEIC0) to be cleared of their interrupt condition before the subprogram terminates. Interrupt Service Routines should exhibit concise code so that they may complete their execution, then rapidly restore the machine to be ready for the next interrupt.

```
task SERIAL_PORT_0_ISR is
begin
  ERROR_COND_0     := SCE0;
  RECEIVED_CHAR_0  := RXB0;
  CHAR_WAITING_AT_0 := TRUE;

  SRIC0 := SRIC0 and 127; -- clear this interrupt
  SEIC0 := SEIC0 and 127; -- and any pending error interrupt

end SERIAL_PORT_0_ISR;
```

Figure 5.13a. An Ada-like ISR, which collects an incoming character.

Apart from the use of the C keyword interrupt, the ISR task looks like a normal C function. However, it is necessary for the translator to automatically insert the expression FINT in order to satisfactorily terminate an internal interrupt for the V25 chip. The expression FINT is expanded by a macro in "standard.c" (listed in Appendix F) to produce appropriate code for the V25 CPU.

```
void interrupt serial_port_0_isr(void){

error_cond_0= mem_var_get( 61440, 32619);
 received_char_0        = mem_var_get( 61440, 32608);
 char_waiting_at_0      = true;

 mem_var_put( 61440, 32621, mem_var_get( 61440, 32621) &127);
 mem_var_put( 61440, 32620, mem_var_get( 61440, 32620) &127);

 FINT;
}/* end of serial_port_0_isr*/
```

Figure 5.13b. Translation of the ISR in Fig 5.13a.

The largest area of "behind the scenes" work done by the translator is that of automatic production of management code for co-routine tasks, which differ substantially from all other subprogram types covered so far. To demonstrate this, both Ada-like and "Safe C" code for the two co-routine tasks: PRODUCER and CONSUMER, will now be examined.

PRODUCER (Fig 5.14b) polls the real-time clock "seconds" register (at I/O mapped variable REAL_TIME_SECONDS_PORT), logging and flagging the occurrence of every fresh second. In order to make life easier for CONSUMER, PRODUCER transforms the raw BCD coded "seconds" byte into two representative characters, in readiness for transmission. Explicit type casts are necessary to convert the BYTE data into CHARACTER data to be stored in readiness for collection by CONSUMER.

```
task PRODUCER is
TEMPORARY_BYTE: BYTE;

ASCII_ORDINAL_0 : constant BYTE := 48;  --'0' position in the ASCII table

UPPER_NYBBLE   : constant BYTE := 240;
LOWER_NYBBLE : constant BYTE := 15;

begin
 if ( REAL_TIME_SECONDS_PORT /= THE_SECONDS_STORE ) then
   -- First log the new value
   THE_SECONDS_STORE := REAL_TIME_SECONDS_PORT;

   -- We need to split up the 10's and Units columns of the seconds
   -- First the 10's or Most Significant Digit...
   -- isolate upper 4 bits, in which we will find the '10's column
   TEMPORARY_BYTE := THE_SECONDS_STORE and UPPER_NYBBLE;
   -- move them into the lower 4 bit area
   TEMPORARY_BYTE := TEMPORARY_BYTE / 16;
   -- make this into the byte representation of an ASCII character
   TEMPORARY_BYTE := TEMPORARY_BYTE + ASCII_ORDINAL_0;
   -- and save this Most Significant Digit
   -- in readiness for the Consumer process
   -- n.b. the use of explicit type-cast
   SECONDS_MSD := CHARACTER( TEMPORARY_BYTE);


   -- now isolate the 'units' component of the "seconds"
   TEMPORARY_BYTE := THE_SECONDS_STORE and LOWER_NYBBLE;
   -- make this into the byte representation of an ASCII character
   TEMPORARY_BYTE := TEMPORARY_BYTE + ASCII_ORDINAL_0;
   -- and save this Least Significant Digit
   -- in readiness for the Consumer process
   SECONDS_LSD := CHARACTER( TEMPORARY_BYTE);

   NEW_SECOND       := TRUE;
 end_if;

end PRODUCER;
```

Figure 5.14a. PRODUCER- the simpler of the co-routine tasks.

The translator recognises that PRODUCER is a co-routine, but because it has only one job to perform (i.e. the storage of new "seconds" when they occur), it does not need to generate any co-routine management code. Of interest though, is the translation of the routine reading I/O mapped variable REAL_TIME_SECONDS_PORT, which is accomplished through the macro "i_o_var_get".

```
void producer(void){

unsigned char temporary_byte;
/*Co-routine Management Section*/
//note the use of the macro "i_o_var_get" to read the I/O mapped variable
//REAL_TIME_SECONDS_PORT, which was located at I/O address 2.
  if( i_o_var_get( 2) != the_seconds_store ){
    the_seconds_store       = i_o_var_get( 2) ;
    temporary_byte          = the_seconds_store &240;
    temporary_byte          = temporary_byte / 16;
    temporary_byte          = temporary_byte + 48;
    seconds_msd             = (char)temporary_byte/*Explicit Type Cast*/;
    temporary_byte          = the_seconds_store &15;
    temporary_byte          = temporary_byte + 48;
    seconds_lsd             = (char)temporary_byte/*Explicit Type Cast*/;
    new_second              = true;
  }/*end if*/
}/* end of producer*/
```

Figure 5.14b. Translation for the PRODUCER co-routine of Fig 5.14c.

The CONSUMER task, in reality, has three distinct jobs to do in life:

- If a character has been received and flagged by the RS232 (Serial port 0) ISR, then that character is transmitted from of the RS422 port.
- Similarly, if a character eventuates at the RS422 (Serial port 1) ISR, that character is sent to the RS232 port.
- If a fresh second has been logged and flagged by PRODUCER, then the second's character values, as prepared in PRODUCER, are transmitted from the RS232 serial port. These are followed by the transmission of a new-line sequence.

On each invocation, CONSUMER will only perform one of these jobs, exiting at the next SYNCH statement encountered. It is essential, however, that upon its next invocation, CONSUMER will resume execution at the point immediately after that SYNCH statement or, if none exists, at the first job.

```
task CONSUMER is
TEMPORARY_CHAR : CHARACTER;
begin
 if( CHAR_WAITING_AT_0 ) then                                    -- this is job number 1.
  TEMPORARY_CHAR :=CHARACTER(RECEIVED_CHAR_0);
  CHAR_WAITING_AT_0 := FALSE;
  PUT_CHAR( SERIAL_PORT_1, TEMPORARY_CHAR);
 end_if;
 SYNCH;

 if( CHAR_WAITING_AT_1 ) then                                    -- this is job number 2.
  TEMPORARY_CHAR :=CHARACTER(RECEIVED_CHAR_1);
  CHAR_WAITING_AT_1 := FALSE;
  PUT_CHAR( SERIAL_PORT_0, TEMPORARY_CHAR);
 end_if;
 SYNCH;

 if (NEW_SECOND) then                                            -- this is job number 3.
  --Transmit the seconds, most significant digit first
  PUT_CHAR( SERIAL_PORT_0, SECONDS_MSD);
  PUT_CHAR( SERIAL_PORT_0, SECONDS_LSD);

  TEMPORARY_CHAR := CHARACTER(13);
  PUT_CHAR( SERIAL_PORT_0, TEMPORARY_CHAR); -- send a carriage return
  TEMPORARY_CHAR := CHARACTER(10);
  PUT_CHAR( SERIAL_PORT_0, TEMPORARY_CHAR); -- and line feed .

  NEW_SECOND := FALSE;
 end_if;
 SYNCH;

end CONSUMER;
```

Figure 5.15a. Consumer - a hard-working co-routine!

```
void consumer(void){
char temporary_char;
/*Co-routine Management Section*/
//num_of_synch_points is a static variable, which will retain its value between invocations of this
//subprogram.
 static int num_of_synch_points = 1;
 switch (num_of_synch_points){
 case  1:goto T3; break;
 case  2:goto T4; break;
 case  3:goto T5; break;
 }/* switch num_of_synch_points*/
 /*End of Co-routine Management Section*/
 T3: ;
 if( char_waiting_at_0 ){
 temporary_char= (char)received_char_0/*Explicit Type Cast*/;
 char_waiting_at_0= false;
 put_char(1,temporary_char);
 }/*end if*/
 if(++num_of_synch_points >  3)
   num_of_synch_points = 1;
 return; /* Synch point*/

 T4: ;
 if( char_waiting_at_1 ){
 temporary_char= (char)received_char_1/*Explicit Type Cast*/;
 char_waiting_at_1= false;
 put_char(0,temporary_char);
 }/*end if*/
 if(++num_of_synch_points >  3)
   num_of_synch_points = 1;
 return; /* Synch point*/

 T5: ;
 if( new_second ){
 put_char(0,seconds_msd);
 put_char(0,seconds_lsd);
 temporary_char= (char)13/*Explicit Type Cast*/;
 put_char(0,temporary_char);
 temporary_char= (char)10/*Explicit Type Cast*/;
 put_char(0,temporary_char);
 new_second= false;
 }/*end if*/
 if(++num_of_synch_points >  3)
   num_of_synch_points = 1;
 return; /* Synch point*/

}/* end of consumer*/
```

Figure 5.15b. "Safe C" production for CONSUMER.

Clearly, the translator has to do substantial "behind the scenes" work (compare Figs. 5.15a and 5.15b) to manage the relinquishment and subsequent resumption of execution in an orderly and correct manner. Use is made of C's static variables which, when used within a subprogram, may be initialised once and will maintain their last written value between

subprogram invocations. The mechanics of this are that they occupy global variable space, but are only visible from within the subprogram in which they are declared.

The co-routine management code consists of two distinct parts:

- on entry to the co-routine, a decision is made, based upon the value of a static variable which maintains the next job to be done, in order to vector execution to labels placed at the start of each of the co-routine task's jobs;
- before relinquishment of execution, that static variable must be left containing the correct value for resumption at the next job upon any subsequent iteration of the co-routine task.

In order to achieve this, the translator has to inspect the source code of the co-routine to count how many SYNCH points exist within it. When this is known, the switch-case statement which controls the job sequencing can be inserted into the C code. The operational and assignment statements of the task are processed as for any other subprogram, except in the case of SYNCH statements. Before relinquishing control at the SYNCH statement, the static variable is updated, round-robin style, with the value of the next job to be done. When the last job is done, the co-routine management code reverts the execution resumption to the first job.

Code for last subprogram type, *driver*, is to be found in Figs. 5.16a and 5.16b, on the two ensuing pages. This programs's driver, TEST_PROG, initialises the global variables, turns off the machine interrupts, initialises the serial ports, transmits a 'hello' test message from each serial port and then enters an interminable loop calling each of the co-routine tasks in their turn. Note that the ' hello' message is generated within a loop which is terminated with a conditional exit statement.

```
driver test_prog is

the_letter : character;
letter_num,
loop_counter : integer;

begin
  --initialise inter-process data
  NEW_SECOND       := FALSE;
  CHAR_WAITING_AT_0 := FALSE;
  CHAR_WAITING_AT_1 := FALSE;

  -- disable interrupts, as we don't want any occurring so that our setup code isn't interrupted
  disable_int;

  init_registers; --set general purpose output registers

set_serial_port(SERIAL_PORT_0 , 9600, 'n', 8, 1); -- set up serial-port 0 (the RS-232 port)

set_serial_port(SERIAL_PORT_1 , 4800, 'n', 8, 1); -- and serial port 1 (the RS422 port)

  -- now that the ports can cope with interrupts, we may safely enable the interrupts.
  enable_int;

  -- set the serial port handshaking lines to their active state,
  -- so that characters may be transmitted and received.
  SET_DTR(SERIAL_PORT_0);
  SET_DTR(SERIAL_PORT_1);
  -- now a little wake-up message
  letter_num := 1;
  while (true) loop
    case letter_num is
      when 1=> the_letter := 'H';
      when 2=> the_letter := 'e';
      when 3=> the_letter := 'l';
      when 4=> the_letter := 'l';
      when 5=> the_letter := 'o';
      when others => null;
    end_case;
    put_char( SERIAL_PORT_0, the_letter);
    put_char( SERIAL_PORT_1, the_letter);
    loop_counter := loop_counter+1;
    exit when (loop_counter > 5);
  end_loop;
  --now service the co-routines forever...
  while (TRUE) loop
    PRODUCER;
    consumer;
  end_loop;
end test_prog;
```

Figure 5.16a. Ada-like source for driver subprogram TEST_PROG.

```
void test_prog(void){
int letter_num;
int loop_counter;
char the_letter;
  Set_Vector( 52, serial_port_0_isr);
  Set_Vector( 68, serial_port_1_isr);
  new_second= false;
  char_waiting_at_0= false;
  char_waiting_at_1= false;
  disable_int;
  init_registers();
  set_serial_port(0,9600,'n',8,1);
  set_serial_port(1,4800,'n',8,1);
  enable_int;
  set_dtr(0);
  set_dtr(1);
  letter_num= 1;
  while( true ){/*(Loop Label=L6:)*/
    switch( letter_num ){
      case 1:  the_letter= 'h'; break;
      case 2:  the_letter= 'e'; break;
      case 3:  the_letter= 'l'; break;
      case 4:  the_letter= 'l'; break;
      case 5:  the_letter= 'o'; break;
      default :  ; /* null statement */ break;
    } /*switch*/
    put_char(0,the_letter);
    put_char(1,the_letter);
    loop_counter= loop_counter + 1;
    if( loop_counter > 5) EXIT L6;
  } L6: ;
  while( true ){/*(Loop Label=L7:)*/
    producer();
   consumer();
  } L7: ;
}/* end of test_prog*/
```

Figure 5.16b. "Safe C" production for driver subprogram TEST_PROG

Before any statements can be executed in the Ada-like source code, the "Safe C" translation of the driver subprogram (see Fig. 5.16b on the previous page) must resolve all interrupt vector initialisation for ISR tasks. Once again this may be achieved through a macro which the pre-processor will resolve, and Set_Vector is passed the name of the ISR vector (which in C is a pointer to the subprogram's code) and the memory location of the vector. Whilst this, and other areas where macros have been relied upon, would be possible to accomplish explicitly in Safe C, the use of pre-processor macros assist greatly with potential portability in that only the macros (contained in "standard.c") need be changed between target processor implementations.

There remains but one thing to be done as far as the C program is concerned. All C programs must have a main() subprogram, which is the entry point from the startup code. In this instance, the translator manufactures the code for main() subprogram, which merely calls the driver subprogram (in this instance test_prog) named in the Ada-like source.

```
void main(void){
test_prog();}/* main*/
```

Figure 5.17 main() calls the driver TEST_PROG.

