Structured Languages SIG DECUS U.S. CHAPTER

# STRUCTURED LANGUAGES SIG NEWSLETTER

Volume 5 Number 1 April 1981

---

Out with the old, and in with the new:

The Structured Languages SIG now has a new chairman, and several new people within the SIG leadership (see the editors notes). I have decided to step down as chairman after four long years and let someone with new ideas and energy lead the SIG. I will not be dropping out of structured languages entirely, however, since I will still be interested in the implementation of the NBS-Pascal compiler. Over the years, I have seen the SIG evolve from a group of people interested in just Pascal to a group which is interested in all languages which significantly improve the programming environment on DIGITAL computers. I am sure that Bill will enhance the SIG more than I could have at this time. Being a member of the U. S. DECUS Executive Board demands more time than I have had while chairing the SIG. If everybody gives Bill the same support they gave me during these past years I am sure that the SIG will continue to flourish and grow. The SIG is now one of the four largest SIGs within DECUS. Your continued support will probably make it the largest SIG in the next few years.

The Pascal SIG was founded during the 1976 Spring DECUS Symposium in Atlanta. At that time I was chairman of the Networks SIG. The Southern California Local Users Group was heavily involved in the development of a Pascal compiler for RSX-11D at that time. The SCLUG first started to bug DIGITAL about implementing a Pascal compiler on PDP-11s during the 1974 Fall DECUS Symposium in San Diego. Seeing that DIGITAL was not going to produce a Pascal compiler in the near future, the SCLUG along with Brian Lucas at the National Bureau of Standards modified Brian's initial DOS version which was implemented in something called Block Structured Macros to run on RSX-11D and RSX-11M (we said it didn't work on IAS since DIGITAL was saying that what they were doing for IAS didn't work on RSX-11D or M). I presented a paper on the implementation at the 1977 Spring DECUS Symposium in Boston. The response was overwhelming.

I installed the compiler on the PDP-11/60 in the demonstration room and told everybody that they could copy it off of the system if they had a 1200′ tape (the 11/60 only had a TS03). Needless to say, the DIGITAL store sold all of their 1200′ tapes and the 11/60 was busy up until the time they put it on the truck. I then decided that there was a need for an active Pascal SIG, handed over the reins of the Networks SIG to Bill Brindley (that is another story), and resurrected the Pascal SIG. During the Boston symposium I also had the pleasure of sitting down next to Kathleen Jensen at a Pascal BOF session. When they passed around the sign-up sheet for interest in Pascal, Kathleen signed it first, and she just knodded knowingly to my quizical look when I saw her name and made the association. Kathleen later agreed to present a paper on Pascal entitled "Why Pascal" at the 1979 Spring DECUS Symposium in New Orleans. She was very nervous about the presentation and wanted me to back her up on the podium to answer any technical questions about the Pascal language since she only wrote the users quide and Wirth wrote the report part of her and Wirth′s book. Since that time Kathleen has come back partially into the world of Pascal and is presenting a series of tutorials on Pascal. Getting back to my history of the SIG, the years following the Boston symposium saw the Pascal SIG grow into one of the largest SIGs within DECUS. Many people contributed to its growth. Roger Vossler and Bill Heidebrecht at TRW were working closely with me in California as we refined the NBS Pascal compiler. Brian Lucas joined forces with Justin Walker at NBS and implemented a version of the compiler on Unix. Since they had finally given up on BSM, switched to C, and then finally rewrote the compiler in Pascal, I received a copy of their version over the ARPAnet at UCLA. With the help of some collegues at Hughes Aircraft, I was able to port their version over to RSX-11D and M. Also at this time we received the first version of the Swedish Pascal compiler from Seved Torstendahl in Sweden (that is why we call it the Swedish Pascal compiler). At each of the DECUS symposia since Boston, the Pascal SIG and now the Structured Languages SIG has compiled a SIG tape containing the current versions of Pascal compilers and utility routines for distribution. Bill Heidebrecht has spent many hours at symposia copying the many tapes, all of which started with the files on the PDP-11/60 at Boston. James Triplett is now our SIG librarian, and a formal tape copy procedure is in place. The 1200′ tape has grown into two 2400′ tapes containing Pascal, C, and Praxis compilers along with numerous associated software and documentation. It has been an immense pleasure to me to see and be a part of the distribution of software tools to the DECUS community. There are many other people who have helped during the years who I have not mentioned. They know what they have done, and many of you know as well.

