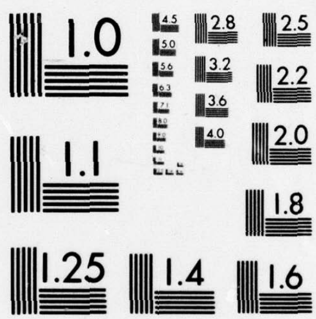
AD-A054 611 NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA F/G 9/2
OVERVIEW OF CONCURRENT PASCAL FOR THE INTERDATA 7/16, DEVELOPED--ETC(U)
FEB 78 D M COTTEL
UNCLASSIFIED NOSC/TD-146 NL

| OF |
AD
A054611

END
DATE
FILMED
7 -78
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

NOSC

(14) NOSC/TD-146

Technical Document 146

(6)

OVERVIEW OF CONCURRENT PASCAL FOR THE INTERDATA 7/16

Developed for the Channel Adaptive Receiver Testbed

(10) D.M. Cottel

(11) 15 February 1978

(9) Final Report, February—November 1976

Prepared for
Naval Electronic Systems Command
and
Naval Sea Systems Command

(12) 15p.

(16) F21221

(17) XF212217d1

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

RR GAVAZZI, CAPT, USN
Commander

HL BLOOD
Technical Director

ADMINISTRATIVE INFORMATION

The work reported here was sponsored by NAVELEX 310/NAVSEA 06H1, under Task Number XF21.221.701.U011 and pursued under the Acoustic Communications Exploratory Development Block Program.

Released by
RH Hearn, Head
Electronics Division

Under authority of
DA Kunz, Head
Fleet Engineering Department

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NOSC TD 146	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* OVERVIEW OF CONCURRENT PASCAL FOR THE INTERDATA 7/16 Developed for the Channel Adaptive Receiver Testbed		5. TYPE OF REPORT & PERIOD COVERED Final Report February – November 1976 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) DM Cottel		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Ocean Systems Center San Diego, CA 92152		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS XF 21.221.701.U011
11. CONTROLLING OFFICE NAME AND ADDRESS NAVELEX 310 and NAVSEA 06H1 Washington, DC		12. REPORT DATE 15 February 1978 13. NUMBER OF PAGES 12 pages
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release. Distribution Unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Concurrent Pascal, Interdata, Kernel, minicomputer operating systems, peripheral devices

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report describes the software system developed to support the Channel Adaptive Receiver (CAR) Project at the Naval Ocean Systems Center. Reasons are given for choosing the programming language Concurrent Pascal, as a basis for the software support system. An overview of the Concurrent Pascal implementation for the Interdata 7/16 minicomputer is presented.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

CONTENTS

1. INTRODUCTION . . . page 2
2. CHOOSING THE LANGUAGE . . . 2
3. IMPLEMENTATION FOR THE INTERDATA 7/16 . . . 6
4. PERFORMANCE . . . 10
5. CONCLUSIONS . . . 11
6. REFERENCES . . . 12

1. INTRODUCTION

This report describes the software system developed to support the Channel Adaptive Receiver (CAR) project at the Naval Ocean Systems Center (NOSC). This project involved building a test facility for evaluating an underwater communication technique.

The significant factor in the software system is the use of the programming language Concurrent Pascal recently developed by Per Brinch Hansen[1]. In Concurrent Pascal, the more widely known language Pascal[2] is extended to include the ability to handle concurrent tasks, that is, tasks which are carried on simultaneously within a computer.

The following sections give the reasons for the choice of the language as the basis for the software support system and then describe the Concurrent Pascal implementation for an Interdata 7/16 minicomputer. Some data describing the compiler size and performance is included for those who may be considering Concurrent Pascal for their own installation. Finally, there are some observations on the success of this software development.

For additional information concerning the hardware systems, refer to the hardware documentation[3]. A user's manual for concurrent programming on the CAR facility also exists containing considerable detail concerning the available software and its use[4].