5.2 Evidence found that supports each of the Research Questions

The Main Research Question's components, followed by the Main Question itself, are re-stated and addressed in turn.

a) "What elements of computer languages are desirable to express solutions as low cost, medium performance, embedded systems?"

Desirable element	Brief reason for desirability	Demonstrated in...
strong typing	allows the program to model realistically the data of the real world, yet minimise programmer induced errors in data transfer	Fig. 5.12, Fig. 5.14, Fig. 5.15.
flexible parameter passing mechanisms	permits subprograms to communicate with the world outside their frame of operations through the passing of parameterised values	Fig. 5.11, Fig. 5.16.
recursion	allows an elegance of coding	Fig. 5.10
exception handling	permits the program to handle abnormal run-time events and recover gracefully	deferred for a later study
concurrency	in a Real-Time sense, allows the programmer adequate control of both timing and sequencing of operations	Fig. 5.6, Fig 5.14, Fig. 5.15
elimination of constructs with indeterminate execution times	constructs which do not automatically relinquish control of the processor present potential points at which a program can become non-deterministic	Fig. 5.12
interrupt types	allow rapid servicing of events which are asynchronous to normal program flow	Fig. 5.5, Fig. 5.13
modularity	when correctly applied, permits information hiding, more manageable code modules and enhances maintainability	Fig. 5.4, Fig. 5.11
low-level control	permits the programmer intimate control over the underlying machine	Fig. 5.9, Fig. 5.11
portability	as hardware develops, permits code to be configured rapidly for new processors	Fig. 5.3b, Appendix E

Figure 5. 18 Relating the desirable elements to evidence of this study's successful response to the research questions.

b) "Can the elements noted in (a) be expressed in an Ada-like language?"

Yes, this study has resulted in the production of a translator which, save for the deferred topic of exception handling, satisfies this question.

c) "Can those language elements be implemented using only verifiable elements of the C language?"

Yes, this study has established an O/S independant "Safe C" for the 8086 family of processors, which has proved to be effective with the Microsoft C version 6 compiler, thus providing an intermediate language to implement the Ada-like source code.

The main question:

"Can the desirable features for low cost, medium performance, embedded systems programming be provided in an Ada-like language, and then translated to "Safe C" to achieve reliable run-time efficiency? "

During chapter three, a list of these desirable features was developed, chiefly in response to the problems outlined in chapter two. In chapter four, the method by which this study proposed to address each of these features was outlined, (although the topic of exception handling was deferred to a later study). In this chapter a demonstration program was presented, written in an Ada-like language. Although small in scale the test program was representative of a typical real-world project and, when translated into "Safe C", was successfully embedded into real-world target hardware to predictably perform its required work. Based upon this evidence then, whilst noting the exception (sic) of exception handling, the answer to the main research question is in the affirmative.

5.3 Unanticipated Findings

There were no unanticipated findings in this study.

5.4 Summary

The target hardware represents a typical system in current use for medium-performance embedded systems and it allows demonstration of all of the significant features developed by this project. The target system is an off-the-shelf item, based on the NEC V25 microcontroller with 1Mbyte address range. It provides on-board support for EPROM, RAM, both memory-mapped and I/O-mapped I/O ports, including timers, serial-ports, parallel ports and a Real-Time clock. However, in order to configure the system to perform a useful task, substantial initialisation of on-chip registers and interrupt vectors are performed.

The test program, written in an Ada-like language, is carefully structured to:

- demonstrate the activities of the translator in response to the list of desirable features for real-time systems languages identified in chapter three; and
- be indicative of the type of work which could be expected from such a system.

It is perhaps worth noting that the test program required two iterations before the target system functioned correctly, i.e. debugging time was minimal.

Suitable annotated extracts from the test program are provided, together with the translated output expressed in "Safe C", showing the code that is generated to accomplish the desirable features.

Finally, based upon the demonstration incorporated in the test program, answers are provided to the Main Research Question and its components as stated in chapter two.

6 Conclusion

To introduce this final chapter, let us first examine the raison d'être for this project. The author, an embedded-systems engineer, has for some time been aware of the problems associated with implementing embedded-system software without the benefit of sophisticated and expensive development environments. In particular, the author observed from his own experiences and from those of similarly employed colleagues, that there is considerable time spent in debugging embedded software, often due to errors which could have been avoided by stronger typing. Whilst the presence of a good debugger can greatly assist in discovering bugs of this nature, the author came to feel that debugging time could be reduced if such preventable errors were trapped. The author has used the Ada and C languages extensively in substantial projects and has an awareness of the strengths and weaknesses of both languages. In short, as expressed in chapter one, this project was born of a desire to harness in a formal manner the strengths of these languages in such a way as to reduce preventable coding errors, thus enhancing the reliability of an embedded implementation.

In formalising the problem during chapter two, it became necessary to define clearly the kind of embedded system with which the study concerns itself; to define the areas where the exclusive use of C or Ada is likely to cause problems; and, to suggest areas where a combination of the strengths of those languages may yield an advantage to the programmer.

The literature was then reviewed from the perspective of ascertaining the peculiar needs of embedded-systems software. The features identified as being desirable for embedded-systems and, in particular, those with Real-Time needs were formed into a list. This list formed a convenient framework for explaining and justifying each element, or desirable feature, which would be addressed in the project implementation. Potential alternative languages were examined to see the level of support they offered to the peculiar needs of Real-Time embedded systems. From this the Ada language, in spite of its shortcomings, emerged as the most complete. Having ascertained that Ada's shortcomings lie substantially in the area of deterministic concurrency and resource greediness, existing alternatives in these areas were explored for workable solutions. This concluded with the

emergent ideas of using a co-operating tasking mechanism to overcome the first shortcoming, and of employing a subset of the C language (known to have no O/S dependency) for efficient code generation to overcome the second.

In designing the pilot translator from the Ada-like language to "Safe C", in order to prove the worth of the emergent ideas, the list of desirable features was again used as a framework. The methods of implementing the response to each desirable feature were described, and a formal complete specification of the Ada-like language was developed to respond to the desirable features. This specification constitutes Appendix B of this document. In deference to its size and complexity, any response to the feature of exception handling was at this stage deferred to be a subject of a later study.

In order to test the translator a suitable hardware target system was chosen to exemplify the one defined in chapter two. A non-trivial test program was devised, written in the Ada-like language, to demonstrate the translator's ability to support the desirable language features, and which performed a workload indicative of that which could be expected in a real-world project. A variation of the classical theme of the producer - consumer relationship was chosen to test the pilot implementation for its handling of concurrency and Real-Time response to external events. Other issues, such as low-level control, were tested by the complex nature of initialising the machine's registers so that the system performed in a controlled manner. The combination of hardware and test software performed a dual task of real-time serial link protocol conversion and periodic timing transmission.

Having translated the source code of the test program, the "Safe C" output was compiled and then run in the target system to verify its effectiveness. Pertinent extracts of the Ada-like language source code were presented and examined together with their translated "Safe C" representation, now known to be working. This examination was based upon the list of desirable Real-Time language features and its findings from this examination were used to answer to the Main Research question and its components.

The study was directed toward achieving reliability and efficiency of coding for low-cost medium-performance embedded systems. The pilot implementation has shown that this is possible, retaining the many advantages of using a strongly-typed concurrent Ada-like language, yet still benefitting from the efficient compilation and code-generation of current C compilers.

Implications for embedded-systems engineers include:

- a potential reduction in debugging time, as errors from type-mismatches are made preventible;
- the availability of a natural expression for co-operative deterministic concurrency;
- automated attachment of interrupt service routine tasks to hardware interrupt vectors; and
- an increase in code portability.

There is no reason, however, why the techniques used in this project should be limited to embedded systems. Should the intermediate language be extended to include O/S dependancies, the co-operative tasking method employed would serve admirably in a single-tasking operating system (e.g. MS-DOS). With its use, competition for resources would be eliminated, as only one process would be allowed to enter the potentially sensitive O/S code.

Implications for future research include the addition of effective exception handling to the existing implementation, a quantitative analysis of the time penalty of the co-routine management overhead, a broadening of the existing implementation to embrace current Ada standards, and an adaptation of the co-operating concurrency model to include the eventuality of multiple-processors.

Appendix A: A Glossary of terms used in this document

Ada-like The United States Department of Defense, the effective owners of Ada, has "disallowed subsets" (MacLennan, 1987, p327) of Ada for portability reasons. Consequently, the resulting dialect of the study will be known as "an Ada-like language" or simply "Ada-like".

BCD **B**inary **C**oded **D**ecimal notation is where one (eight bit) byte contains a coded decimal digit [0000 .. 1010] in each of its pair of four bits.

Chip computing jargon for integrated circuit (I.C.). The chip is actually a reference to the slice of silicon circuitry which forms the working part of the integrated circuit package.

EPROM **E**rasable **P**rogrammable **R**ead **O**nly **M**emory. Read-only non-volatile semiconductor memory that is erasable in ultra-violet light and is reprogrammable.

.EXE file An executable file produced by MS-DOS linkers. The code and data contained may be re-located to available memory areas by the O/S according to information contained in a standard header at the start of the .EXE file.

I/O Input/Output. Two types of access are generally available:

- Programmed I/O, where data to and from Input/Output devices is accessed using special IN and OUT instructions from the CPU. These devices are mapped separately from main memory.
- Memory mapped I/O, where data is mapped together with main memory and accessed using similar instructions to those for normal declared variables.

JSD Jackson Structured Development. A method widely used to express the design of Real-Time systems.

.MAP file A file produced by MS-DOS linkers cross-referencing symbols (e.g. variable names and subprogram names) to re-locatable addresses within the executable (.EXE) file.

Monitor programs Small, supervisory, programs used to give a primitive level of facility and control over a computer system.

Orthogonality Where a language provides a relatively small set of primitive constructs that can be combined in a relatively small number of ways, and where every combination is legal and meaningful, to build the control and data structures of the language. Adapted from Sebesta, 1989, p6.

O/S Operating System which, in a general purpose computer, affords application programs the basic operations such as file handling, keyboard buffering, and facilities for general input and output (e.g. MS-DOS, OS/2, Unix). O/Ss are available for embedded systems (e.g. OS-9, QNX, VRTX32, PDOS.) and these afford more low level facilities.

PC Personal Computer. Includes those according to the IBM standard (80x86 based machines) and Apple Macintosh (680x0 based machines).

RAM Random Access Memory. Semiconductor read/write volatile memory. Data is lost if the power is turned off.

Real time system

(i) "Any system in which the time at which output is produced is significant."(Burns & Wellings, 1990, p2).

(ii) "Systems that handle asynchronous events in a timely and deterministic manner." (Isherwood, 1991). The word timely should be taken in context of the total system, with some systems requiring microsecond response, while others requiring response measured in seconds.

Re-entrant code may be interrupted in mid execution and then re-executed in a second, or subsequent, instance. A subprogram which may be interrupted in mid execution by, for example, an external hardware interrupt and then re-executed within the interrupt service routine is re-entrant.

Relocatable code may be loaded into memory at a location according to available space, then run at that location.

ROM Read Only Memory. Computer memory in which data can routinely be read, but written to once only using special means when the ROM is manufactured. The ROM is used for storing data or programs on a permanant basis.

RS232 Recommended Standard, number 232, is laid down by the Electronic Industry Association (EIA) for communications using bipolar voltages with respect to a common ground for serial communications.

RS422 Recomended Standard, number 422, is laid down by the Electronic Industry Association (EIA) for defining balanced (or differential) serial communications using two signal wires for each direction. Much higher transmission rates may be achieved than with RS232.

Run-time system A nucleus of frequently used routines which an executing program may rely upon to perform such functions as string operations, concurrency control and memory allocation.

Safe C A subset of the C language which is known to have no dependency on Operating System support for the implementation tested.

Simplicity A language with a small number of elementary components to learn is relatively simple to learn.

Target system The computer system, where the program being developed will reside, at the heart of which will be a specific microprocessor.

Validation Checks if the product's functions are what the customer really wants. As Boehm (1981) suggests 'Are we building the right product?'.

Verification Checks whether the product under construction meets the requirements definition. Often confused with validation. Boehm (1981) suggests 'Are we building the product right?'.

Appendix B: An Ada-like language

B.1 Grammar for an Ada-like language

Listed below is an Extended Backus Naur Form (EBNF) grammar for the Ada-like language developed in this project. Conventions used in this grammar description are:

- keywords and punctuation are depicted in **bold** font;
- non-terminal symbols are enclosed in < and >;
- terminal symbols are represented by themselves;
- items enclosed in square braces [ ] are optional
- items enclosed in curly braces { } may be iterated zero or more times;
- the symbol -› separates the left and right sides of a production. If no left side of a production is shown, the preceding left side applies.