As I said before, I am not dropping out of the

Structured Languages SIG. My involvement will be with the NBS Pascal compiler on the RT-11 and Unix operating systems. The RT-11 version of the compiler for FIS and FPP systems is available from me on floppy disks. Just call me at (406) 243-2883) to find out how to get a copy. It will also be on future SIG tapes for those of you who have tape drives.

I am looking forward to being able to contribute even more to the SIG. Even though I will not be the chairman, I will be active and waiting to see what the rest of the DECUS community will come up with in the future. Thank you for your past support and interest.

John R. Barr...

From the editor:

In this issue of the newsletter I have included the remainder of Hal Morris´ article on C from the last newsletter, a note from David O´Connor that describes an implimentation of the ´pipe´ operation under RT-11, a note from Gary Beckman who directs our RATFOR effort, and a note from Dr. James Greenwood about the implimentation of Praxis that he has developed at Lawrence Livermore National Laboratory. Please also notice the announcement from DIGITAL of the seminar "Pascal as a Second Language". Instructors for the seminar are Kathleen Jensen and Gil Roeder. This seminar is not associated with the symposium at Miami, however, Kathleen Jensen will present a one day seminar, "Introduction to Pascal", at Miami. Details are in your Preliminary Program.

We are fortunate to be able to add two very good people to our staff of newsletter editors. James Greenwood will be the feature editor for Praxis and Jim Flournoy will be the feature editor for FORTH. You are encouraged to contacted them if you have any questions, comments or articles related to their interests. Since we have had several changes and additions to the SIG staff I have included a complete list below.

The spring symposium in Miami on May 18-21 promises to be a very good one with a wide variety of presentations. The SIG is sponsoring the following sessions:

| | | |
|---|---|---|
| Mon | 2:00-3:30 | Introduction to Structured Languages |
| | 3:45-4:45 | Concurrent Languages Panel |
| | 4:45-5:45 | Concurrent Euclid |
| | 6:30-8:30 | SIMULA |
| | 8:30-9:30 | NBS Pascal Report |
| Tue | 3:45-4:45 | C for Systems Programmers |

Wed 3:45-4:45 PRAXIS and ADA
4:45-5:45 PRAXIS tutorial

Thu 2:00-3:00 Structured Languages SIG meeting

You can find a description of these sessions in your Preliminary Program. And, don't forget the pre-symposium seminar on Pascal given by Kathleen Jensen. The seminar will be held on Sunday, May 17 from 9am to 5pm.

Finally, I want to pass on to you two requests that I have received.

Does anyone have a C compiler which will run on the DECSYSTEM-20? Contact:
Ron Smith
AMG Associates Inc.
1725 Jeff Davis Hyway, Suite 704
Arlington, Vr 22202
(703) 892-5600

Does anyone have a C, Pascal, or equivalent structured language compiler which will run under RSX-11M and generate machine code for an 8085 microprocessor? Contact:
Bob Martin
Teradyne Central, Inc.
3368 Commercial Ave.
Northbrook, IL 60062
(312) 291-4300

Structured Languages SIG

Chairman - Bill Carroll
Moore Systems
1730 Technology Drive
San Jose, CA 95110
(408) 297-9920 XØ354

Librarian - James Triplett
Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138
(617) 661-1840

Symposia - Hal Morris
Park View at Madison, Apt. 18G
Lawrence Harbor, NJ 08879
(201) 949-7466

Newsletter - Roy Touzeau, Editor
Department of Computer Science
University of Montana
Missoula, MT 59812
(406) 243-2883

Feature Editors -

C

Steven McGeady
2143 SE Bush Street
Portland, Oregon 97202
(503) 235-2462

Pascal

John Barr
Department of Computer Science
University of Montana
Missoula, MT 59812
(406) 243-2883

Praxis

James Greenwood
Lawrence Livermore National Lab
Box 5508 L-464
Livermore, CA 94550
(415) 422-5369

RATFOR

Gary Beckman
Joint Center for Radiation Therapy
50 Binney Street
Boston, MA 02115

FORTH

Jim Flournoy
Box 2455
San Rafael, CA 94902
(213) 280-4365

Implementors -

NBS Pascal -

RSX

J. Bill Heidebrecht
TRW/DSSG R2/1170
One Space Park
Redondo Beach, CA 90278
(213) 535-2804

RSTS

Brian Nelson
Computer Center
University of Toledo
Toledo, Ohio

RT11

John Barr
Department of Computer Science
University of Montana
Missoula, MT 59812
(406) 243-2883