2. CHOOSING THE LANGUAGE

There are apparently many alternative approaches to developing minicomputer software. Programs can be written in machine Assembly Language immediately upon delivery of the hardware. Or, the manufacturers will provide (at a price) various software routines and operating systems. What factors, then, led to the decision to invest the time and manpower in developing the Concurrent Pascal language for the support of the CAR system?

2.1 HARDWARE REQUIREMENTS

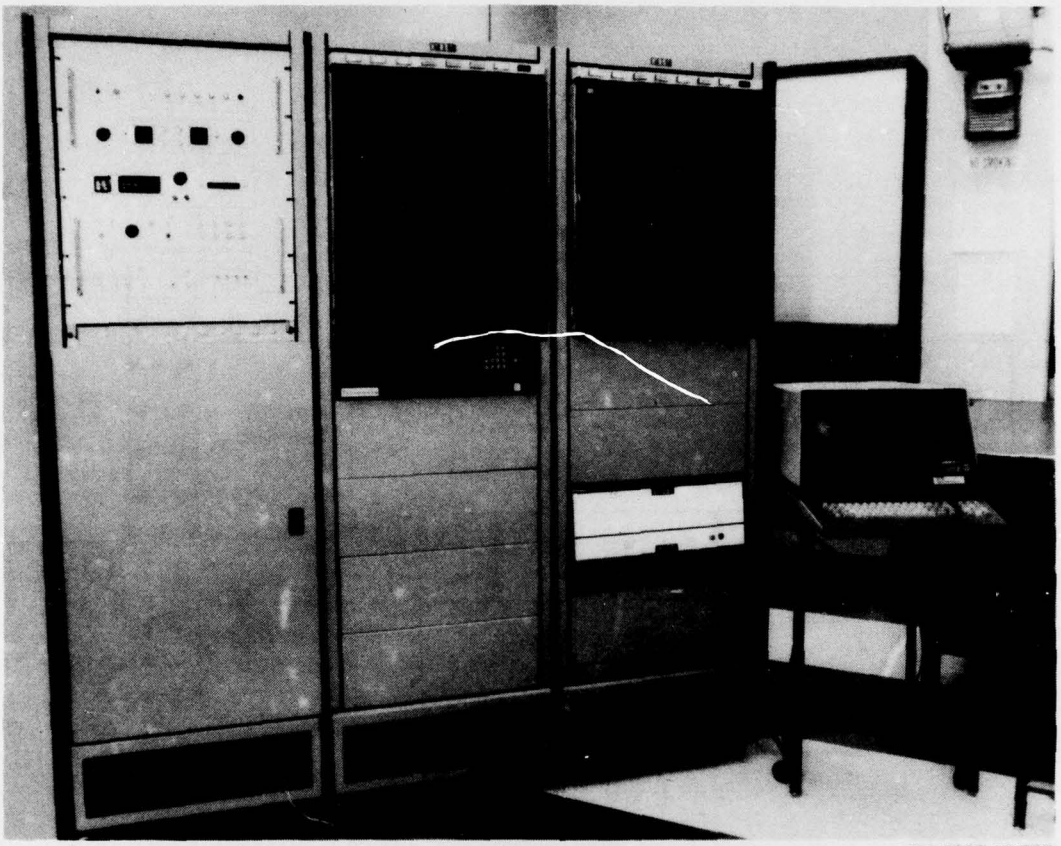
The CAR facility allows an experimenter to establish a test configuration from a minicomputer terminal, enter the various test parameters, initiate the test, and display or record the results (Figure 1). The hardware involved is an assembly of standard minicomputer hardware based on an Interdata 7/16[5], various peripherals, and several devices built by engineers at NOSC[6]. The configuration of these devices is shown in Figure 2.

There are three important hardware features which influenced software decisions:

1. multiple peripherals in operation simultaneously,
2. real-time operation required, and
3. unique NOSC built peripheral hardware.