<actual option> -› [(<id> {, <id>})]

<address literal> -› < digit > {< digit >} in the range 0..1048575

<alpha character> -› a..z

-› A..Z

alphanumeric character -› <alpha character>

-› _

-› <digit>

<ascii character> -› ascii NUL .. ascii DEL

<assignment statement> -› <id> := <expression>;

<basic loop> -› **loop** <statement> {<statement>}**end loop**

<bitwise operator> -› **and**

-› **or**

<boolean expression> -› **(** <factor> [<relational operator> <factor>] **)**

-› <unary operator> <factor>

<boolean literal> -› FALSE

-› TRUE

<byte literal> -› < digit > {< digit >} in the range 0..255

<call statement> -› <id> <actual option> ;

<case statement> -› **case** <id> **is** {<when list> }<others clause> **end_case ;**

<character literal> -› '<ascii character>'

<constant option> -› **[constant]**

<digit> -› 0..9

<discrete range> -› <factor> .. <factor>

<exit statement> -› **exit [when** <boolean expression>] ;

<expression> -› <factor> [<operator> <factor>]

-› <type cast>

<factor> -› <id>

-› <literal>

<float literal> -› [–]<digit > {< digit >}. < digit > {< digit >}

<formal part> -› (<parameter declaration list>)

<formal part option> -› [<formal part>]

<id> -› <alpha character>{<alphanumeric character>}

<id list> -› <id> {, <id>}

<if statement> -› **if** <boolean expression>

then <statement> {<statement>}

[**else** <statement> {<statement>}]

end_if ;

<initialisation option> -› := <value>

<integer literal> -› [–]<digit >{< digit >}

<interrupt control statement> -› **disable_int ;**

-› **enable_int ;**

<iteration clause> -› **for** <id> **in** <discrete range>

-› **while** <boolean expression>

<literal> -› <boolean literal>

-› <byte literal>

-› <character literal>

-› <float literal>

-› <integer literal>

-› <word literal>

<location clause>	-›	**map_at** <word literal>
	-›	**use_at** <address literal>
<logical operator>	-›	**and**
	-›	**or**
<loop statement>	-›	[<iteration clause>] <basic loop> **;**
<mode>	-›	**in**
	-›	**in_out**
	-›	**out**
<multiplying operator>	-›	*
	-›	/
<null statement>	-›	**null** ;
<object declaration>	-›	<id list> : <constant option> <type> <initialisation option>;
<operator>	-›	<bitwise operator>
	-›	<logical operator>
	-›	<multiplying operator>
	-›	<plus operator>
	-›	<relational operator>
	-›	<unary operator>
<others clause>	-›	**when others** => <statement> {<statement>}
<parameter declaration>	-›	<id list>: [<mode>] <type>
<parameter declaration list>	-›	<parameter declaration> {; <parameter declaration>}
<plus operator>	-›	+
	-›	–
<relational operator>	-›	=
	-›	/=
	-›	>=
	-›	<=
	-›	<
	-›	>
<representation clause>	-›	**for** <id> <location clause>
<return statement>	-›	**return** [<expression>] **;**

<statement>	-›	<assignment statement>
	-›	<call statement>
	-›	<case statement>
	-›	<exit statement>
	-›	<if statement>
	-›	<interrupt control statement>
	-›	<loop statement>
	-›	<null statement>
	-›	<return statement>
	-›	<synchronisation statement>
<subprogram body declaration>	-›	<subprogram specification> **is** {<object declaration>}
		begin {<statement>} **end** <id> **;**
<subprogram declaration>	-›	<subprogram specification> **;**
<subprogram specification>	-›	**driver** <id>
	-›	**function** <id> <formal part option> **return** <type>
	-›	**procedure** <id> <formal part option>
	-›	**task** <id>
<synchronisation statement>	-›	**synch ;**
<type>	-›	**address**
	-›	**boolean**
	-›	**byte**
	-›	**character**
	-›	**float**
	-›	**integer**
	-›	**word**
<type cast>	-›	<type> **(** <factor>**)**
<unary operator>	-›	**not**
<value>	-›	<id>
	-›	<literal>
<when list>	-›	**when** <integer literal> => <statement> {<statement>}
<word literal>	-›	< digit > {< digit >} in the range 0..65535

B.2 Tokens recognised by the scanner of the Ada-like language

Keywords:

address	and	begin	boolean	byte
case	character	constant	disable_int	driver
else	enable_int	end	end_case	end_if
end_loop	exit	float	for	function
if	in	in_out	integer	is
loop	map_at	not	null	or
others	out	procedure	return	synch
task	then	use_at	when	while
word				

Operators

Operator type		Evaluation order	Order of precedence
Unary	not	Right to Left	Highest
Multiplication	* /	Left to Right	
Plus	+ –	Left to Right	
Relational	= /= < <= > >=	Left to Right	
Bitwise	and or	Left to Right	
Logical	and or	Left to Right	Lowest

Punctuators

, .. : ; , ( ) := =>

Data types supported:

address boolean byte character float Integer word

B.3 Additional notes

Identifiers must begin with a letter and consist of letters, underscores and digits. Their length should be of less than twenty-nine characters. This allows for some additions to identifiers before they are submitted to the C compiler, whose general limit is thirty-two characters.

Literals

- Integer literals can contain only digits and minus signs. They must start with a digit or minus sign and are legal in the range -32767..32768.
- Float literals can contain digits, a minus sign, a single decimal point. They must start with a minus sign or a digit, and a digit must both precede and follow the decimal point.
- Character literals are characters within single quotes e.g. 'c'.
- Byte literals are digits and are legal in the range 0..255.
- Boolean literals are either TRUE or FALSE.
- Word literals consist of digits and are legal in the range 0..65535.
- Address literals consist of digits and are legal in the range 0..1048575. The upper limit refelcts the twenty bit address range of the target machine.

Comments are prefixed by the compound symbols -- and conclude at the next new-line.

Visibility and scope of variables:

Two levels of scope are provided: namely, global for variables which are visible from all subprograms, and local scope for variables declared locally within subprograms. Parameters have the same visibility as locally declared variables. Local variables and constants obscure similarly named global declarations.

Case sensitivity is not significant, in fact all characters are read as lower case.

Tokens are separated by white space. They are, therefore, not allowed to persist beyond a new-line in the source text.

Appendix C: "Safe C".

Safe C: Operating system hosted compilers feature, within their run-time libraries, dependencies on Operating System (O/S) support, which are to be avoided in embedded programs. These are often in the form of input and output techniques to console and file buffers provided by the O/S. Other dependencies may manifest themselves via requests for dynamic memory so that internal calculations may be performed, such as in numeric to character data conversion and formatting. However, there will certainly exist a small component of the run-time library which is not dependent on the operating system.

The following functions have been found by the author to be MSDOS independent in the case of Microsoft C version 6. The functions are shown by category together with the C header file in which their prototype may be found and a brief description. For more detail the user is referred to the literature, in particular the Microsoft C Runtime Library Reference.

Memory manipulation functions	**prototypes found in <memory.h>**
memcpy	copies contents of source area in memory to a destination area
memcmp	compares contents two areas in memory
memchr	seeks and returns the first occurrence of a character in the specified area of memory.
memset	sets all bytes in the specified area in memory to the supplied character

Character conversion functions	**prototypes found in < ctype.h>**
toascii	the ASCII value
tolower	the lowercase equivalent
toupper	the uppercase equivalent

Character classification functions	prototypes found in < ctype.h>
isalnum	alphanumeric character?
isalpha	alphabetic character?
isascii	ASCII character?
iscntrl	control character?
iscsym	letter, underscore, or digit?
iscsymf	letter or underscore?
isdigit	decimal digit?
isgraph	printable character, not space?
islower	lowercase letter?
isprint	printable character?
ispunct	punctuation character?
isspace	white-space character?
isupper	uppercase letter?
isxdigit	hexadecimal digit?

String Manipulation Functions	prototypes found in <string.h>
strcat	concatenates second string to first
strchr	finds the first occurrence of a given character in the string
strcmp	compares two strings lexicographically
strcpy	copies a string
stricmp	as strcmp but case insensitive
strlen	get the length of the string
strncat	concatenates n characters of second string to first
strncmp	as strcmp but only for n characters
strncpy	copys first n characters of a string
strnicmp	as strncmp but case insensitive
strnset	initialise n characters of a string to a given character
strrchr	find last occurence of a given character
strset	sets all characters of a string to a given character
strstr	find a substring
strtok	find next token in a string

Port I/O	prototypes found in <conio.h>
inp	the byte read from port
inpw	the word that was read
outp	the byte output
outpw	the word output

Data conversion	prototypes found in <stdlib.h>
atof	string to a floating point number
atoi	string to an integer number
atol	string to a long integer number
atold	string to a long double (floating point) number
ecvt	converts a double (floating point number) to a string, using a statically allocated buffer
fcvt	converts a floating point number to a string, using a statically allocated buffer
gcvt	converts a floating point number to a string, using the provided buffer
itoa	converts an integer number to a string
ltoa	as itoa but for long integer number
strtod	converts string to a double (floating point) number
strtol	converts string to a long integer number
strtoul	converts string to an unsigned long integer number
ultoa	converts unsigned long integer to a string

Math functions	prototypes found in <math.h>
atan	arctangent of supplied number.
floor	largest integer less than the supplied number
ceil	smallest integer greater than the supplied number
abs	absolute value of the supplied integer
fabs	absolute floating point value
fmod	floating point remainder
sin	sine of supplied number, in radians
cos	cosine of supplied number, in radians
tan	tangent of supplied number, in radians
log	natural logarithm of supplied number
sqrt	square-root of supplied number
exp	exponential of supplied number

Appendix D: Effective use of Ada Generics to realise the symbol trees.

The purpose of this appendix is to demonstrate the worth of Ada's generics with respect to the project by using, as an of example, some of the source code used in the implementation. Without the facilities offered by generics, which allow the re-use of one piece of code for more than one purpose, the implementation would have been much more onerous.

While processing the Ada-like source code for type checking and potential illegal symbol duplication, it was necessary for the translator to contain methods for rapid storage and retrieval of the encountered symbols. In all, five such symbol trees were necessary, one each for:

- global constants, held for the life of the progam's translation;
- global variables, held for the life of the progam's translation;
- local constants, held for the life of the current subprogam's translation;
- local variables, held for the life of the current subprogam's translation; and
- suprograms, held for the life of the progam's translation.

Ada's generics were used to advantage in providing an instantiation of a symbol tree for each of the five, based in turn, upon an unbalanced binary tree structure. Demonstration of the dependencies which exist upon each of the respective generics are shown in Figure D.1.

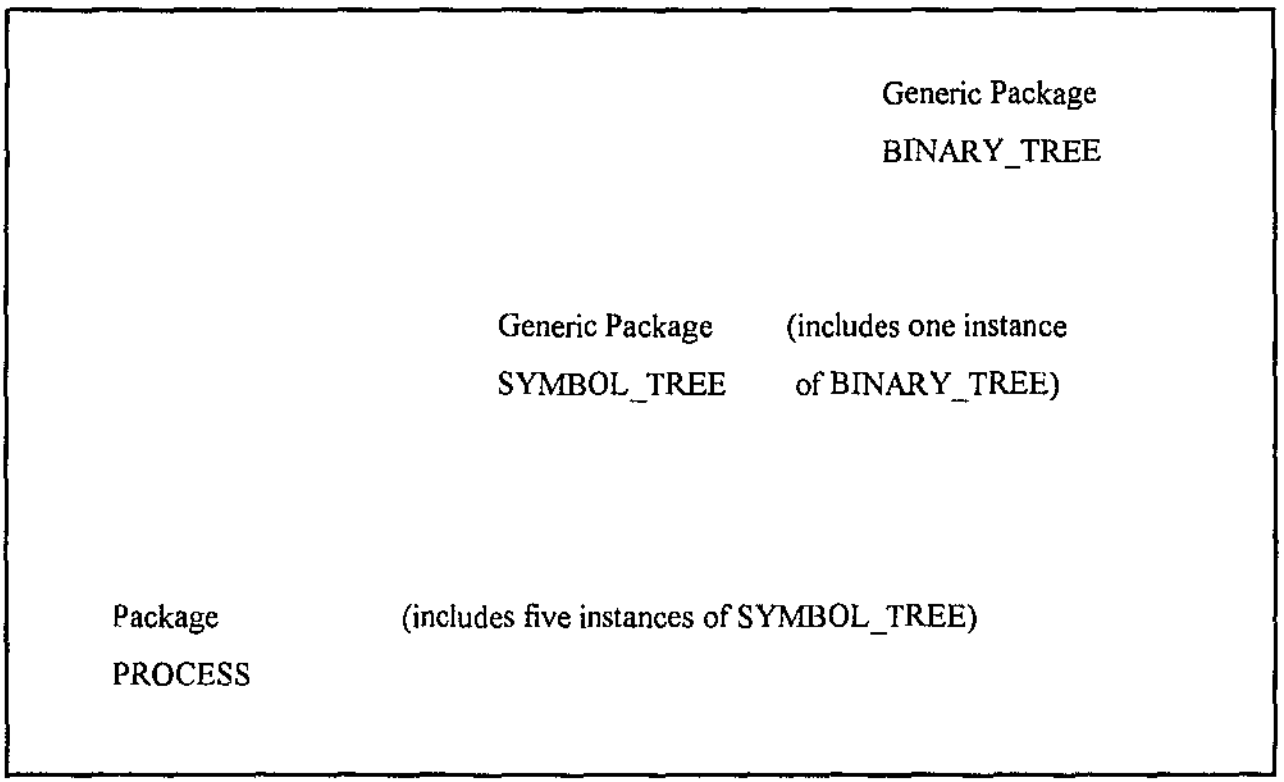

Figure D.1. Annotated dependency chart of symbol tree generic instantiations.