Unix

John Barr
Roger Vossler
TRW/DSSG R2/1170
Justin Walker
Interactive Systems
Gaithersburg, Maryland 20760
(301) 963-0100

Swedish Pascal - Gerry Pellitier
Transport Canada
Place de Ville
Ottawa, Canada K1A 0N8
(613) 994-2378

C - Rob Denny
Martin Minow

RATFOR - Joe Sventik
Dave Sykes

Standards - Chuck Grant
Lawrence Livermore National Laboratory
Box 5508 L-152
Livermore, CA 94550
(415) 422-3869

# PASCAL as a Second Language

| | | | |
|---|---|---|---|
| WASHINGTON, D.C.<br>January 27 - 29, 1981<br>Ramada Inn<br>Lanham, Maryland<br>(301) 459-1000 | LOS ANGELES<br>February 23 - 25, 1981<br>Del Webb's Newporter Inn<br>Newport Beach, California<br>(714) 644-1700 ext. 504 | SAN FRANCISCO<br>February 25 - 27, 1981<br>Santa Clara Marriott<br>Santa Clara, California<br>(408) 988-1500 | MINNEAPOLIS<br>March 23 - 25, 1981<br>Radisson Downtown Hotel<br>Minneapolis, Minnesota<br>(612) 333-2181 |
| DALLAS<br>March 25 - 27, 1981<br>Fairmont Hotel<br>Dallas, Texas<br>(214) 748-5454 | BOSTON<br>April 27 - 29, 1981<br>Marriott Hotel<br>Newton, Massachusetts<br>(617) 969-1000 | NEW YORK<br>April 29 - May 1, 1981<br>The New York Statler<br>New York, New York<br>(212) 736-5000 | ATLANTA<br>May 20 - 22, 1981<br>Omni International<br>Atlanta, Georgia<br>(404) 659-0000 |

Are you looking for a new development language because you are having problems implementing modern structured programming techniques?

Are your programmers asking for a more modern language to help them do their job better?

Is it time to invest in your programmers' future by training them in a modern structured language?

Are the costs of your software maintenance over budget by a factor of 2 or 3 or more?

. . . Or do you just want a good, short but intensive, course on Pascal?

If the answer is YES to any of the above questions, Digital's 2-1/2-day seminar on Pascal will be of interest to you.

**Why Pascal?**

The questions have been asked, "Why does the computer community need another language? And even if it does, why Pascal?"

**PASCAL**
**Today's Language**
**for**
**Structured Programming**

The development of any computer program goes through various stages of abstraction and refinement. Pascal was created to provide a means of expressing these abstractions more easily. With the current emphasis on structured methodology and structured programming, Pascal is an ideal language by which to standardize your programming methods.

The language constructs of Pascal inherently reflect program structures which otherwise would have to be hand-coded. By providing a facility for clearly expressing programming solutions, Pascal promotes self-documentation, thereby aiding program readability and maintainability. At the compiler level, Pascal's design provides for efficient implementation, and excellent error detection and diagnostic capability. The richness of the language combined with a growing number of excellent implementations have promoted Pascal's popularity among programmers and project leaders alike.

Pascal aids project leaders in cutting costs of program development and maintenance by allowing them to develop a clear-cut framework of programming standards. A recent *BusinessWeek* report cites one claim that programming in a structured language such as Pascal "can make programming as much as 10 times faster and can cut the cost of software by 30%-75%."

As a leader in the implementation of Pascal in the minicomputer field, Digital presents an opportunity for you to learn Pascal and the benefits it can bring to your environment.

**Features**

This seminar:

- Gives you an insight into why Pascal was developed
- Teaches you all the elements of "standard" Pascal
  - Data structures
  - Control statements
  - Structure of a Pascal program
- Exposes you to the advantages of using Pascal
  - Inherent documentation features
  - Ease of maintainability
  - Transportability of programs between systems
- Requires you to write several programming exercises demonstrating the key features of Pascal
- Shows you how Pascal is positioned in the computer language spectrum

**Benefits**

- You will make a 2-1/2-day intensive investment in learning. When you leave you'll be able to start programming in Pascal.
- You will leave with a thorough understanding of Pascal as a language.
- You will be able to take algorithms written in another language and start re-coding them in Pascal.