---

1. Brinch Hansen, P., "The Programming Language Concurrent Pascal," *Information Science*, California Institute of Technology, Pasadena, CA, February 1975
2. Jensen, K. and Wirth, N., *Pascal: User Manual and Report*, Springer-Verlag, New York, N.Y., 1974
3. Juniper, M.D., "Overview of the Adaptive System Hardware Developed for the Channel Adaptive Receiver Program," TD 141, Naval Ocean Systems Center, San Diego, CA, 1977
4. Cottel, D.M. and Zaun, J.A., "Concurrent Pascal: User's Manual for the Interdata 7/16," Preliminary Release, Naval Ocean Systems Center, San Diego, CA, December 1976
5. "Model 7/16 Users Manual," Interdata, Inc. Publication number 29-261, 1971.
6. Juniper, M.D., "Overview of the Adaptive System Hardware Developed for the Channel Adaptive Receiver Program," TD 141, Naval Ocean Systems Center, San Diego, CA 1977



LRO 7303-12-77B

Figure 1. Car Testbed Facility

The minicomputer is primarily used for sequencing operations and for control of devices and data flow. During experiments it is necessary to operate the displays, for instance, at the same time that data is being read from the disk or tape (or both). In addition, as is often the case in signal processing applications, both data inputs and display outputs must be handled in real-time. This requires that as much of the input/output operations as possible be overlapped; for example, the next input data sample cannot be held up until a tape operation completes.

The handling of concurrent operations is one of the most difficult of programming tasks. Most programmers (indeed, most people) are experienced in handling one task after another in a sequential fashion. The proper handling of the synchronization of concurrent tasks requires the greatest care in preventing unwanted and unforeseen interactions between tasks[7]. The Concurrent Pascal language provides the tools for correctly handling these interactions. In fact, by enforcing certain rules and access rights, it totally prevents the programmer from making many potentially dangerous constructions.

7. Habermann, A.N., "Introduction to Operating Systems Design," Science Research Associates, Inc., Palo Alto, CA, 1976

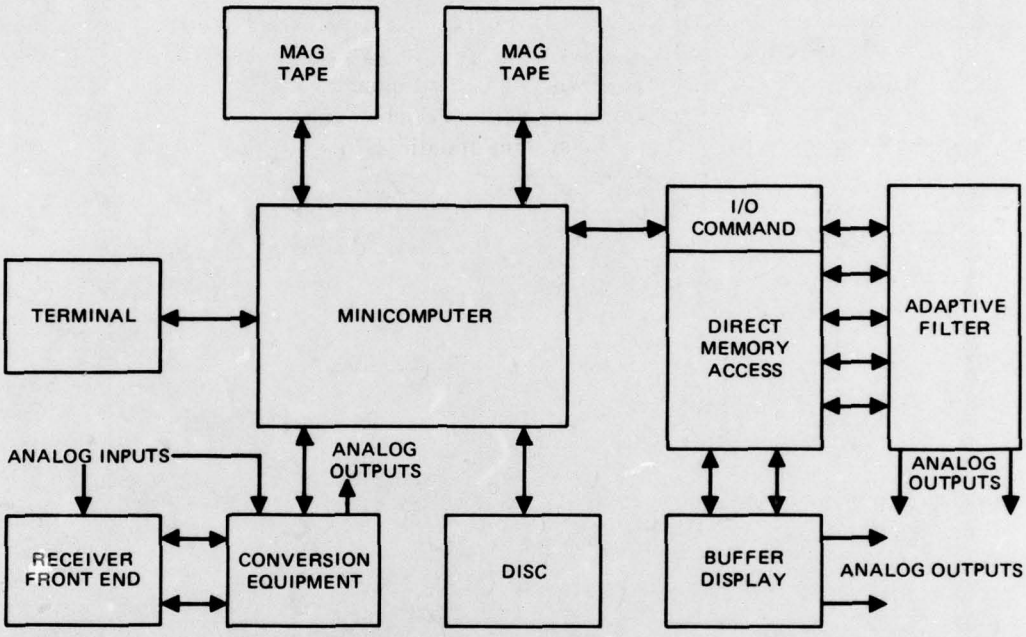


Figure 2. Test-bed component interconnections.