Complete listings are now given of both specification and bodies for generic packages BINARY_TREE and SYMBOL_TREE. The specification code necessary to achieve the five instantiations within package PROCESS is also provided.

```
--
--package specification for BINARY_TREE
--this package specification and its body are based upon the
--excellent discussion chapter on trees in Booch (1987a).
--
generic
  type ITEM is private;
package BINARY_TREE is
  type TREE is private;
  type CHILD is (LEFT, RIGHT);

  procedure CLEAR (THE_TREE    : in out TREE);

  procedure CONSTRUCT(          THE_ITEM              : in     ITEM;
                                AND_THE_TREE          : in out TREE;
                                ON_THE_CHILD          : in     CHILD);

  procedure SWAP_CHILD (        THE_CHILD             : in     CHILD;
                                OF_THE_TREE           : in out TREE;
                                AND_THE_TREE          : in out TREE);
  function IS_NULL (      THE_TREE      : in TREE) return BOOLEAN;

  function ITEM_OF (      THE_TREE      : in TREE) return ITEM;

  function CHILD_OF (     THE_TREE      : in TREE;
                          THE_CHILD     : in CHILD) return TREE;
private
  type NODE;
  type TREE is access NODE;
  type NODE is
    record
      THE_ITEM          :ITEM;
      LEFT_SUBTREE      :TREE;
      RIGHT_SUBTREE             :TREE;
    end record;
end BINARY_TREE;
```

```
--
-- package body BINARY_TREE
--
with UNCHECKED_DEALLOCATION;
package body BINARY_TREE is
 NULL_TREE: constant TREE := null;
 procedure DEALLOCATE is new  UNCHECKED_DEALLOCATION(NODE, TREE);

 procedure CLEAR (THE_TREE : in out TREE) is
 begin
  DEALLOCATE (THE_TREE);
 end CLEAR;

 function IS_NULL (THE_TREE   : in TREE) return BOOLEAN is
 begin
  return (THE_TREE = NULL_TREE);
 end IS_NULL;

 procedure CONSTRUCT (       THE_ITEM              : in     ITEM;
                             AND_THE_TREE          : in out TREE;
                             ON_THE_CHILD          : in     CHILD) is
 begin
  if ON_THE_CHILD = LEFT then
    AND_THE_TREE :=
        new NODE'(      THE_ITEM              => THE_ITEM,
                        LEFT_SUBTREE          => AND_THE_TREE,
                        RIGHT_SUBTREE         => null);
  else
    AND_THE_TREE :=
        new NODE'(      THE_ITEM              => THE_ITEM,
                        LEFT_SUBTREE          => null,
                        RIGHT_SUBTREE         => AND_THE_TREE);
  end if;
 end CONSTRUCT;

 procedure SWAP_CHILD (      THE_CHILD             : in     CHILD;
                             OF_THE_TREE           : in out TREE;
                             AND_THE_TREE          : in out TREE) is
  TEMPORARY_NODE : TREE;
 begin
  if THE_CHILD = LEFT then
    TEMPORARY_NODE := OF_THE_TREE.LEFT_SUBTREE;
    OF_THE_TREE.LEFT_SUBTREE := AND_THE_TREE;
  else
    TEMPORARY_NODE := OF_THE_TREE.RIGHT_SUBTREE;
    OF_THE_TREE.RIGHT_SUBTREE := AND_THE_TREE;
  end if;
  AND_THE_TREE := TEMPORARY_NODE;
 end SWAP_CHILD;
```

```
  function ITEM_OF (THE_TREE : in TREE) return ITEM is
  begin
    return THE_TREE.THE_ITEM;
  end ITEM_OF;

  function CHILD_OF (   THE_TREE      : in TREE;
                        THE_CHILD     : in CHILD) return TREE is
  begin
    if THE_CHILD = LEFT then
      return THE_TREE.LEFT_SUBTREE;
    else
      return THE_TREE.RIGHT_SUBTREE;
    end if;
  end CHILD_OF;

end BINARY_TREE;
```

```
--
--package specification SYMBOL_TREE
--
-- satisfies the need to be able to :-
-- - add symbols to the tree, no duplicates
-- - retrieve any symbol from the tree, according to an equality test on a key field of the symbol
-- - destroy the tree
generic
 type SYMBOL is private;
 type KEY is private;

 -- is_equal & is_less_than allow instantiator to select a key for equality
 -- testing of symbols.
 with function IS_EQUAL    (    LEFT   : SYMBOL;
                                RIGHT  : SYMBOL) return BOOLEAN;
 with function IS_LESS_THAN (   LEFT   : SYMBOL;
                                RIGHT  : SYMBOL) return BOOLEAN;

 with function EQUALS_KEY(      LEFT   : KEY;
                                RIGHT  : SYMBOL) return BOOLEAN;
 with function KEY_LESS_THAN (        LEFT   : KEY;
                                      RIGHT  : SYMBOL) return BOOLEAN;

 with procedure DELETE_SYM ( THE_SYMBOL : in out SYMBOL);
 -- in case the SYMBOL has any dynamic memory elements for the caller to deallocate
 with procedure PROCESS_SYM ( THE_SYMBOL :   SYMBOL);
 --for the traverse routine (for extracting data from the tree)

package SYMBOL_TREE is
 function ADD    ( THE_SYMBOL   :      SYMBOL) return BOOLEAN;
 function DESTROY return BOOLEAN;
 function FOUND(THE_KEY : KEY) return BOOLEAN;
 procedure RETRIEVE (  THE_KEY                :        KEY;
                       IN_THE_SYMBOL          : in out  SYMBOL;
                       SUCCESS                : in out  BOOLEAN);
 procedure TRAVERSAL;
end SYMBOL_TREE;
```

```
--package body SYMBOL_TREE
with BINARY_TREE;
package body SYMBOL_TREE is

--the instantiation of BINARY_TREE...
 package SYMBOL_BIN_TREE is new BINARY_TREE (ITEM => SYMBOL);
 use SYMBOL_BIN_TREE;
 THE_TREE : TREE;

 function FOUND( THE_KEY : KEY; IN_THE_TREE: TREE) return BOOLEAN;

 procedure INSERT(     THE_SYMBOL  :        SYMBOL;
                       IN_THE_TREE : in out TREE;
                       SUCCESS     : in out BOOLEAN);

 procedure RETRIEVE(   THE_KEY             :        KEY;
                       THE_SYMBOL          : in out SYMBOL;
                       FROM_THE_TREE       : in     TREE;
                       SUCCESS             : in out BOOLEAN);

 procedure TRAVERSE ( THIS_TREE : in TREE);

 procedure DESTROY ( THIS_TREE : in out TREE);

 function  ADD     ( THE_SYMBOL :  SYMBOL) return BOOLEAN is
 SUCCESS : BOOLEAN;
 begin
  INSERT( THE_SYMBOL  => THE_SYMBOL,
          IN_THE_TREE => THE_TREE,
          SUCCESS     => SUCCESS);
  return SUCCESS;
 end ADD;

 function FOUND(THE_KEY : KEY) return BOOLEAN is
 begin
  return FOUND( THE_KEY, THE_TREE);
 end FOUND;

 function FOUND( THE_KEY : KEY; IN_THE_TREE: TREE) return BOOLEAN is
 begin
  if not IS_NULL(IN_THE_TREE) then
   if EQUALS_KEY( THE_KEY, ITEM_OF( IN_THE_TREE)) then
        return TRUE;
   elsif KEY_LESS_THAN( THE_KEY, ITEM_OF( IN_THE_TREE))
   then return FOUND( THE_KEY, CHILD_OF(IN_THE_TREE, LEFT));
   else return FOUND( THE_KEY, CHILD_OF(IN_THE_TREE, RIGHT));
   end if;
  end if;
  return FALSE;
 end FOUND;
```

```
procedure INSERT(       THE_SYMBOL  :        SYMBOL;
                        IN_THE_TREE : in out TREE;
                        SUCCESS     : in out BOOLEAN) is
 TEMPORARY_TREE : TREE := IN_THE_TREE;
begin
 if IS_NULL(IN_THE_TREE) then
  CONSTRUCT(            THE_ITEM            => THE_SYMBOL,
                        AND_THE_TREE        => IN_THE_TREE,
                        ON_THE_CHILD        => LEFT);
  SUCCESS := TRUE;
 else
  if IS_EQUAL( LEFT  => THE_SYMBOL,
               RIGHT => ITEM_OF( IN_THE_TREE))
  then -- cannot add to existing record
       SUCCESS := FALSE;
  elsif -- <
  IS_LESS_THAN( LEFT  => THE_SYMBOL,
                RIGHT => ITEM_OF( IN_THE_TREE))
  then
       TEMPORARY_TREE := CHILD_OF(      THE_TREE  => IN_THE_TREE,
                                        THE_CHILD => LEFT);
       INSERT(          THE_SYMBOL  => THE_SYMBOL,
                        IN_THE_TREE => TEMPORARY_TREE,
                        SUCCESS     => SUCCESS);
       if SUCCESS then
        SWAP_CHILD(             THE_CHILD           => LEFT,
                                OF_THE_TREE         => IN_THE_TREE,
                                AND_THE_TREE        => TEMPORARY_TREE);
       end if;
  else -- >
       TEMPORARY_TREE := CHILD_OF(      THE_TREE  => IN_THE_TREE,
                                        THE_CHILD => RIGHT);
       INSERT(          THE_SYMBOL          => THE_SYMBOL,
                        IN_THE_TREE         => TEMPORARY_TREE,
                        SUCCESS             => SUCCESS);
       if SUCCESS then
        SWAP_CHILD(             THE_CHILD           => RIGHT,
                                OF_THE_TREE         => IN_THE_TREE,
                                AND_THE_TREE        => TEMPORARY_TREE);
       end if;
  end if;
 end if;
end INSERT;

procedure RETRIEVE ( THE_KEY        :        KEY;
                     IN_THE_SYMBOL  : in out SYMBOL;
                     SUCCESS        : in out BOOLEAN) is
begin
 RETRIEVE(  THE_KEY             => THE_KEY,
            THE_SYMBOL          => IN_THE_SYMBOL,
            FROM_THE_TREE       => THE_TREE,
            SUCCESS             => SUCCESS);
end RETRIEVE;
```

```
procedure RETRIEVE(  THE_KEY              :        KEY;
                     THE_SYMBOL           : in out SYMBOL;
                     FROM_THE_TREE        : in     TREE;
                     SUCCESS              : in out BOOLEAN) is
begin
 if not IS_NULL(FROM_THE_TREE) then
  if EQUALS_KEY(     LEFT   => THE_KEY,
                     RIGHT  => ITEM_OF( FROM_THE_TREE))
  then
       SUCCESS := TRUE;
       THE_SYMBOL := ITEM_OF( FROM_THE_TREE);
  elsif -- <
  KEY_LESS_THAN(  LEFT   => THE_KEY,
                     RIGHT  => ITEM_OF( FROM_THE_TREE))
  then
       RETRIEVE(     THE_KEY              => THE_KEY,
                     THE_SYMBOL           => THE_SYMBOL,
                     FROM_THE_TREE
                           => CHILD_OF(  THE_TREE  => FROM_THE_TREE,
                                         THE_CHILD => LEFT),
                     SUCCESS              => SUCCESS);
  else -- >
       RETRIEVE(     THE_KEY              => THE_KEY,
                     THE_SYMBOL           => THE_SYMBOL,
                     FROM_THE_TREE
                           => CHILD_OF(  THE_TREE  => FROM_THE_TREE,
                                         THE_CHILD => RIGHT),
                     SUCCESS              => SUCCESS);
  end if;
 end if;
end RETRIEVE;

procedure DESTROY ( THIS_TREE : in out TREE) is
TEMPORARY_SYMBOL : SYMBOL;
TEMPORARY_TREE  : TREE;
begin
 if not IS_NULL( THIS_TREE) then
  TEMPORARY_SYMBOL := ITEM_OF( THIS_TREE);
  TEMPORARY_TREE := CHILD_OF(  THE_TREE     => THIS_TREE,
                               THE_CHILD    => LEFT);
  DESTROY( TEMPORARY_TREE );

  TEMPORARY_TREE := CHILD_OF(  THE_TREE     => THIS_TREE,
                               THE_CHILD    => RIGHT);
  DESTROY( TEMPORARY_TREE );

  DELETE_SYM( THE_SYMBOL => TEMPORARY_SYMBOL);

  CLEAR (THE_TREE => THIS_TREE);
 end if;
end DESTROY;
```

```
  function DESTROY return BOOLEAN is
  begin
    DESTROY ( THIS_TREE => THE_TREE);
    return TRUE;
  exception
    when others => return FALSE;
  end DESTROY;

  procedure TRAVERSE ( THIS_TREE : in TREE) is
  begin
    if not IS_NULL ( THIS_TREE) then
      TRAVERSE  ( THIS_TREE   => CHILD_OF(  THE_TREE  => THIS_TREE,
                                            THE_CHILD => LEFT));
      PROCESS_SYM( THE_SYMBOL  => ITEM_OF( THIS_TREE));
      TRAVERSE  ( THIS_TREE   => CHILD_OF(  THE_TREE  => THIS_TREE,
                                            THE_CHILD => RIGHT));
    end if;
  end TRAVERSE;

  procedure TRAVERSAL is
  begin
    TRAVERSE ( THIS_TREE => THE_TREE);
  end TRAVERSAL;

end SYMBOL_TREE;
```

Taken from the package PROCESS, the following five Ada statements are all that is necessary to create the required five instances of SYMBOL_TREE. Those subprogram names passed as parameters to the package instantiation (e.g. DELETE, EXTRACT), are visible from within package PROCESS.

```
package GLOBAL_CONST_TREE
        is new SYMBOL_TREE( SYMBOL          => VAR_PTR,
                            KEY             => STRING,
                            IS_EQUAL        => IS_EQUAL,
                            IS_LESS_THAN    => IS_LESS_THAN,
                            EQUALS_KEY      => IS_EQUAL,
                            KEY_LESS_THAN   => IS_LESS_THAN,
                            PROCESS_SYM     => EXTRACT,
                            DELETE_SYM      => DELETE );

 package GLOBAL_VAR_TREE
        is new SYMBOL_TREE( SYMBOL          => VAR_PTR,
                            KEY             => STRING,
                            IS_EQUAL        => IS_EQUAL,
                            IS_LESS_THAN    => IS_LESS_THAN,
                            EQUALS_KEY      => IS_EQUAL,
                            KEY_LESS_THAN   => IS_LESS_THAN,
                            PROCESS_SYM     => EXTRACT,
                            DELETE_SYM      => DELETE );

--contents used as replacements within in the sub-prog scope
package LOCAL_CONST_TREE
        is new SYMBOL_TREE( SYMBOL          => VAR_PTR,
                            KEY             => STRING,
                            IS_EQUAL        => IS_EQUAL,
                            IS_LESS_THAN    => IS_LESS_THAN,
                            EQUALS_KEY      => IS_EQUAL,
                            KEY_LESS_THAN   => IS_LESS_THAN,
                            PROCESS_SYM     => EXTRACT,
                            DELETE_SYM      => DELETE );

 package LOCAL_VAR_TREE
        is new SYMBOL_TREE( SYMBOL          => VAR_PTR,
                            KEY             => STRING,
                            IS_EQUAL        => IS_EQUAL,
                            IS_LESS_THAN    => IS_LESS_THAN,
                            EQUALS_KEY      => IS_EQUAL,
                            KEY_LESS_THAN   => IS_LESS_THAN,
                            PROCESS_SYM     => EXTRACT,
                            DELETE_SYM      => DELETE );

 package SUB_PROG_TREE
        is new SYMBOL_TREE( SYMBOL          => SUB_PROG_PTR,
                            KEY             => STRING,
                            IS_EQUAL        => IS_EQUAL,
                            IS_LESS_THAN    => IS_LESS_THAN,
                            EQUALS_KEY      => IS_EQUAL,
                            KEY_LESS_THAN   => IS_LESS_THAN,
                            PROCESS_SYM     => EXTRACT,
                            DELETE_SYM      => DELETE );
```