over

**SEMINAR OUTLINE**

- I. Introduction to Pascal
  - A. History
  - B. Why Pascal?
  - C. Program Reading
- II. Basic Concepts of Pascal
  - A. The Vocabulary of Pascal
    - Nomenclature
    - Symbols
    - Delimiters
    - Comments
    - Identifiers
  - B. Standard Data Types
  - C. Operators
  - D. Constants, Variables, and Expressions
  - E. Standard Identifiers
- III. Programming and Control Flow
  - A. Structure of a Pascal Program
  - B. Declarations
  - C. Statements
    - Simple
    - Conditional
    - Compound
    - Repetitive
- IV. Data Types
  - A. Scalar
  - B. Subrange
  - C. Structured
    - Arrays
    - Records
    - Sets
    - Files
  - D. User-Defined Data Types
- V. Procedures and Functions
  - A. Parameters
  - B. Scope
  - C. Global vs. Local Variables
  - D. Standard Procedures and Functions
- VI. Advanced Concepts
  - A. Dynamic Variables
  - B. Record Variants
  - C. Recursion
- VII. Putting It All Together
  - A. Again . . . Why Pascal?
  - B. Differences between Pascal and Other Programming Languages
  - C. Discussion
    - Extensions to Pascal
    - Status of ANSI standardization

- You will leave with a personal set of course materials that will be a convenient and valuable reference when back on the job.
- You will learn how Pascal can facilitate the process of establishing programming standards for project development.

**Who Should Attend**

- *Professional Programmers* who want to be up-to-date on one of the most popular structured languages, Pascal
- *Software Project Leaders* who are considering using Pascal for a project
- *Software Product Development Managers* who are planning to use Pascal as a development language
- *Assembly Language Programmers* who want to learn a higher level language
- *Microprocessor System Designers* who need an understanding of "standard" Pascal on which to base investigations into variations and extensions
- *Anyone* who is responsible for the productivity of programmers

Programming experience is a prerequisite for this seminar, but no knowledge of Pascal is assumed. Several programming exercises are included which will provide attendees with experience in writing Pascal. Attendees will be given time during the seminar to analyze assignments and derive solutions. There will also be two short evening assignments.

**Course Materials**

A copy of a leading textbook in Pascal, "Pascal User Manual and Report," by K. Jensen and N. Wirth, Springer Verlag, NY, 1974.

A set of Seminar Notes for future reference.

A set of handouts of debugged Pascal programs that are valid solutions to the seminar exercises.

**Faculty**

Kathleen Jensen and Gil Roeder will be your team of instructors during the Pascal seminar.

Kathleen worked for Niklaus Wirth at The Swiss Federal Institute of Technology (ETH) in Zurich for three years during the infancy of the Pascal language. While at ETH, she taught Pascal programming and co-authored the *Pascal User Manual and Report.* She also contributed to various research projects including the Pascal "P Compiler." Currently, Kathleen is employed by Educational Services within Digital Equipment Corporation where she is involved in new project development.

Gil is president of Retrieval Techology, a consulting and training firm located in Chelmsford, MA. Gil draws on an extensive background of application development and system design in several diverse industries including newspapers and publishing, energy, and manufacturing. In addition to his applications experience, Gil has been involved in intensive system development in the areas of communications and database management systems.

Gil's current interests include the design of database-oriented business systems and the integration of database management within organizations. He has used Pascal extensively in the development of these applications.

**REGISTRATION & CONFIRMATION**

To register for Digital's seminar, fill out the registration form or call the Registrar at (617)493-2858. After your reservation has been received, you will receive a confirmation letter that will include time schedules, hotel and meeting room locations, and other details. If you have additional questions, please call (617)493-2858.

Early registration is recommended. However, enrollment will remain open until two weeks before the starting date of the seminar.

## Praxis Axis

James R. Greenwood

Praxis is a high-level systems implementation language designed for control and communications programming. Praxis was developed by the Laser Fusion Program of Lawrence Livermore National Laboratory for control system programming on the Nova Fusion Facility. Three compilers and substantial documentation exist and will be in the public domain. The language project was led by Dr. J.R. Greenwood from its inception in 1978 to completion in January 1981. Praxis represents an actual investment of approximately $1,000,000 by LLNL over a two-year period. The language has been in operational use at LLNL since June 1980, and is now the standard programming language for controls applications within the Laser Fusion program.