Finally, although Interdata has available operating systems which are for real-time applications, they would first have to be modified extensively to handle those peripheral devices built by NOSC. These include real-time displays and fast data buffering through direct memory access channels. The input/output portions of a typical operating system are usually the most difficult portions to understand, with many special purpose details for each unique device. For these reasons, the "simple modification of an existing system" is actually much more difficult than is immediately apparent.

2.2 OPERATIONAL REQUIREMENTS

It must be kept in mind that those using the CAR testbed are not programmers but signal processing analysts, engineers, and technicians. It is important that the software not only avoid crashing, but also be reliable in the sense that it will always do what the user expects it to do. This means he must be confident that he can specify a test and be sure that the results are the correct results. In addition, when errors do occur, the system must make it clear to the user exactly what the problems are.

Because of the nature of an experimental facility, it is a fact of life that software changes will be frequent. Each new user will require different data, access to previously unavailable parameters, or a myriad of other unforeseen possibilities. It is mandatory, therefore, that the software system be changeable in such a way that the changes have minimum ramifications in other areas of the program, and in a way that will not require

an extensive debugging effort. Here again, the result must be no less reliable than the original system. This necessity requires that the code be readable and understandable. In addition, for any software system to be complete, normal documentation is required.

Studies have shown[8] that three quarters of the total cost of software development is actually incurred after the initial development and testing. That fraction would normally be expected to increase for the case where frequent changes are certain. Making the code reliable and changeable is economically justified. Such a code provides cost reductions when changing needs of users require system updating.

2.3 ALTERNATIVES

The method of implementing the CAR software support system falls to these three choices:

1. use assembly language
2. modify an Interdata operating system
3. use a high level concurrent language

Here is a summary of the advantages and disadvantages of these choices.

2.3.1 ASSEMBLY LANGUAGE

The use of the Interdata Assembler has two distinct advantages: it is immediately available without requiring extensive development time and effort, and it can be made (by a programming artist) very efficient in terms of memory space required and the speed of execution of the code. Unfortunately, these advantages are overwhelmed by the disadvantages. Correct assembly language coding requires extreme discipline to avoid situations where changes have repercussions in many other parts of the code. The additional problems of writing concurrent code have already been mentioned. Assembly language code requires more debugging time than that written in a high level language. Further, the very process of making the code efficient and fast also causes it to be totally unintelligible and therefore virtually impossible to change without creating additional problems which require further debugging. Making the major revisions which are certain to be required in this experimental system would be prohibitively expensive.

For all this, assembly language has of necessity usually been the choice for these problems. Since no high level language was available, system designers simply had to require much less flexibility. In some cases, a hybrid configuration can be used by writing the concurrent parts in assembly language and using some higher language for those portions of a sequential nature.

2.3.2 OPERATING SYSTEM MODIFICATION

This has most of the same disadvantages as assembly language usage, with the advantages greatly diluted. Although apparently a lot of work has been done already, in fact the code is still in assembly language and is therefore subject to all the same problems.

---

8. Ross, D.T., "Homilies for Humble Standards," Comm. ACM, 11, 19 November 1976.

Adding to or modifying an existing program of this nature, usually without adequate internal documentation, is very difficult – yet the drivers for the NOSC hardware must be added. Further, the real-time aspects may have been optimized for some other application and would require further modification for the CAR system in particular. Finally, the minicomputer software provided by the manufacturers is known to be rudimentary even for simple sequential uses. In reality, most users with real-time requirements (and many without) simply write their own special purpose programs, rather than try to make available routines fit their problem.