Appendix E: The use of the C pre-processor to achieve inter-compiler compatibility.

```
/*
 FILE NAME    :        "standard.c"

 DESCRIPTION  :
                       Not all C compilers are the same, particularly where
                       low-level machine specific features are concerned.
                       Fortunately, there exists with C compilers a
                       pre-processor which interprets macros and replaces
                       defined statements.

                       The MS-DOS C compilers supplied by Microsoft and
                       Borland provide unique identification to these
                       pre-processors, which can then be used to advantage
                       in reconciling differences between the different C
                       dialects.

 AUTHOR       :        Michael Collins.
*/

/*
 Include file <dos.h> contains the Borland C MK_FP, inp & outp definition.
 The fact that it also contains definitions of O/S dependant functions
 is irrelevent if we never use them. The translator sees to it
 that we never use O/S dependant functions.
*/
#include <dos.h>

typedef          int        BOOLEAN ;        /*in C BOOLEANs are normally
                                               contained in integers*/

typedef unsigned char    BYTE ;                    /* 8-bit data */
typedef unsigned int     WORD ;                    /* 16-bit data */
typedef unsigned long    DWORD ;                   /* 32-bit data */
typedef double           QWORD ;                   /* 64-bit data */

typedef void far *       PTR ;                     /* Pointer to any data type */
typedef BYTE far *             BYTE_PTR ;          /* Pointer to 8-bit data */
typedef WORD far *             WORD_PTR ;          /* Pointer to 16-bit data */
typedef DWORD far *      DWORD_PTR ;               /* Pointer to 32-bit data */
typedef QWORD far *      QWORD_PTR ;               /* Pointer to 64-bit data */

//All C source from the translator is in lower case
#define word WORD
#define byte BYTE
#define FALSE 0
#define TRUE !FALSE
#define true TRUE
#define false FALSE
#define EXIT goto  /* EXITs from Loops are replaced by goto*/
```

```
#if       defined(__TURBOC__) /*Borland's Compiler identifier*/
/*
 Set_Vector macro places an interrupt service routine address
 into its hardware vector
*/
#define  Set_Vector(location, addr) *(void interrupt far (far * far *)())((location)) = (addr)

#else     /* Must be Microsoft C */
/*MK_FP mimics Borland's function of that name to create a far (20 bit)
 pointer from a segment and offset*/
#define  MK_FP(seg,ofs)  ((void far *) (((unsigned long)(seg) << 16) | (unsigned)(ofs)))

/* Set_Vector a la Microsoft...*/
#define  Set_Vector(location, addr) *((DWORD_PTR) ((location))) = ((DWORD) (addr))

/*Now supply some really low level alignment for Microsoft C*/
#define asm              _asm
#define  enable()        _enable()
#define  disable()       _disable()
#define emit             _emit
#define asm              _asm
#endif

/*
 The Ada-like to "Safe C" translator outputs i_o_var get & put
 but both C compilers understand inp and outp respectively.
*/
#define i_o_var_get inp
#define i_o_var_put outp

/*
  Aligning memory mapped variables mem_var put & get
  is a little more complicated. Note the use of MK_FP
  for which we defined a macro above. Macros can reside
  within macros, which makes the pre-processor a really
  powerful tool.
*/
#define mem_var_get(a,b)  \\
  (*((byte far *)MK_FP((a),(b))))
#define mem_var_put( seg, ofs, valu) \\
  (*((byte far *)MK_FP((seg),(ofs))))=(valu)
```

```
/*
  Rapid processor interrupt control, via in-line
  assembly language instructions.
*/
#define enable_int    asm sti
#define disable_int   asm cli

/*
  The V25 needs a special command to complete interrupts
  which have been initiated by internal on-chip register events.
  We can substitute some in-line assembly code to achieve this.
*/
#define FINT asm {asm emit 0x0f asm emit 0x92}
        /* signal end of INTERNAL V25 interrupt */
```

Appendix F: The test program, written in the Ada-like language developed in this project, and the translation in "Safe C".

F.1 The Test program: TEST_PROG.ADL

```
-- PROGRAM NAME     : TEST_PROG.ADL
-- WRITTEN IN       : AN ADA-LIKE LANGUAGE
-- AUTHOR           : MIKE COLLINS
-- DESCRIPTION      : Performs protocol conversion between the
--                  : RS232 serial link
--                  :  @ 9600bps, 8 data, no parity, 1 stop
--                  : and the
--                  : RS422 serial link
--                  :  @ 4800bps, 8 data, no parity, 1 stop
--                  : and also emits an updated "second" count
--                  : once per second from the RS232 port.
--
-- NOTES            : All machine specific names used in
--                  : this program are taken from the V25
--                  : literature. Specifically this is
--                  : "uPD70320/322 (V25tm) 16-bit, Single-Chip
--                  : CMOS Microcomputers" and is available from
--                  : NEC Electronics Inc. or
--                  : George Brown Group,
--                  : 294 South Road, Hilton S.A. 5033, Australia.
--
-- Global constants used in the program
--
SERIAL_PORT_0 : constant INTEGER := 0;
SERIAL_PORT_1 : constant INTEGER := 1;

-- Global data-stores, used for inter-process communication
THE_SECONDS_STORE,
RECEIVED_CHAR_0,
RECEIVED_CHAR_1,
ERROR_COND_0,
ERROR_COND_1     : byte;

-- Most Significant Digit (MSD) of current Real-Time clock chip second
SECONDS_MSD,
-- and the Least Significant Digit (LSD) of the current second
SECONDS_LSD      : CHARACTER;

NEW_SECOND,
CHAR_WAITING_AT_0,
CHAR_WAITING_AT_1 : boolean;


-- I/O mapped register...
-- The real-time clock seconds (as opposed to hours, minutes &etc)
-- register is located at I/O map address 2
REAL_TIME_SECONDS_PORT : BYTE;
for REAL_TIME_SECONDS_PORT map_at 2;
```

```
--some memory mapped I/O w.r.t. On chip Serial Port #0

BRG0,          --Baud Rate Generator 0
SCC0,          --Serial Communication Control Register 0
SCE0,          --Serial Communication Error Register 0
SCM0,          --Serial Comm. Mode Register 0
RXB0,          --Receive character Buffer 0
TXB0,          --Transmit character Buffer 0
SRIC0,         --Serial Receive Interrupt Control 0
SEIC0,         --Serial Error Interrupt Control 0
STIC0          --Serial Transmit Interrupt Control 0
   : byte;
for BRG0       use_at 1015658;
for SCC0       use_at 1015657;
for SCE0       use_at 1015659;
for SCM0       use_at 1015656;
for RXB0       use_at 1015648;
for TXB0       use_at 1015650;
for SRIC0      use_at 1015661;
for SEIC0      use_at 1015660;
for STIC0      use_at 1015662;

--Now all the registers for On chip Serial Port 1
BRG1,          --Baud Rate Generator 1
SCC1,          --Serial Communication Control Register 1
SCE1,          --Serial Communication Error Register 1
SCM1,          --Serial Comm. Mode Register 1
RXB1,          --Receive character Buffer 1
TXB1,          --Transmit character Buffer 1
SRIC1,         --Serial Receive Interrupt Control 1
SEIC1,         --Serial Error Interrupt Control 1
STIC1          --Serial Transmit Interrupt Control 1
   : byte;
for BRG1       use_at 1015674;
for SCC1       use_at 1015673;
for SCE1       use_at 1015675;
for SCM1       use_at 1015672;
for RXB1       use_at 1015664;
for TXB1       use_at 1015666;
for SRIC1      use_at 1015677;
for SEIC1      use_at 1015676;
for STIC1      use_at 1015678;
```

```
PRC,            -- Processor Control Register 1eb
TBIC,           -- Time Base Interrupt Control Register
PM0,            -- Port 0 Mode Port
PMC0,           -- Port 0 Mode Control Port
P0,             -- Port 0 data port
PM1,            -- Port 1 Mode Port
PMC1,           -- Port 1 Mode Control Port
P1,             -- Port 1 data port
PM2,            -- Port 2 Mode Port
PMC2,           -- Port 2 Mode Control Port
P2              -- Port 2 data port
    : byte;

for PRC         use_at  1015787;
for TBIC        use_at  1015788;
for PM0         use_at  1015553;
for PMC0        use_at  1015554;
for P0          use_at  1015552;
for PM1         use_at  1015561;
for PMC1        use_at  1015562;
for P1          use_at  1015560;
for PM2         use_at  1015569;
for PMC2        use_at  1015570;
for P2          use_at  1015568;


driver test_prog; -- this is the main sub-program
          -- only one is allowed per program

function CHAR_LENGTH_CONVERT( THE_LENGTH   : integer  ) return byte;
procedure INIT_REGISTERS;
function PARITY_CONVERT    ( THE_PARITY   : CHARACTER) return byte;
procedure PUT_CHAR(    TO_THE_PORT : INTEGER;
                       THE_CHAR    : CHARACTER);

function SET_BRG( THE_BAUD_RATE : INTEGER) return BYTE ;
procedure SET_DTR( THE_SERIAL_PORT: INTEGER);
function SET_SCC( THE_BAUD_RATE : INTEGER) return BYTE;
procedure SET_SERIAL_PORT (  THE_PORT                 : INTEGER;
                             THE_BAUD_RATE            : INTEGER;
                             THE_PARITY               : CHARACTER;
                             BITS_PER_CHARACTER       : INTEGER;
                             THE_STOP_BITS            : INTEGER );

function STOP_BITS_CONVERT  ( THE_STOP_BITS: integer  ) return byte;

--announce some interrupt service routines:

task SERIAL_PORT_0_ISR;              --for on-chip serial port 0
for  SERIAL_PORT_0_ISR  use_at 52;   --located at memory address 52

task SERIAL_PORT_1_ISR;              --for on-chip serial port 1
for  SERIAL_PORT_1_ISR  use_at 68;   --located at memory address 68

task CONSUMER;                       -- a co-routine task
task PRODUCER;                       -- a second co-routine task
```

```
--
--Subprogram bodies...
--
--a parameter-less procedure
procedure INIT_REGISTERS is
begin
 PRC := 68;

 PM0  := 125; -- all input except /DTR0 and      Clock-out
 PMC0 := 128; -- Bit 7 to clock-out, others to port mode
 P0   := 2;   -- set /DTR0 inactive

 PM1  := 15;  -- Port1 upper 4 bits output
 PMC1 := 0;   -- lower 3 bits must always be input
 P1   := 0;   -- leaving  Port1.4 .. Port1.7 for programmable i/o

 PM2  := 0;   --
 PMC2 := 0;
 P2   := 0;
end INIT_REGISTERS;

-- this function demonstrates a simple case statement, unconditional returns
-- and recursion.
function SET_BRG( THE_BAUD_RATE : INTEGER) return BYTE is
DEFAULT_BAUD : constant INTEGER := 9600;
begin
  case THE_BAUD_RATE is
    when 110   => return 142;
    when 150   => return 208;
    when 300   => return 208;
    when 600   => return 208;
    when 1200  => return 208;
    when 2400  => return 208;
    when 4800  => return 208;
    when 9600  => return 208;
    when 19200 => return 208;
    when others => return SET_BRG( DEFAULT_BAUD); --default to 9600 baud
  end_case;
end SET_BRG;
```

```
--this function demonstrates a simple case statement, variable allocation,
-- recursion and a simple function return statement.
function SET_SCC( THE_BAUD_RATE : INTEGER) return BYTE is
--DEFAULT_BAUD : constant INTEGER := 9600;
RETURN_VAR   : BYTE;
begin
 case THE_BAUD_RATE is
  when 110   => RETURN_VAR := 8;
  when 150   => RETURN_VAR := 7;
  when 300   => RETURN_VAR := 6;
  when 600   => RETURN_VAR := 5;
  when 1200  => RETURN_VAR := 4;
  when 2400  => RETURN_VAR := 3;
  when 4800  => RETURN_VAR := 2;
  when 9600  => RETURN_VAR := 1;
  when 19200 => RETURN_VAR := 0;
  when others => RETURN_VAR := SET_SCC( 9600);
 end_case;

 return RETURN_VAR;
end SET_SCC;

function CHAR_LENGTH_CONVERT( THE_LENGTH: integer)  return byte is
begin
 case THE_LENGTH is
  when 7     => return 0;
  when 8     => return 8;
  when others => return 8; --default to 8 bits
 end_case;
end CHAR_LENGTH_CONVERT;

function PARITY_CONVERT( THE_PARITY : CHARACTER) return byte is
RETURN_VAL : byte;
begin
 if (THE_PARITY = 'o')  then RETURN_VAL := 32; end_if;
 if (THE_PARITY = 'e')  then RETURN_VAL := 48; end_if;
 if (THE_PARITY = 'n')  then RETURN_VAL := 0;  end_if;
 return RETURN_VAL;
end PARITY_CONVERT;
```

```
--this function demonstrates the use of deterministic
-- control structures, when associated with machine hardware
-- which can change state asynchronously w.r.t. normal
-- program flow.
procedure PUT_CHAR(   TO_THE_PORT : INTEGER;
                      THE_CHAR    : CHARACTER) is
LOOP_COUNTER : INTEGER;
begin
 if (TO_THE_PORT = 0) then
  for  LOOP_COUNTER in 1 .. 10000 loop
   if (STIC0 >= 128)   --test if we can send the char
   then
    STIC0 := STIC0 and 127;  -- remove buffer empty flag
    TXB0  := BYTE( THE_CHAR);        -- demonstrate an explicit type-cast
                                     -- while putting the char into the
                                     -- transmission register
    exit;                            -- then get out of the loop
   end_if;
  end_loop;
 else
  for  LOOP_COUNTER in 1 .. 10000  loop --as above, but for Serial port 1
   if (STIC1 >= 128)  then           -- can we send the char?
    STIC1 := STIC1 and 127;
    TXB1  := BYTE( THE_CHAR);
    exit;
   end_if;
  end_loop;

 end_if;

end PUT_CHAR;


function STOP_BITS_CONVERT ( THE_STOP_BITS : integer) return byte is
begin
 if (THE_STOP_BITS = 2) then return 4;
 else return 0; end_if;
end STOP_BITS_CONVERT;

procedure SET_DTR( THE_SERIAL_PORT: INTEGER) is
DTR0_ON : constant BYTE := 253; --and this with P0 to set DTR0 active
DTR1_ON : constant BYTE := 16;  --or  this with P1 to set DTR1 active
begin
 case THE_SERIAL_PORT is
  when 0 => P0 := P0 and DTR0_ON;
  when 1 => P1 := P1 or  DTR1_ON;
  when others => null;
 end_case;
end SET_DTR;
```

```
--All this just to set up the port to send/receive a character!!
procedure SET_SERIAL_PORT (  THE_PORT         : INTEGER;
                             THE_BAUD_RATE    : INTEGER;
                             THE_PARITY       : CHARACTER;
                             BITS_PER_CHARACTER : INTEGER;
                             THE_STOP_BITS    : INTEGER ) is
ASYNCH           : constant BYTE := 1;
TX_READY         : constant BYTE := 128;
RX_ENABLE        : constant BYTE := 64;
ERROR_INT_DISABLE_MASK : constant BYTE := 71; --disable error Interrupts
TX_ENABLE        : constant BYTE := 64; -- enable Tx generally


BIT_PARAMS : BYTE;
begin
 BIT_PARAMS := ASYNCH;
 BIT_PARAMS := BIT_PARAMS or TX_READY;
 BIT_PARAMS := BIT_PARAMS or RX_ENABLE;
 BIT_PARAMS := BIT_PARAMS or PARITY_CONVERT( THE_PARITY);
 BIT_PARAMS := BIT_PARAMS or CHAR_LENGTH_CONVERT( BITS_PER_CHARACTER);
 BIT_PARAMS := BIT_PARAMS or STOP_BITS_CONVERT(THE_STOP_BITS);

 case THE_PORT is
  when 0 =>
   BRG0  := SET_BRG (THE_BAUD_RATE);
   SCC0  := SET_SCC (THE_BAUD_RATE);
   SCM0  := BIT_PARAMS;
   SEIC0 := SEIC0 and ERROR_INT_DISABLE_MASK;
   SEIC0 := SEIC0 or 64;
   SRIC0 := SRIC0 and 7;
   STIC0 := STIC0 and 199;
   STIC0 := STIC0 or 64;
  when 1 =>
   BRG1  := SET_BRG (THE_BAUD_RATE);
   SCC1  := SET_SCC (THE_BAUD_RATE);
   SCM1  := BIT_PARAMS;
   SEIC1 := SEIC1 and ERROR_INT_DISABLE_MASK;
   SEIC1 := SEIC1 or 64;
   SRIC1 := SRIC1 and 7;
   STIC1 := STIC1 and 199;
   STIC1 := STIC1 or 64;
  when others => null;
 end_case;

end SET_SERIAL_PORT;
```

```
-- and this to receive a character
task SERIAL_PORT_0_ISR is
begin
 ERROR_COND_0     := SCE0;
 RECEIVED_CHAR_0  := RXB0;
 CHAR_WAITING_AT_0 := TRUE;

 SRIC0 := SRIC0 and 127; -- clear this interrupt
 SEIC0 := SEIC0 and 127; -- and any pending error interrupt

end SERIAL_PORT_0_ISR;


task SERIAL_PORT_1_ISR is
begin
 ERROR_COND_1     := SCE1;
 RECEIVED_CHAR_1  := RXB1;
 CHAR_WAITING_AT_1 := TRUE;

 SRIC1 := SRIC1 and 127; -- clear this interrupt
 SEIC1 := SEIC1 and 127; -- and any pending error interrupt

end SERIAL_PORT_1_ISR;
```

```
-- producer, in its sequence, polls the real-time clock "seconds" register
-- if there is a difference from the last reading it collects the
-- new value and prepares it for the producer to use.
--
task PRODUCER is
TEMPORARY_BYTE: BYTE;

ASCII_ORDINAL_0 : constant BYTE := 48;  --'0' position in the ASCII table

UPPER_NYBBLE : constant BYTE := 240;
LOWER_NYBBLE : constant BYTE := 15;

begin
 if ( REAL_TIME_SECONDS_PORT /= THE_SECONDS_STORE ) then
   -- First log the new value
   THE_SECONDS_STORE := REAL_TIME_SECONDS_PORT;

   -- We need to split up the 10's and Units columns of the seconds
   -- First the 10's or Most Significant Digit...
   -- isolate upper 4 bits, in which we will find the '10's column
   TEMPORARY_BYTE := THE_SECONDS_STORE and UPPER_NYBBLE;
   -- move them into the lower 4 bit area
   TEMPORARY_BYTE := TEMPORARY_BYTE / 16;
   -- make this into the byte representation of an ASCII character
   TEMPORARY_BYTE := TEMPORARY_BYTE + ASCII_ORDINAL_0;
   -- and save this Most Significant Digit
   -- in readiness for the Consumer process to send it
   SECONDS_MSD := CHARACTER( TEMPORARY_BYTE);


   -- now isolate the 'units' component of the "seconds"
   TEMPORARY_BYTE := THE_SECONDS_STORE and LOWER_NYBBLE;
   -- make this into the byte representation of an ASCII character
   TEMPORARY_BYTE := TEMPORARY_BYTE + ASCII_ORDINAL_0;
   -- and save this Least Significant Digit
   -- in readiness for the Consumer process
   SECONDS_LSD := CHARACTER( TEMPORARY_BYTE);

   NEW_SECOND       := TRUE;
 end_if;

end PRODUCER;
```

```
-- The CONSUMER task, in reality has three jobs to do in life:
--  1) If a char has been received by RS232 (Serial port 0) ISR, then
--     it sends that char out of the RS422 port.
--  2) Similarly, if a char arrives at the RS422 (Serial port 1) ISR,
--     that char is sent to the RS232 port
--  3) If a New second has arrived (and this occurs once per sec),
--     then the second's value [00..59] is sent to the RS232 serial
--     port.

task CONSUMER is
TEMPORARY_CHAR : CHARACTER;

begin
  -- this is job number 1.
  if( CHAR_WAITING_AT_0 ) then
    TEMPORARY_CHAR :=CHARACTER(RECEIVED_CHAR_0);
    CHAR_WAITING_AT_0 := FALSE;
    PUT_CHAR( SERIAL_PORT_1, TEMPORARY_CHAR);
  end_if;
  SYNCH;

  -- this is job number 2.
  if( CHAR_WAITING_AT_1 ) then
    TEMPORARY_CHAR :=CHARACTER(RECEIVED_CHAR_1);
    CHAR_WAITING_AT_1 := FALSE;
    PUT_CHAR( SERIAL_PORT_0, TEMPORARY_CHAR);
  end_if;
  SYNCH;

  -- this is job number 3.
  if (NEW_SECOND) then
    --Transmit the seconds, most significant digit first
    PUT_CHAR( SERIAL_PORT_0, SECONDS_MSD);
    PUT_CHAR( SERIAL_PORT_0, SECONDS_LSD);

    TEMPORARY_CHAR := CHARACTER(13);
    PUT_CHAR( SERIAL_PORT_0, TEMPORARY_CHAR); -- send a carriage return
    TEMPORARY_CHAR := CHARACTER(10);
    PUT_CHAR( SERIAL_PORT_0, TEMPORARY_CHAR); -- and line feed .

    NEW_SECOND := FALSE;
  end_if;
  SYNCH;

end CONSUMER;
```

```
driver test_prog is

the_letter : character;
letter_num,
loop_counter : integer;
begin
 --initialise inter-process data
 NEW_SECOND       := FALSE;
 CHAR_WAITING_AT_0 := FALSE;
 CHAR_WAITING_AT_1 := FALSE;

 -- disable maskable interrupts, as we don't want any occurring
 -- whilst we are setting the V25 peripheral ports ready for action.
 disable_int;

 --set general purpose output registers
 init_registers;

 -- set up serial-port 0 (the RS-232 port)
 set_serial_port(SERIAL_PORT_0 , 9600, 'n', 8, 1);

 -- now set up serial port 1 (the RS422 port)
 set_serial_port(SERIAL_PORT_1 , 4800, 'n', 8, 1);

 -- now that the ports can cope with interrupts, we may safely
 -- enable the interrupts.
 enable_int;

 -- set the serial port handshaking lines to their active state,
 -- so that characters may be transmitted and received.
 SET_DTR(SERIAL_PORT_0);
 SET_DTR(SERIAL_PORT_1);

 -- now a little wake-up message
 letter_num := 1;
 while (true) loop
  case letter_num is
   when 1=> the_letter := 'H';
   when 2=> the_letter := 'e';
   when 3=> the_letter := 'l';
   when 4=> the_letter := 'l';
   when 5=> the_letter := 'o';
   when others => null;
  end_case;
  put_char( SERIAL_PORT_0, the_letter);
  put_char( SERIAL_PORT_1, the_letter);

  loop_counter := loop_counter+1;
  exit when (loop_counter > 5);
 end_loop;

 while (TRUE) loop
  PRODUCER;
  consumer;
 end_loop;

end test_prog;
```