The compilers are written in Praxis (  40,000 lines) and operate on two processors: VAX/VMS and PDP-11/RSX-11M. The kernal of the Nova control system written in Praxis (35,000 lines) is operational at this time. Three compilers are operational:

VAX/VMS generating VAX code
VAX/VMS generating PDP-11 code
PDP-11/RSX-11M generating PDP-11 code

Praxis is the practice of the programming art, science, and skill. It is a high-order language designed for the efficient programming of control and systems applications. It is a comprehensive, strongly typed, block-structured language in the tradition of Pascal, with much of the power of the unavailable Ada language. Praxis supports the development of systems composed of separately compiled modules with user defined data types and exception handling. Also sophisticated control constructs, and encapsulated data and routines are built into the language. Direct access to machine facilities, efficient bit manipulation, and interlocked critical regions are provided in the language.

Additional information is provided in the manuals listed below which are available from NTIS or LLNL:

| | |
|---|---|
| o An Introduction to Praxis | UCRL-52957 |
| o Praxis Language Reference Manual (315 pages) | UCRL-15331 |
| o Programming in Praxis (230 pages) | UCRL-5xxxx |
| o Praxis Input/Output Interface Report | UCRL-15xxx |
| o Praxis Internals Document | UCRL-15xxx |

Published reports which describe the language and implementation include:

- o A Comparison of Programming Languages: Ada, Praxis, Pascal, C — UCRL-15xxx
- o A Tour through the Praxis Compiler — UCRL-15xxx

The public domain versions of the compilers are being distributed currently. Contact:

Dr. J.R. Greenwood
Lawrence Livermore National Laboratory
P.O. Box 5508 L-481
Livermore, CA. 94550
(415) 422-5369

In addition negotiations are under way with some vendors to distribute, support, and maintain the DEC compilers. In addition Praxis will be made available under UNIX and other 16 bit micros.

In summary, the Praxis language is specifically designed to be WITHIN the state of the art of language design yet suitable for advanced control and systems implementation needs. Complex language features such as generic procedures, overloading of operators, and predefined tasking have been intentially omitted. We felt that these concepts were either not understood well enough to be incorporated at this time, or that they should not be part of the language.

In conclusion, Praxis is an extremely powerful, modern programming language that goes beyond Pascal and yet is available today.

JG:vb
2109f

# RATFOR

Ever since my name appeared in both DECUSCOPE and the SIG's newsletter, there has been a steady stream of mail for me asking that the sender be kept informed of what is happening in and around RATFOR. Some people have even requested information concerning specific preprocessors. It is my hope that this newsletter be the source of such information. I am unable to answer individually each request to be kept up to date, which is why I volunteered my time to be the editor of the RATFOR part of the SIG's newsletter. If you have questions, suggestions, or interesting things you have done send them to me:


Gary Beckmann<br>
Joint Center for Radiation Therapy<br>
50 Binney Street<br>
Boston, MA 02115


Please send everything in camera ready form (8.5x11 paper, inch margins all around, and dark enough print for reproduction -- if you can't get a decent Xerox of it I probably can't either). If your question is about a specific preprocessor, I will do my best to get the question to the implementors of it and an answer from them. Everything else that seems within reason will go into the newsletter. (If you are claiming to have implemented an operating system in RATFOR, please be prepared to substantiate that claim.)

There are many versions of the RATFOR preprocessor floating around and I would like to make some attempt to bring them all together and produce one preprocessor that the SIG would 'support'. There are two formal groups of which I am aware that are working presently with a RATFOR preprocessor: one is the Structured FORTRAN Working Group of the RSX/IAS SIG and the other is the Software Tools people. I would like to hear from everyone who has implemented a preprocessor and who is interested in starting some communication as to what should and should not be included in a preprocessor. This information could then be offered to the community of users by the SIG. I will attempt to act as a nexus in this communication. Hopefully we can schedule a meeting or two at the next symposium in Florida where we can thrash over our thoughts in person.

Feel free to write -- sometimes my queue gets pretty full but I eventually get around to responding in some manner. Until the next newsletter,

Gary

Editor's note: The following article is the second part of the introductory article on the programming language C written by Hal Morris. It is continued from the last issue of the newsletter.

I/O Processing and Interrupts on the PDP-11

This section gives some examples of how to do I/O using C without the aid of an operating system. The programs presented could in fact be run without an operating system assuming one can find a way to load and start them. My purposes in doing this are:

(1) To show that it can be done (even efficiently).

(2) To provide for some people (including myself) a better basis for understanding I/O on the PDP-11 (and similar machines) than the usual method which relies heavily on assembler language routines in which anything of generality is hidden by obscure details.