2.3.3 HIGH LEVEL CONCURRENT LANGUAGE

Until recently, of course, this was not one of the available choices. The availability of Concurrent Pascal is a significant step toward generating reliable and understandable real-time operating systems. Concurrent Pascal provides the structured language facilities of Pascal as well as the concurrent tools necessary to properly support the CAR testbed. This means the code can be written in an understandable way with all aspects of concurrent task interaction becoming more apparent. The modularity enforced by the language ensures that small changes do not propagate to other parts of the program.

A potential disadvantage is that a compiler for a high level language usually generates machine code which takes more space and runs slightly slower than that produced by an expert programmer. Experts are not usually available, however, so only in extreme circumstances is this too high a price. In fact, it makes more sense for both economic and reliability reasons to compensate for these drawbacks, if necessary, by adding more memory, making hardware modifications, or developing an optimizing compiler.

2.4 THE DECISION

The only apparent reason not to choose Concurrent Pascal as the CAR support system would be if the cost or time of developing the language for the Interdata 7/16 were prohibitive. The availability of the compiler and other implementation documentation from Per Brinch Hansen at a nominal price made the decision clear.

3. IMPLEMENTATION FOR THE INTERDATA 7/16

3.1 CONCURRENT PASCAL

Concurrent Pascal was designed by Per Brinch Hansen as a language for structured programming of computer operating systems[9]. Because of the need to handle multiple peripheral devices simultaneously, the language is ideally suited to the CAR support problem. The sequential language Pascal has been extended to include concurrent constructions, primarily processes and monitors, which allow the compiler to check access rights and various task interactions before the program is ever executed.

The language constructions which Concurrent Pascal provide[s] lead to the writing of relatively short and simple modules whose function is easy to [u]nd check. This type

9. Brinch Hansen, P., "Concurrent Pascal Report," *Information Science*, California Institute of Technology, Pasadena, CA, June 1975

of construction makes it extremely simple for the programmer to test parts of the system separately, then assemble the parts with no danger that a new section can create new bugs in an already tested module. Relatively large programs can be put together, mainly from debugged modules, with the result that little new debugging time is required.

The compiler has been modified to meet the needs of the CAR program. As obtained from Brinch Hansen, the output of the compiler was an intermediate language which was executed by interpreting the intermediate instructions using a machine language Kernel program. Since this was too slow for the CAR real-time application, the compiler was changed to generate output code directly in Interdata machine language. Although this procedure increases the size and complexity of the compiler, it dramatically increases the speed at which programs are able to execute.

The original compiler consisted of seven passes. This means that the actual compilation is split into seven sequentially executed programs or passes, with each pass partially reducing the data, then transferring it to the next pass in an intermediate format. The final two passes, which generated interpreter code, were replaced by four new passes (now nine passes altogether) which generate Interdata 7/16 machine instructions. One of the new passes does a peephole optimization of the generated code: modifications such as replacing long instructions with shorter ones, or removing redundant instructions. As a result, the code generated by the modified compiler is really very good: it takes less memory space than that generated by the interpreting version, and it will run many times faster.

Although the compiler was changed internally, the actual language remains the same with the single exception of standard procedures and functions. Some were changed or eliminated, while others were added to provide the specific capabilities needed for the CAR system.

3.2 OPERATING PROCEDURES

The Concurrent Pascal compiler would normally be run on the same computer for which the output code is intended. The compiler output would simply be loaded and executed. This was the case, for example, with Brinch Hansen on his Concurrent Pascal operating system SOLO[10]. Although SOLO was implemented for the Interdata 7/16, the compiler (written in Sequential Pascal) could not be maintained there because of the restricted memory size (64 kilobytes). Time and manpower considerations prevented making the modifications necessary to allow the compilers to run on the Interdata.