F.2 The Translated "Safe C" for the test program:

```
#include "standard.c"
/********** Global constants *********/
#define serial_port_0 0
#define serial_port_1 1
/********** Global Variables *********/
volatile unsigned char brg0;
volatile unsigned char brg1;
volatile BOOLEAN char_waiting_at_0;
volatile BOOLEAN char_waiting_at_1;
volatile unsigned char error_cond_0;
volatile unsigned char error_cond_1;
volatile BOOLEAN new_second;
volatile unsigned char p0;
volatile unsigned char p1;
volatile unsigned char p2;
volatile unsigned char pm0;
volatile unsigned char pm1;
volatile unsigned char pm2;
volatile unsigned char pmc0;
volatile unsigned char pmc1;
volatile unsigned char pmc2;
volatile unsigned char prc;
volatile unsigned char real_time_seconds_port;
volatile unsigned char received_char_0;
volatile unsigned char received_char_1;
volatile unsigned char rxb0;
volatile unsigned char rxb1;
volatile unsigned char scc0;
volatile unsigned char scc1;
volatile unsigned char sce0;
volatile unsigned char sce1;
volatile unsigned char scm0;
volatile unsigned char scm1;
volatile char seconds_lsd;
volatile char seconds_msd;
volatile unsigned char seic0;
volatile unsigned char seic1;
volatile unsigned char sric0;
volatile unsigned char sric1;
volatile unsigned char stic0;
volatile unsigned char stic1;
volatile unsigned char tbic;
volatile unsigned char the_seconds_store;
volatile unsigned char txb0;
volatile unsigned char txb1;
```

```
/********** Sub program prototypes ****/
unsigned char char_length_convert(int the_length);
void consumer(void);
void init_registers(void);
unsigned char parity_convert(char the_parity);
void producer(void);
void prt_char(int to_the_port,char the_char);
void interrupt serial_port_0_isr(void);
void interrupt serial_port_1_isr(void);
unsigned char set_brg(int the_baud_rate);
void set_dtr(int the_serial_port);
unsigned char set_scc(int the_baud_rate);
void set_serial_port(int the_port,int the_baud_rate,char the_parity,int bits_per_character,int
the_stop_bits);
unsigned char stop_bits_convert(int the_stop_bits);
void test_prog(void);
/********** Sub programs **************/
void init_registers(void){

 mem_var_put( 61440, 32747, 68);
 mem_var_put( 61440, 32513, 125);
 mem_var_put( 61440, 32514, 128);
 mem_var_put( 61440, 32512, 2);
 mem_var_put( 61440, 32521, 15);
 mem_var_put( 61440, 32522, 0);
 mem_var_put( 61440, 32520, 0);
 mem_var_put( 61440, 32529, 0);
 mem_var_put( 61440, 32530, 0);
 mem_var_put( 61440, 32528, 0);
}/* end of init_registers*/
```

```
unsigned char set_brg(int the_baud_rate){

 switch( the_baud_rate ){
 case 110:  return 142;
 break;
 case 150:  return 208;
 break;
 case 300:  return 208;
 break;
 case 600:  return 208;
 break;
 case 1200:  return 208;
 break;
 case 2400:  return 208;
 break;
 case 4800:  return 208;
 break;
 case 9600:  return 208;
 break;
 case 19200:  return 208;
 break;
 default :  return  set_brg(9600);
 break;
 } /*switch*/
}/* end of set_brg*/

unsigned char set_scc(int the_baud_rate){

unsigned char return_var;
 switch( the_baud_rate ){
 case 110:  return_var= 8;
 break;
 case 150:  return_var= 7;
 break;
 case 300:  return_var= 6;
 break;
 case 600:  return_var= 5;
 break;
 case 1200:  return_var= 4;
 break;
 case 2400:  return_var= 3;
 break;
 case 4800:  return_var= 2;
 break;
 case 9600:  return_var= 1;
 break;
 case 19200:  return_var= 0;
 break;
 default :  return_var=  set_scc(9600);
 break;
 } /*switch*/
 return return_var;
}/* end of set_scc*/
```

```
unsigned char char_length_convert(int the_length){

 switch( the_length ){
 case 7:  return 0;
 break;
 case 8:  return 8;
 break;
 default :  return 8;
 break;
 } /*switch*/
}/* end of char_length_convert*/

unsigned char parity_convert(char the_parity){

unsigned char return_val;
 if( the_parity == 'o' ){
 return_val= 32;
 }/*end if*/
 if( the_parity == 'e' ){
 return_val= 48;
 }/*end if*/
 if( the_parity == 'n' ){
 return_val= 0;
 }/*end if*/
 return return_val;
}/* end of parity_convert*/

void put_char(int to_the_port,char the_char){

int loop_counter;
 if( to_the_port == 0 ){
 for( loop_counter=1; loop_counter<=10000; loop_counter++){/*(Loop Label=L1:)*/
 if(  mem_var_get( 61440, 32622) >= 128 ){
 mem_var_put( 61440, 32622, mem_var_get( 61440, 32622) &127);
 mem_var_put( 61440, 32610, (BYTE)the_char/*Explicit Type Cast*/);
 EXIT L1;
 }/*end if*/
 } L1: ;
 } else {/*else part*/
 for( loop_counter=1; loop_counter<=10000; loop_counter++){/*(Loop Label=L2:)*/
 if(  mem_var_get( 61440, 32638) >= 128 ){
 mem_var_put( 61440, 32638, mem_var_get( 61440, 32638) &127);
 mem_var_put( 61440, 32626, (BYTE)the_char/*Explicit Type Cast*/);
 EXIT L2;
 }/*end if*/
 } L2: ;
 }/*end if*/
}/* end of put_char*/
```

```
unsigned char stop_bits_convert(int the_stop_bits){

 if( the_stop_bits == 2 ){
 return 4;
 } else {/*else part*/
 return 0;
 }/*end if*/
}/* end of stop_bits_convert*/

void set_dtr(int the_serial_port){

 switch( the_serial_port ){
 case 0: mem_var_put( 61440, 32512, mem_var_get( 61440, 32512) &253);
 break;
 case 1: mem_var_put( 61440, 32520, mem_var_get( 61440, 32520) | 16);
 break;
 default : ; /* null statement */
 break;
 } /*switch*/
}/* end of set_dtr*/

void set_serial_port(int the_port,int the_baud_rate,char the_parity,int bits_per_character,int
the_stop_bits){

unsigned char bit_params;
 bit_params= 1;
 bit_params= bit_params | 128;
 bit_params= bit_params | 64;
 bit_params= bit_params | parity_convert(the_parity);
 bit_params= bit_params | char_length_convert(bits_per_character);
 bit_params= bit_params | stop_bits_convert(the_stop_bits);
 switch( the_port ){
 case 0: mem_var_put( 61440, 32618, set_brg(the_baud_rate));
 mem_var_put( 61440, 32617, set_scc(the_baud_rate));
 mem_var_put( 61440, 32616, bit_params);
 mem_var_put( 61440, 32620, mem_var_get( 61440, 32620) &71);
 mem_var_put( 61440, 32620, mem_var_get( 61440, 32620) | 64);
 mem_var_put( 61440, 32621, mem_var_get( 61440, 32621) &7);
 mem_var_put( 61440, 32622, mem_var_get( 61440, 32622) &199);
 mem_var_put( 61440, 32622, mem_var_get( 61440, 32622) | 64);
 break;
 case 1: mem_var_put( 61440, 32634, set_brg(the_baud_rate));
 mem_var_put( 61440, 32633, set_scc(the_baud_rate));
 mem_var_put( 61440, 32632, bit_params);
 mem_var_put( 61440, 32636, mem_var_get( 61440, 32636) &71);
 mem_var_put( 61440, 32636, mem_var_get( 61440, 32636) | 64);
 mem_var_put( 61440, 32637, mem_var_get( 61440, 32637) &7);
 mem_var_put( 61440, 32638, mem_var_get( 61440, 32638) &199);
 mem_var_put( 61440, 32638, mem_var_get( 61440, 32638) | 64);
 break;
 default : ; /* null statement */
 break;
 } /*switch*/
}/* end of set_serial_port*/
```

```
void interrupt serial_port_0_isr(void){

 error_cond_0= mem_var_get( 61440, 32619) ;
 received_char_0= mem_var_get( 61440, 32608) ;
 char_waiting_at_0= true;
 mem_var_put( 61440, 32621, mem_var_get( 61440, 32621) &127);
 mem_var_put( 61440, 32620, mem_var_get( 61440, 32620) &127);

 FINT;
}/* end of serial_port_0_isr*/

void interrupt serial_port_1_isr(void){

 error_cond_1= mem_var_get( 61440, 32635) ;
 received_char_1= mem_var_get( 61440, 32624) ;
 char_waiting_at_1= true;
 mem_var_put( 61440, 32637, mem_var_get( 61440, 32637) &127);
 mem_var_put( 61440, 32636, mem_var_get( 61440, 32636) &127);

 FINT;
}/* end of serial_port_1_isr*/

void producer(void){

unsigned char temporary_byte;
/*Co-routine Management Section*/
 if( i_o_var_get( 2) != the_seconds_store ){
 the_seconds_store= i_o_var_get( 2) ;
 temporary_byte= the_seconds_store &240;
 temporary_byte= temporary_byte / 16;
 temporary_byte= temporary_byte + 48;
 seconds_msd= (char)temporary_byte/*Explicit Type Cast*/;
 temporary_byte= the_seconds_store &15;
 temporary_byte= temporary_byte + 48;
 seconds_lsd= (char)temporary_byte/*Explicit Type Cast*/;
 new_second= true;
 }/*end if*/
}/* end of producer*/
```

```
void consumer(void){

char temporary_char;
/*Co-routine Management Section*/
 static int num_of_synch_points = 1;
 switch (num_of_synch_points){
 case  1:goto T3; break;
 case  2:goto T4; break;
 case  3:goto T5; break;
 }/* switch num_of_synch_points*/
 /*End of Co-routine Management Section*/
 T3: ;
 if( char_waiting_at_0 ){
 temporary_char= (char)received_char_0/*Explicit Type Cast*/;
 char_waiting_at_0= false;
 put_char(1,temporary_char);
 }/*end if*/
 if(++num_of_synch_points >  3)
   num_of_synch_points = 1;
 return; /* Synch point*/

 T4: ;
 if( char_waiting_at_1 ){
 temporary_char= (char)received_char_1/*Explicit Type Cast*/;
 char_waiting_at_1= false;
 put_char(0,temporarv_char);
 }/*end if*/
 if(++num_of_synch_points >  3)
   num_of_synch_points = 1;
 return; /* Synch point*/

 T5: ;
 if( new_second ){
 put_char(0,seconds_msd);
 put_char(0,seconds_lsd);
 temporary_char= (char)13/*Explicit Type Cast*/;
 put_char(0,temporary_char);
 temporary_char= (char)10/*Explicit Type Cast*/;
 put_char(0,temporary_char);
 new_second= false;
 }/*end if*/
 if(++num_of_synch_points >  3)
   num_of_synch_points = 1;
 return; /* Synch point*/

}/* end of consumer*/
```

```
void test_prog(void){

int letter_num;
int loop_counter;
char the_letter;
 Set_Vector( 52, serial_port_0_isr);
 Set_Vector( 68, serial_port_1_isr);
 new_second= false;
 char_waiting_at_0= false;
 char_waiting_at_1= false;
 disable_int;
 init_registers();
 set_serial_port(0,9600,'n',8,1);
 set_serial_port(1,4800,'n',8,1);
 enable_int;
 set_dtr(0);
 set_dtr(1);
 letter_num= 1;
 while( true ){/*(Loop Label=L6:)*/
 switch( letter_num ){
 case 1:  the_letter= 'h';
 break;
 case 2:  the_letter= 'e';
 break;
 case 3:  the_letter= 'l';
 break;
 case 4:  the_letter= 'l';
 break;
 case 5:  the_letter= 'o';
 break;
 default :  ; /* null statement */
 break;
 } /*switch*/
 put_char(0,the_letter);
 put_char(1,the_letter);
 loop_counter= loop_counter + 1;
 if( loop_counter > 5) EXIT L6;
 } L6: ;
 while( true ){/*(Loop Label=L7:)*/
 producer();
 consumer();
 } L7: ;
}/* end of test_prog*/

 void main(void){
 test_prog();}/* main*/
```