(3) To provide simple models for real I/O programming in C. For instance, stand-alone applications, perhaps because the machine is very small, or the application very specialized, and no operating system does an adequate job. Another use for bare-machine I/O is an inline device driver, that is one which is linked into the application program. There is some discussion of when to use this technique and when to write a handler in the RT-11 Self-Paced Course. This is what my biggest example is, at any rate. It is fairly easy, at any rate, involving a 6-line assembler "front-end" for the Interrupt Service Routine, and otherwise, only C code. Essentially the same front-end could be used on any inline device driver. An actual handler (to be installed in the operating system) is a good bit more difficult for most operating systems, which make some stiff requirements on its structure (E.g., fixed offsets from the beginning and/or end of the handler act as variables through which the system communicates with the handler; the number of words in the handler must be contained in some fixed location.). Nevertheless, with some help from assembler inserts, front-ending, and so forth, one can write an RT-11 driver in (mostly) C. For a complex enough device, it may be quite worthwhile, which is the oppinion of an acquaintance who wrote a CAMAC handler in C for RT-11. Apparently UNIX (trademark of

Bell Laboratories), which is mostly written in C, is written to make it much easier to write handlers in C.

UNIBUS Facts for the Hardware Novice:

I must agree that the concept of the UNIBUS and related bus structures is as outstanding as DEC keeps telling us. C can do most bare-machine I/O by itself on any machine with such a structure, whereas without the UNIBUS or a related bus structure, generalized I/O (as opposed to reading and writing text) is so specialized to the machine that it can not be built into a high-level language A device plugged into the UNIBUS communicates with the CPU through its own device registers, which, to the CPU (and to your program) work just like memory locations except that their addresses are higher than normal memory. To communicate with the device, the CPU, under control of a program, moves a word or byte into a device register or sets certain of its bits. The meaning of this to the device may be "Print this character.", or "Don't bother me.", or "It's O.K. to bother me now." (technically: enable or disable interrupts). All this depends on what bits were set or which register got a word/byte moved into it. The device may place information in one of its own registers to say something about its state. If a program must wait for a device to be in a certain state, it may repeatedly read a device register, staying in a tight loop until the desired state is attained. Typical meanings of messages from the device are: "I've got new input in my other register.", "Help me, I'm out of paper.", or "I'm not busy now.". Such states of the device may also cause interrupts, but a discussion of device registers without interrupts should come 1st.

NOTE 1: On the smaller 11s, the highest 2K or 4K numbers which could be addresses, i.e., could be written with 16 bits, are simply not allowed to be memory addresses, and are reserved for use as device register addresses. This is why LSI-11s, 11/10s, and others can have at most 28K or 30K words of memory, instead of the 32K words which the 16 bit size of an address would allow.

NOTE 2: An intelligent but not-yet knowlegeable reader might well be bugged by the fact that his/her terminal can be attached to a number of makes of computer, whereas I am talking about devices having properties specific to the UNIBUS. Since I am going to be dealing with a terminal in most of the examples, I should point out that a terminal is not plugged into the UNIBUS. It is connected to a controller or interface, which is plugged into the UNIBUS and which is made only for UNIBUS computers.

C Examples: I/O Using Only Device Registers:

I will refer to the printing mechanism or video display of a terminal as just the "printer". Either receives data from the interface (NOTE 2) in the same way. I am going to write a program for a terminal interfaced as the console to output the letter 'C'. The console has 4 1-word registers starting at address 0177560, of which the 4th (6 bytes beyond the 1st) is the output buffer register. Moving a character this register's low order byte will cause it to print. I will make the output buffer accessible via OBUF, having the properties of a char variable, via the following macros:

```
#define ByteAt(loc)       (* (char *) (loc))
        /* Recall this is contents of loc treated as a pointer
         * to a char variable.
         */
#define DLREGADDR         0177560
#define OBUF              ByteAt(DLREGADDR+06)
```

Then, provided I don't collide with the operating system, the following program will print the letter 'C':

```
[ #defines]
main()
{
        OBUF = 'C';
}
```

If I want to print several characters, there is a slight problem, which is that the CPU can send characters much faster than the terminal can recieve them, and it does not wait to see what happens to the last character it sent before sending a new one. This is one purpose of the other register associated with terminal output, called the Status Register. This tells various things about the state of the device, and in particular bit 7 is 1 if the printer is ready to receive a character and 0 if not. (Bit 7 is only set by the device; attempts by a program to set it will be ignored.) The condition that bit 7 is set can be expressed as follows:

```
#define Bit(n)        (1 << (n))
```