An alternate method of preparing Concurrent Pascal programs for execution on the Interdata is therefore used as shown in Figure 3. The facilities of a Univac 1110 computer at NOSC, including the file handling utilities and a text editor, are used to create the source code. In addition, several utility programs were written specifically for the CAR project to provide formatted listings and other text manipulation. Interdata machine code is then generated by cross-compiling, that is, the compiler runs in the Univac 1110 but outputs code for the Interdata. The code is transported on magnetic tape to the minicomputer where the Interdata Disc Operating System[11] and some Interdata utilities are used to prepare a boot tape. When this tape is loaded into the machine using the normal boot procedure, the

---

10. Brinch Hansen, P., "The SOLO Operating System: A Concurrent Pascal Program," *Information Science*, California Institute of Technology, Pasadena, CA, June 1975

11. "Common Assembler Language (CAL) User's Manual," Interdata, Inc., Publication number 29-375, 1974

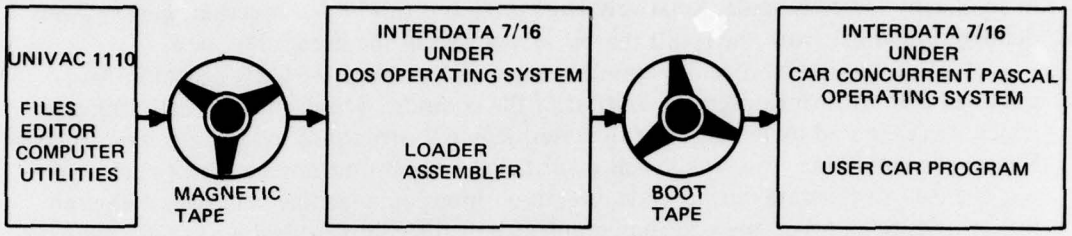


Figure 3. Steps to executing a Concurrent Pascal Program.

Concurrent Pascal program begins execution. It becomes, in effect, the operating system. While this sequence is somewhat elaborate, it allows the use of a large library of existing Univac utilities with a minimum of further software development for the Interdata.

The language and runtime system both support the calling and execution of Sequential Pascal programs. Sequential Pascal is a derivative of the Concurrent language with the concurrent constructions removed and a few restrictions (pointers and recursive routine calls) lifted[12]. To date, these facilities have not been needed for direct CAR support, but are available should the application demand it.

3.3 THE RUNTIME SYSTEM

"Runtime System" means the components which are in operation during the execution of a program. This section describes the following four parts of the runtime system and their general functions and characteristics:

1. the Kernel,
2. the IO machine,
3. Library routines, and
4. Concurrent compiler output.

Their relationship can be visualized as shown in Figure 4.

3.3.1 KERNEL

The Kernel is an assembly language program which implements a hypothetical computer having a set of instructions tailored to running concurrent programs. These virtual instructions include process initialization, process communication signals, and process interaction protection. Also provided are entries to allow waiting for a specified time interval or to await an error. The Interdata memory protection hardware is used by the Kernel to protect the operating system (the Concurrent Pascal program) from inadvertent destruction by users (Sequential Pascal programs). All user and system errors are detected and indicated to the operator.

---

12. Brinch Hansen, P. and Hartmann, A.C., "Sequential Pascal Report," *Information Science*, California Institute of Technology, Pasadena, CA, July 1975

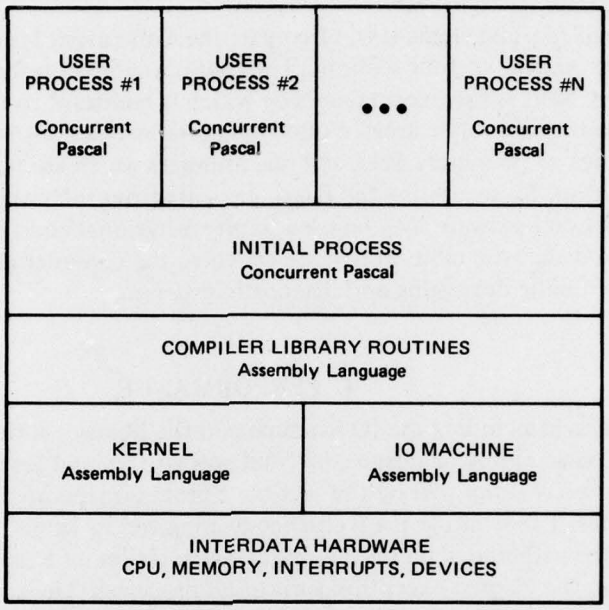


Figure 4. Concurrent Pascal runtime system hierarchy.