Appendix G: Listing of Start-up code used to support the test program.

```
; The primary function of the start-up code is to set up the run-time
; environment before passing control to C function main().
; The start-up code performs the following functions:
; 1) Initialize hardware and check RAM.
; 2) Copy initializers from ROM to RAM to setup initialized program variables
;    to proper initial values.
; 3) Zero all uninitialized program variables.
; 4) Setup data segment.
; 5) Setup stack segment.
; 6) Pass control to C function main().
;
; This provides for a fairly minimal start-up of the V25.
; It relies substantially upon the work of: Pillay(1990);
; sample code provided with the locator, from
; Systems & Software, Inc. Irvine California;
; and sample code provided with the V25 development system
; purchased from Sturt Technology, Adelaide, South Australia.
;
; This code is included solely to give an indication of what needs
; to be done before the High level language program receives
; control of the CPU.
;
      NAME MSC_START_UP_CODE
;
          PUBLIC __acrtused
__acrtused        EQU     1
; setting __acrtused in this manner prevents the Microsoft linker from
; pulling in the DOS dependant startup code.
;
; Specify stack size.
;STACK_SIZE   EQU     1000H
;
; All Segment names used conform to those used by Microsoft C Version 6.0a
; refer to the Microsoft C compiler manuals for further explanation.
;
BEGFDATA SEGMENT PARA PUBLIC 'FAR_DATA_BEG'
          PUBLIC _bfdata
_bfdata LABEL BYTE     ; the beginning of initialized data
                       ; in FAR_DATA class.
                ; key = (_bfdata =>begin far data)
BEGFDATA ENDS
;
FAR_DATA_START SEGMENT PARA PUBLIC 'FAR_DATA'
FAR_DATA_START ENDS

ENDFDATA SEGMENT PARA PUBLIC 'FAR_DATA_END'
          PUBLIC _efdata
_efdata LABEL BYTE     ; the end of initialized data
                       ; in FAR_DATA class.
ENDFDATA ENDS
```

```
BEGFBSS SEGMENT PARA PUBLIC 'FAR_BSS_BEG'
        PUBLIC _bfbss
_bfbss LABEL BYTE       ; the beginning of uninitialized
                        ; data in FAR_BSS class.
BEGFBSS ENDS
;
FAR_BSS_START SEGMENT PARA PUBLIC 'FAR_BSS'
FAR_BSS_START ENDS
ENDFBSS SEGMENT PARA PUBLIC 'FAR_BSS_END'
        PUBLIC _efbss
_efbss LABEL BYTE       ; the end of uninitialized data
                        ; in FAR_BSS class.
ENDFBSS ENDS

BEGHBSS SEGMENT PARA PUBLIC 'HUGE_BSS_BEG'
        PUBLIC _bhbss
_bhbss LABEL BYTE       ; the beginning of uninitialized
                        ; data in HUGE_BSS class.
BEGHBSS ENDS
HUGE_BSS_START SEGMENT PARA PUBLIC 'HUGE_BSS'
HUGE_BSS_START ENDS
ENDHBSS SEGMENT PARA PUBLIC 'HUGE_BSS_END'
        PUBLIC _ehbss
_ehbss LABEL BYTE       ; the end of uninitialized data
                        ; in HUGE_BSS class.
ENDHBSS ENDS

DGROUP GROUP NULL,_DATA,CONST,ENDDATA,_BSS,ENDBSS,STACK

NULL SEGMENT PARA PUBLIC 'DATA_BEG'
; This segment contains 16 bytes of zeros.
; If a (DS:0) null pointer assignment
; occurs, these byte locations will be overwritten.
; We can use this to check for null pointer assignment.
        PUBLIC _bdata   ; the beginning of initialized data.
_bdata  LABEL BYTE
        DB 16 DUP (0)
NULL ENDS

_DATA SEGMENT WORD PUBLIC 'DATA'
; Segment with class name DATA contains initialized variables.
_DATA ENDS

CONST SEGMENT WORD PUBLIC 'CONST'
; Segment with class name CONST contains constants.
CONST ENDS

ENDDATA SEGMENT PARA PUBLIC 'DATA_END'
        PUBLIC _edata
_edata  LABEL BYTE      ; the end of initialized data.
ENDDATA ENDS

_BSS    SEGMENT WORD PUBLIC 'BSS'
; Segment with class name BSS contains uninitialized variables.
_BSS    ENDS
```

```
ENDBSS          SEGMENT WORD PUBLIC 'BSS_END'
        PUBLIC _end
_end    LABEL BYTE    ; the end of uninitialized data.
ENDBSS          ENDS

STACK SEGMENT PARA STACK 'STACK'
        DW STACK_SIZE DUP (?)
stack_top LABEL WORD
STACK ENDS
;
;we must declare the C entry point so the assembler code can find it
        EXTRN _main:FAR         ;C main()
STARTUP_TEXT SEGMENT PARA PUBLIC 'CODE'
ASSUME  CS:STARTUP_TEXT
        ASSUME DS:DGROUP, SS:DGROUP

        PUBLIC START_
START_:

        PUBLIC _start_  ;Must be paragraph aligned (i.e. offset is 0)
_start_:                ;and the address where program code starts.

        CLI     ; turn off the interrupts at the earliest possible moment!!
; Initialise hardware by re-locating
; the on-chip V25 registers
; from their old location of 0FFFFH to
; 0F7E0H. Why there? Because the example
; code I first used located to this point.
; It worked, so I left well alone!!!
; For internal register mapping, refer to NEC
; V25 documentation, specifically:
;"uPD70320/322 (V25tm) 16 Bit, Single Chip CMOS Microcomputers"
; available from NEC Electroncis Inc.
;  or
; George Brown Group
; 294 South Road,
; Hilton, S.A. 5033.
; Australia.
;
;
IDB_SEG         EQU 0FFFFH
IDB_LOC         EQU 0FH

NEW_IDB_SEG EQU 0F7E0H
NEW_IDB_LOC EQU 0F7H

        MOV AX, IDB_SEG
        MOV ES, AX
        MOV AL, NEW_IDB_LOC
        MOV ES:BYTE PTR IDB_LOC, AL
;now address at new location
        MOV AX, NEW_IDB_SEG
        MOV ES, AX
        MOV AL, 0FFH
        MOV ES:BYTE PTR 0102H, AL
        MOV AL, 0B2H
        MOV ES:BYTE PTR 01E1H, AL
        MOV AX, 05555H
```

```
        MOV ES:WORD PTR 01EBH, AX
        MOV AL, 04CH
        MOV ES:BYTE PTR 01EBH, AL
        MOV AL, 040H
        MOV ES:BYTE PTR 010AH, AL
;
; Perform variable initialization.  Initializers are copied from ROM to RAM.
;
        PUBLIC _init_begin
_init_begin:
        CLD
;       Transfer Count
        MOV AX,OFFSET DGROUP:_edata     ; Transfer counter
        CMP AX,0
        JZ no_init_data
        MOV CX,AX
;       Destination
        MOV AX,SEG _bdata
        MOV ES,AX       ; Destination ES:[DI]
        MOV DI,0        ; Start of initialized variable area in RAM
;       Source
        MOV AX,SEG _etext    ; Source DS:[SI]
        MOV DS,AX       ; Start of initializer storage in ROM
        MOV SI,0
;       Begin BYTE transfer
REP     MOVSB           ; Begin byte transfer from ROM to RAM
no_init_data:
;  Clear uninitialized data area in DGROUP group

        MOV CX,OFFSET DGROUP:_end ; End of 'BSS' class in RAM
        MOV DI,OFFSET DGROUP:_edata ; Start of 'BSS' class in RAM
        SUB CX,DI       ; Size of 'BSS' class in bytes
        JCXZ no_uninit_data
        MOV AX,0        ; Initialize to 0
REP     STOSB
no_uninit_data:
;  Initialize FAR_DATA data in RAM with initializers stored in ROM

;       Transfer Count
        MOV AX,SEG _bfdata
        MOV CX,SEG _efdata
        SUB CX,AX       ; Compute size of FAR_DATA segments in paragraphs
        JCXZ loopend    ; No FAR_DATA class
        MOV DX,CX       ; Saves transfer count in paragraphs
;       Destination
        MOV ES,AX       ; Destination ES:[DI]
        MOV DI,0        ; Start of FAR_DATA class in RAM
;       Source
        MOV AX,SEG _etext    ; Source DS:[SI]
        MOV DS,AX       ; Start of FAR_DATA initializer storage in ROM
        MOV SI,OFFSET DGROUP:_edata     ; _edata is paragraph aligned
;       Normalize Source Pointer
        MOV AX,SI       ; Process base of source pointer
        MOV CL,4
        SHR AX,CL       ; Divide by 16
        MOV BX,AX
        MOV AX,DS
        ADD AX,BX
```

```
        MOV DS,AX       ; Adjust base of source pointer
        MOV SI,0        ; Offset of source pointer is zero
        MOV AX,DX       ; Restore transfer count in paragraphs
loopbegin:
        CMP AX,1000H    ; More than 64K bytes to transfer?
        JBE lastxfer    ; No
        MOV CX,8000H    ; Prepare to transfer 8000H words
        SUB AX,1000H    ;
        JMP SHORT xferbegin
lastxfer:
        MOV CL,3
        SHL AX,CL       ; Number of WORDs = paragraph * 8
        MOV CX,AX       ; Set up transfer count in terms of WORDs
        MOV AX,0        ; No more to transfer
xferbegin:
REP     MOVSW                   ; Transfer WORDs from ROM to RAM
        CMP AX,0        ; Any more data to transfer?
        JE loopend      ; No
        ; Adjust Source and Destination pointers
        MOV BX,AX       ; Saves transfer count
        MOV AX,DS
        ADD AX,1000H
        MOV DS,AX
        MOV AX,ES
        ADD AX,1000H
        MOV ES,AX
        MOV SI,0
        MOV DI,0
        MOV AX,BX       ; Restores transfer count
        JMP loopbegin
loopend:

;   Clear uninitialized data area in FAR_BSS class
;       Transfer Count
        MOV AX,SEG _bfbss
        MOV CX,SEG _efbss
        SUB CX,AX       ; Compute size of FAR_BSS segments in paragraphs
        JCXZ loopfend   ; No FAR_BSS class
;       Destination
        MOV ES,AX       ; Destination ES:[DI]
        MOV DI,0        ; Start of FAR_BSS class in RAM
;       Transfer Count
        MOV AX,CX
loopfbegin:
        CMP AX,1000H    ; More than 64K bytes to initialize?
        JBE lastfxfer   ; No
        MOV CX,8000H    ; Prepare to transfer 8000H words
        SUB AX,1000H    ;
        MOV BX,AX       ; Saves transfer count
        JMP SHORT xferfbegin
lastfxfer:
        MOV CL,3
        SHL AX,CL       ; Number of WORDs = paragraph * 8
        MOV CX,AX       ; Set up transfer count in terms of WORDs
        MOV AX,0        ; No more to transfer
        MOV BX,AX       ; Saves transfer count
xferfbegin:
        MOV AX,0
```

```
REP     STOSW           ; Initialize WORDs to zero
        MOV AX,BX       ; Restore transfer count
        CMP AX,0        ; Any more data to transfer?
        JE loopfend     ; No
        ; Adjust Destination pointers
        MOV AX,ES
        ADD AX,1000H
        MOV ES,AX
        MOV DI,0
        MOV AX,BX       ; Restore transfer count
        JMP loopfbegin
loopfend:

;  Clear uninitialized data area in HUGE_BSS class
;       Transfer Count
        MOV AX,SEG _bhbss
        MOV CX,SEG _ehbss
        SUB CX,AX       ; Compute size of HUGE_BSS segments in paragraphs
        JCXZ loophend   ; No HUGE_BSS class
;       Destination
        MOV ES,AX       ; Destination ES:[DI]
        MOV DI,0        ; Start of HUGE_BSS class in RAM
;       Transfer Count
        MOV AX,CX
loophbegin:
        CMP AX,1000H    ; More than 64K bytes to initialize?
        JBE lasthxfer   ; No
        MOV CX,8000H    ; Prepare to transfer 8000H words
        SUB AX,1000H    ;
        MOV BX,AX       ; Saves transfer count
        JMP SHORT xferhbegin
lasthxfer:
        MOV CL,3
        SHL AX,CL       ; Number of WORDs = paragraph * 8
        MOV CX,AX       ; Set up transfer count in terms of WORDs
        MOV AX,0        ; No more to transfer
        MOV BX,AX       ; Saves transfer count
xferhbegin:
        MOV AX,0
REP     STOSW           ; Initialize WORDs to zero
        MOV AX,BX       ; Restore transfer count
        CMP AX,0        ; Any more data to transfer?
        JE loophend     ; No
        ; Adjust Destination pointers
        MOV AX,ES
        ADD AX,1000H
        MOV ES,AX
        MOV DI,0
        MOV AX,BX       ; Restore transfer count
        JMP loophbegin
loophend:
;
```

```
; Setup data and stack segment here
;
        MOV AX,DGROUP
        MOV DS,AX       ; Setup data segment
        MOV ES,AX
        ASSUME DS:DGROUP
        MOV SS,AX       ; Setup stack pointer
        MOV SP,OFFSET DGROUP:STACK_TOP
        ASSUME SS:DGROUP

        CALL _main      ; Pass control to C main() function
        HLT             ; Embedded programs have no right to
                        ; return, so halt the processor.
STARTUP_TEXT        ENDS
;
; the next section provides for an area where initialised data
; can be copied from. This data is normally placed after the program
; code by the locator.
;
C_ETEXT SEGMENT PARA PUBLIC 'CODE_END'
        PUBLIC _etext
_etext  LABEL BYTE  ; This label marks the end of program code.
    DB 16 DUP(?)    ; Required bytes
C_ETEXT ENDS
;
        END START_  ; The end of the assembler code.
```