(A word with just bit n set)

```
#define IsOn(n, x)       notNULL((x) & Bit(n))
```

x & Bit(n) is non-zero (and equals Bit(n)) iff bit n of x is on. notNULL means what is says:

```
#define notNULL(whatnot)    (whatnot)
```

(whatnot has the same truth value as whatnot != NULL.)
Finally:

```
#define IsOff(n, x)        (!IsOn(n, x))
```

So the program becomes:

```
[ miscellaneous #defines ]
#define OSTAT   WordAt(DLREGADDR+04)

main()
{
/*1*/  static char    out[] = "Hello world\r\n" ;
       register char  *o;
/*2*/  o = out;
       while(notNULL(*o))
       {     while(IsOff(7, OSTAT))
                  {}       /*  Sit tight. */
/*3*/        OBUF = *o++;
       }
}
```

The program just loops as long as the printer says it isn't ready to print (Bit 7 of the STATus register "IsOff".) Note that in [1], out is an array of characters (whose size is determined by the initialization. [2] and [3] use the buffer and pointer technique described in the example of printing the system date.

Interrupt-Driven I/O:

The next big exercize will be a program which prints several lines of text using an inline asynchronous device driver. This is linked into the program; it is not installed in the operating system.

The console terminal interface, which on my machine is a DL-11, has one type of interrupt which says that a character has been typed at the keyboard, and another type of interrupt associated with printing. The printer interrupt occurs whenever the printer is ready to print a character. I.e., it interrupts whenever it isn't busy. Now this can't be quite true. If it were, the printer would continuously interrupt any time the CPU could not keep it busy. This leads to the subject of enabling and disabling interrupts. When a particular type of interrupt is disabled, then the event which would normally cause it doesn't cause it. This may be illustrated very simply.

Normally, keyboard interrupts are always allowed to happen. This means that whenever you type something on the keyboard, the monitor goes and does something, even if it seems to ignore the character typed. In particular, it will note whether the character typed was a ctrl/C or not, and if 2 ctrl/Cs are typed consecutively, a running program will be aborted, unless this feature has been overridden. Let me broadly sketch what the interrupt does. It takes control away from the running program, regardless of what is going on, saves any registers to be used, as well as "status" information (so that an interrupt can occur between "test x" and "branch if 0", for instance) and then it looks at what was typed. Then it goes through a complex set of decisions and actions. If the running program has requested a line of text (from TT:) for instance, then the new character from the keyboard usually goes into a buffer. Under the same circumstances, if the character is a carriage return, the buffer's contents are moved to where the user said to put the next line, and a line feed and a NULL (0) byte are appended. If the character is a ctrl/U, the buffer is cleared. A ctrl/C may cause an abort, and so forth. The demonstration program will turn off the keyboard interrupt so that ctrl/C will have no affect. The program will also ring the terminal bell 100 times so that you know when it is running (Try writing and debugging such a program without the last feature.).

We are still dealing only with device registers. The convention with UNIBUS devices is to have a register called the Status Register such that setting bit 6 (to 1) enables interrupts, while clearing it (setting it to 0) disables interrupts. Here are two macros for turning on and off specific bits in anything comparable to an int:

```
#define TurnOn(n, x)        ((x) |= Bit(n))
#define TurnOff(n, x)       ((x) &= ~Bit(n))
```

This leads to:

```
#define EnabInt(statreg)   TurnOn(6, statreg)
#define DisabInt(statreg)  TurnOff(6, statreg)
```

which turns interrupts for a given device on or off, given the device's Status Register in a form such as OSTAT above. The terminal's input, or keyboard status register may be defined as:

```
#define ISTAT WordAt(DLREGADDR)
```

since it is the st of the DL-11's registers.

So finally here is the program:

```
[ miscellaneous #defines ]
#define BELL    07
      /* Ascii char for ctrl/G, i.e. BELL. */
main()
{
      long   i; /* 32 bit integer (on PDP-11) */

      DisabInt(ISTAT);

      for (i=0L ; i<10000000L ; ++i)  /* for i=1 to 10000000 */
           if(i % 100000L  == 0)    /* 10000 divides i evenly */
                 OBUF = BELL;

      EnabInt(ISTAT);

      while(YES)
           ;          /* Infinite loop, but ctrl/C now works */
}
```

(Constants ending in ´L´ have 32 bits.) To test this properly, you need to run the program, then type ctrl/C frantically while the terminal beeps stupidly. This will probabally remind you of some real life experience. When it stops beeping, it goes into an infinite loop, but it can now be made to abort via a ctrl/C.