3.3.2 IO MACHINE

The IO machine is implemented at the same level as the Kernel. An assembly language program, the IO machine was designed to reduce the machine and device dependent portions of the Concurrent code while still giving the advantages of programming device operations in the higher level language. The details of interrupt handling and device addressing are kept out of the user's hands.

The IO machine is an interpreter: an array of generalized device instructions is passed to the IO machine where they are executed one by one. The instruction set was chosen to provide the basic requirements for device control: issue a command, get the status of the device, and do a data transfer. Additional details are not necessary at the user level. It should not be important, for example, what method is used to transfer the data (IO instructions or direct memory access, bytes or multiple byte transfers) because that is a function of the particular hardware configuration.

3.3.3 LIBRARY

Library routines are used to shorten the code generated by the compiler. When a complicated or lengthy piece of code is required (such as a subroutine call), the compiler simply inserts the code to cause a jump to the library routine. These routines are usually quite small and are written in assembly language. Library routines include the various methods of calling routines, copying large data blocks, and operations on sets (a Pascal data form).

3.3.4 USER PROGRAM

The final runtime element is, of course, the Concurrent Pascal program. The output of the compiler consists of four sections. The first is a table of reference addresses for the library routines. Next is the executable code which is reentrant so that multiple references can be made to the same code areas. Following these instructions is a table of literal constants: strings of characters, sets, and real numbers which are never changed during program execution. By separating the fixed elements from the program data areas, the memory protection hardware may be used to give additional code protection and reliability. In the future, a diagnostic table will be appended to the compiler output to provide the data needed for a symbolic debugging and diagnostic program.

4. PERFORMANCE

The Kernel (including the IO machine and the library routines) consists of 4500 lines of Interdata assembly language code, and was written and tested in five man-months. This task was greatly simplified by the fact the Kernel is made up of many nearly independent routines. The sizes for the Kernel code are given in Table 1. The largest version includes code for gathering data on the frequency and type of Kernel entry, as well as detailed data on the elapsed execution time of all processes. These instructions were removed for the second version. Finally, some of the CAR programs required extraordinary measures to squeeze them into the Interdata memory. For those cases, a minimum Kernel was created by removing all code related to timed waits and sequential programs. The first 700 bytes of memory in the Interdata are reserved for special uses. This space is included in Table 1.

Table 1. Size of Kernel Code.

Kernel Version	Space In Bytes
Full Kernel	9000
No Test Data	8000
Concurrent Only	5900

Modifying the compiler involved not only replacing the final two passes with four new ones, but also changing earlier passes as necessary to provide data for the new mode of code generation. This turned out to be a definite disadvantage of the multiple pass organization (small changes tended to propagate through many passes). The compiler conversion took about one man-year to complete. Table 2 summarizes the results of compiling the nine concurrent passes using the sequential compiler. Code output from the concurrent compiler would show similar characteristics since the code generation and optimization is virtually identical in the two compilers.

The most notable feature of the table is the large range in code reduction obtained by optimizing the output code and removing code to do runtime range checking. The gain is highly dependent on the type of code being written. Pass 2, for example, is almost

Table 2. Sizes of concurrent compiler passes.

Program Name	Final Code Space (Bytes)	Optimization Improvement	Check Removal Improvement	Total Reduction
Pass 1	8828	14%	11%	25%
Pass 2	13302	10	2	12
Pass 3	13070	17	26	43
Pass 4	9994	16	27	43
Pass 5	8758	14	14	28
Pass 6	15008	20	13	33
Pass 7	17150	19	17	36
Pass 8	17930	16	30	46
Pass 9	19540	17	6	23

entirely procedure calls, whereas Pass 8 uses a large number of subrange types. Pass 9 is larger than the others because of the large character arrays required to print error messages in a readable way.