End Text References

Allinson, C. (1994, March), ROMLDR, an Embedded System Program Locator. *The C Users Journal.* pp35-46.

Akerbaek, T. (1993, March). C++, Coroutines and Simulation. *The C Users Journal* pp74-86.

Appleby, D. (1991). *Programming Languages: Paradigm and Practice.* New York: McGraw-Hill.

Arjomandi, E., O'Farrell, W. & Kalas, I (1994, Jan). Concurrency Support for C++: an Overview. *C++ Report* pp. 45-50.

Aucsmith, D. (1988, March). *Ada and Embedded Processors, Experience with ALS/N.* Paper presented at SouthCon 88. Orlando Florida USA.

Baker, T.P. (1988). An improved Run-Time System Interface. *The Journal of Systems and Software,* 8(5) 373-393.

Bentley, J. (1986) Programming Pearls. *Communications of the ACM.* 29(8) pp711-721.

Bhansali, Praful V., Pflug, Bryan K., Taylor, John A., Wooley, John D. (1991) Ada Technology: Current Status and Cost Impact. *Proceedings of the IEEE,* 79(1) 22-29.

Boehm, B. W. (1981), *Software Engineering Economics,* Englewood Cliffs, NJ: Prentice-Hall.

Booch, Grady. (1987). *Software Engineering with Ada.* Menlo Park, California: The Benjamin Cummings Publishing Company, Inc.

Booch, Grady (1987a). *Software Components with Ada.* Menlo Park, California. : The Benjamin Cummings Publishing Company, Inc.

Boussinot, F. & De Simone, R. (1991, Sept). The Esterel Language. *Proceedings of the IEEE.* 79(9) pp1293-1304.

Brinch-Hansen, P.(1975). The Programming Language Concurrent Pascal. In Horowitz. E (Ed) 1985 *Programming Languages: A Grand Tour.* Rockville. Computer Science Press.

Brosgol, B. (1990, Sept.). Ada's fundamental language structures build reliable systems. *EDN* -pp153-166.

Burns, A. & Wellings, A. (1990). *Real-time systems and their programming languages.* Wokingham, UK.: Addison WesleyPublishing Company Ltd.

Colton, R. (1988 March). *Ada on a Bare Machine.* Paper presented at SouthCon 88. Orlando Florida USA.

Cotigny, J.D. & Ple, B. (1991) *Design of a Man-Machine Interface for a Computerised Numerical Controller.* Paper presented to the 1991 International Conference on Industrial Electronics, Control and Instrumentation.

Cullens, C (1994, March). Cross-platform development with Visual C++. *Dr. Dobb's Journal,* 19(3), 64-69.

Cullyer, J. Lucas Professor of Electronics, University of Warwick (1993) Editorial on Safety Critical Systems. *Microprocessors and Microsystems* 17(1) p2.

Deitel (1990). *An Introduction to Operating Systems* Menlo Park: Addison-Wesley Publishing Company.

Digitalk (1992). *Smalltalk V for Windows: Tutorial and Programming Handbook.* Los Angeles: Digitalk Inc.

Dijkstra, E.W.(1965). Co-operating Sequential Processes. In Genuys. F. (Ed) 1968 *Programming Languages.* New York: Academic Press, pp 43-112.

Dobler, H. (1992). Ada on Personal Computers - Some Experiences. *Structured Programming,* 13(4) 193-201.

Duhaut, Bidaud & Fontaine (1992, Sept.). Iada, A language for Robot Programming based on Ada. *Robotics and Autonomous Systems.* pp299-304.

Eckel B. (1993). *C++ Inside & Out,* Berkeley, California, Osborne McGraw-Hill.

Emery and Nyberg (1989). Observations on Portable Ada Systems. In Alvarez. A. (Ed) 1989 *Ada- the Design Choice.* Cambridge: Cambridge University Press, pp 245-255.

Fischer, C. N. & Leblanc, R.J. (1991) *Crafting a Compiler with C.* Menlo Park, California: The Benjamin Cummings Publishing Company. Inc.

Gehaini, N. (1989). *Ada: An Advanced Introduction.* Englewood Cliffs, N.J.: Prentice Hall.

Gehani , N.& Roome, W. (1986, Sept) Concurrent C. *Software-Practice and Experience.* 16(9), p821-844.

Gehani, N. & Roome, W. (1993) *The Concurrent C Programming Language.* Summit NJ: Silicon Press.

Gligor. V & Luckenbaugh G. (1983). *An Assessment of the Real-time Requirements for Programming Languages.* Paper presented to the Proceedings of the Real-time Systems Symposium, Dec 1993.

Harp, K. (1988 March). *Ada on a Limited Address Machine.* Paper presented at SouthCon 88. Orlando Florida USA.

Hoare, C.A.R. (1973, Oct). Hints on Programming Language Design. In Horowitz, E. (Ed) 1985. *Programming Languages: A Grand Tour.* pp. 31-40.

Hoare, C.A.R. (1974, Oct). Monitors: An Operating System Structuring Concept. *Communications of the ACM.* 17(10). pp 549-557.

Hoogeboom. B, & Halang W.A. (1991, Sept.) *The Concept of Time in Software Engineering for Real-Time Systems.* A paper presented to the 3rd International Conference on Software Engineering for Real-Time Systems, Cirencester, U.K. .

Isherwood, D. (1991) *DOS and real-time?.* paper presented at the Third International Conference on Software Engineering for Real-Time Systems. Cirencester, U.K.

Kamrad, Mike. (1992). The Catalogue of Interface Features and Options: Bridge to the Future for Real-Time Ada Applications. In A.Burns (Ed.), *Towards Ada 9X* (pp9-49). Amsterdam: IOS Press.

Laplante, Phil. (1993). *Real-Time Systems Design and Analysis: An Engineers' Reference.* New York: The Institute of Electrical and Electronic Engineers.

Liskov & Snyder (1979). Exception Handling in CLU. In Horowitz. E (Ed) 1985 *Programming Languages: A Grand Tour.* Rockville. Computer Science Press.

MacLennan, Bruce J. (1987). *Principles of Programming Languages: Design, Evaluation and Implementation.* HRW, The Dryden Press: New York.

Mann, Daniel. (1992). The Universal Debugger Interface. *Dr. Dobb's Journal*, 17(9),58-68.

Mark Williams Company (1992). *Coherent Operating System Manual* Illinois: Mark Williams Company.

Mazur, Beth. (1992). Moving from Assembly to C. *Dr. Dobb's Journal,* 17(8), 72-84.

Meyer, Bertrand. (1988). *Object-Oriented Software Construction.* New York, Prentice-Hall,

Mortensen, B. (1984) *Use of Concurrent Pascal in Industrial Systems Programming.* Microprocessing and Microprogramming 14 p155-159

Microsoft Corporation, (1991) *MS-DOS Programmer's Reference* Redmond, Washington: Microsoft Press.

Norton, P. & Yao, P. (1992) *Borland C++ Programming in Windows* New York: Bantam Books.

Odette, Louis L. (1991). *Intelligent Embedded Systems.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, USA.

Parnas, D.L., & Clements, P.C., (1972, Dec) On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM* 15(12) pp1053-1058.

Petzold, C. (1992), *Programming Windows.* Redmond, Washington: Microsoft Press

Phillips, Stuart G., & Rowett, Kevin J. (1991). C++ for Embedded Systems, *Dr. Dobb's Journal,* 16(10), 76-85.

Pillay, Kenneth D. (1990). Relocating loader for MS-DOS .EXE files. *Microprocessors and Microsystems* 14(7), 427-434.

Powers, R.D., & Roark, C. (1990). Ada Support for Real-Time Systems. *Ada Letters.* 10(4) 114-118.

Sebesta, R.W. (1989). *Concepts of Programming Languages.* California: The Benjamin Cummings Publishing Company Inc.

Sims, J. Terry. (1991, October). *An Enhanced Ada Run-Time System for Real-Time Embedded Processors.* Paper presented at the IEEE/AIAA 10th Digital Avionics Systems Conference, Los Angeles, California, USA.

Snow, C.R. (1992). *Languages for Programming.* Cambridge UK: Cambridge University Press

Sommerville, I. (1990). *Software Engineering.* Wokingham: Addison Wesley.

Stapfer, Christian. (1992). Timed Callbacks in C++, *Dr. Dobb's Journal,* 17(10), 72-76.

Stoyenko, A.D.& Kligerman, E., (1986, Sept). *Real-Time Euclid: A Language for Reliable Real-Time Systems.* Transactions on Software Engineering, 12(9) pp941-949.

Struble, D.G., & Wagner, M.J., (1989). *A quantitative evaluation of interrupt handling capabilities in Ada.* Paper presented at TRI-Ada '89, Pittsburgh, USA.

Swartout, William & Balzer, Robert., (1982) On the Intertwining of Specification and Implementation. *Communications of the ACM.* 25(7), 438-440.

Topping, G., Yeung, W.L. (1990), Implementing JSD Designs in Ada- A Tutorial. *ACM SIGSOFT Software Engineering Notes* 15(3), 25-33.

Topping, G., Yeung, W.L. (1991, September). *A Formalisation of Jackson System Development.* A paper presented at the 3rd International Conference on Software Engineering for Real Time Systems at the Royal Agricultural College, Cirencester, UK.

Tucker-Taft, S. (1993). *Ada 9X:From Abstraction-Oriented to Object-Oriented.* Paper presented at the 8th Annual Conference of OOPSLA'93, pp127-136.

Turbo C++ Programmers Guide (1990). Borland International, Scotts Valley CA USA.

Velastin (1991, September). *An Approach to Modular Programming in C.* A paper presented at the 3rd International Conference on Software Engineering for Real Time Systems at the Royal Agricultural College, Cirencester, UK.

Volkman, V. (1993, November). BCC Coroutines, TDE, Lost Algorithms, and Anthony's Tools. *The C Users Journal.* pp119-121.

Walraet, Bob. (1989). *Programming, The Impossible Challenge.* Amsterdam: Elsevier Science Publishers B.V.

Welsh, T. (1993, October). Debugging Embedded Systems. *The C Users Journal.* pp19-30.

Wirth, Niklaus. (1989, January). Designing a System from Scratch. *Structured Programming.*pp10-18.

Wirth N. & Gutknecht, J. (1992). *Project Oberon: The design of an Operating System and Compiler*. New York. ACM Press.

Zave, Pamela & Yeh, Raymond T.(1981, March 9-12). *Executable Requirements for Embedded Systems.* Paper presented at the 5th International Conference on Software Engineering. San Diego California.

Zave, Pamela. (1982). An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering,* SE-8(3), 250-269.