5. CONCLUSIONS

All who have had occasion to write concurrent programs for the CAR project agree that it is not only amazingly easy to assemble complex concurrent jobs, it is also fun to do. During the time the device drivers were being developed, many concurrent programs of 2000 to 3000 lines were written and tested within a week or two. Once compiled, the programs nearly always ran when first loaded. The usual case was to find that the design specifications for the test program needed revision. Since the Kernel testing was completed, there have never been any time dependent bugs. This remarkable experience is probably due not only to the concurrent protection mechanisms in the language, but also because the structure of Concurrent Pascal encourages the use of small modular procedures which are readable and understandable. These experiences confirm the arguments in favor of choosing Concurrent Pascal to support the CAR project.

On the other hand, the IO machine was somewhat of a disappointment in actual use. The concept is still valid – to provide a uniform method for handling all peripheral devices while shielding the user from the machine details such as device addressing or interrupt handling. The problem lies with the unpredictability of the existing device hardware. The disk interface, for instance, requires that a series of output commands meet critical timing constraints. Further, interrupts can occur from any of the three device addresses involved depending upon the current transfer conditions. Special cases like these complicate both the IO machine code and the Concurrent Pascal device handler, and lead to the situation where each device driver has to be individually worked out by experimentation with the hardware.

Nearly every Interdata device dealt with presented some unique interface problem, but the situation is not limited to Interdata hardware. On finding similar problems with his PDP-11 devices, Nicklaus Wirth suggested a set of hardware implementation rules for

peripheral device interfaces which would allow straight forward device handling[13]. A uniform method of handling communication with peripherals would greatly simplify the software interface problems.

All of the software work described in this report was greatly influenced by Michael S. Ball. Among his many contributions were the design of the runtime system, the general course of the compiler modifications, and the design and implementation of the final two compiler passes (code optimization and output).

6. REFERENCES

1. Brinch Hansen, P. The Programming Language Concurrent Pascal. Information Science, California Institute of Technology, February 1975.

2. Jensen, K., and Wirth, N. PASCAL: User Manual and Report. Springer-Verlag, New York, 1974.

3. Juniper, M.D. Overview of the Adaptive System Hardware Developed for the Channel Adaptive Receiver Program. NOSC TD 141, Naval Ocean Systems Center, 1977.

4. Cottel, D.M., and Zaun, J.A. Concurrent Pascal: User's Manual for the Interdata 7/16. Preliminary release, Naval Ocean Systems Center, December 1976.

5. Interdata, Inc. Model 7/16 User's Manual, Publication number 29-261, 1971.

6. Juniper, M.D. Overview of the Adaptive System Hardware Developed for the Channel Adaptive Receiver Program. NOSC TD 141, Naval Ocean Systems Center, 1977.

7. Habermann, A.N. Introduction to Operating Systems Design. Science Research Associates, Inc., Palo Alto, California, 1976.

8. Ross, D.T. Homilies for Humble Standards. Comm. ACM, 19, 11, November 1976.

9. Brinch Hansen, P. Concurrent Pascal Report. Information Science, California Institute of Technology, June 1975.

10. Brinch Hansen, P. The SOLO Operating System: A Concurrent Pascal Program. Information Science, California Institute of Technology, June 1975.

11. Interdata, Inc. Common Assembler Language (CAL) User's Manual, Publication number 29-375, 1974.

12. Brinch Hansen, P., and Hartmann, A.C. Sequential Pascal Report. Information Science, California Institute of Technology, July 1975.

13. Wirth, N. Design and Implementation of MODULA. Software – Practice and Experience, Vol. 7, 1977.

---

13. Wirth, N., "Design and Implementation of MODULA," *Software–Practice and Experience,* Vol. 7, 1977