Interrupts for Hardware Novices:

Besides "pseudo memory locations" known as device registers, most peripherals are wired to have a special relationship with a pair of words in low memory called an interrupt vector (in fact, possibly one vector for input and one for output). The vector is filled by software (unless it´s ROM) with:

(1) the address of an Interrupt Service Routing (ISR) where control transfers when the device´s interrupt occurs,

(2) a value for "PS", or Processor Status, including the priority (a 3 bit number 0-7) which the CPU assumes when the interrupt occurs.

When the device´s interrupt is accepted (which may not happen quite as soon as the device requests it), the following occurs:

(1) The current PC (Program Counter, which is register 7), and current PS are placed on the stack so that the ISR can restore them to their pre-interrupt values if it wishes to act civilized.

(2) New values for the PC and PS are taken from the interrupt vector of the interrupting device. The replacement of PC should start the CPU executing a routine which is designed to respond to the interrupt. Of course with faulty software it might really go anywhere and almost certainly result in a system crash.

Priority levels and Interrupts:

At this point, you should know at least one thing about the CPU´s priority level, which is that it may change in response to interrupts (see comments on "PS", above). I will use the term "soft priority" of a device for the priority encoded in the second word of the interrupt vector, which is the priority the CPU takes on when the device interrupts. "Soft priority" is, as implied, a function of what software put in the device´s vector. Each device also has a "hard priority", which is wired into the device. When a device attempts to interrupt, if its hard priority is higher than the priority at which the CPU is running, the interrupt occurs. Otherwise, it is kept pending, i.e., waiting until the CPU´s priority falls below the devices hard priority. If you consider, for instance, that the system clock´s interrupt cannot be pending more than 1/60th of a second, or the system time will become inaccurate, it should be clear that the system cannot run for long periods at maximum priority. If a your interrupt is kept pending, the CPU is probably executing an ISR for a device with soft priority at least as high as your hard priority. An interesting consequence is that a devices soft priority should be at least as high as its hard priority, or it might interrupt its own ISR, which for various reasons is very undesireable.

C Example: Terminal printer driver

The interrupt-driven printer driver presented has 2 parts. In addition to the ISR which jumps in whenever the printer interrupt is enabled and the printer is ready to print, there is an initialization routine "InitOut()", which helps the ISR know what to do when the printer wants to print (tells it where to find some characters to be printed), and then tells the printer to interrupt (invoking the ISR) when it is ready.

The main program calls InitOut() when it has a line to be printed. The arguments are the line to be printed, and a pointer to an address, "done" which is to equal NO (== 0) while there are characters left to print, and is set to YES (== 1) when the ISR wakes up (the printer is ready) and there is nothing left to print. InitOut() halts any ongoing printing job, copies the two arguments to variables which it and the ISR share, sets user's "done" to NO, and then tells the printer to interrupt when ready. Note that the printer will interrupt immediately unless it is printing the last character from its last request. When the ISR wakes up due to an interrupt, "out" and "pDone" will point to the line to be printed and the user's done signal.

The driver requires the main program to place the address of the ISR in the vector, which it does. This is where one starts needing a good bit of knowlege about a particular machine and compiler. First consider the C program "isr()":

```
...
isr()   /* C Interrupt Service Routine */
{
        if(isNULL(*out))          /* out of output */
        {       *pDone = YES;
                DisabInt(OSTAT);
        }
        else                      /*   send to OBUF; advance */
                OBUF = *out++;    /* the char pointer "out". */
}
```

Via inspection of the assembler translation of this program and some reading of Whitesmith's documentation, you would see that:

(1) The routine does not save registers 0 and 1. All other registers are preserved across function calls, however.

(2) It is not designed to be an interrupt routine. An interrupt service routine must (besides preserving all registers) return via the ReTurn from Interrupt, or RTI instruction, which restores PS (processor status, including priority and condition codes). Actually, there are equivalent things which it can do (and which RT-11 handlers do), but an ordinary subroutine return is not appropriate.

(3) The solution to problems presented by (1) and (2) involves noting that isr() can be called from an assembler program via "JSR PC,ISR". Thus a very simple assembler program (called ISRMAC) can act as a front end for isr(). An interrupt service routine should have no arguments, which is why the call is so simple.

So, part of the solutions (one which can be applied to any inline C interrupt routine) is to make the C function a subroutine of an assembler function which

(1) saves R0 and R1
(2) calls the C interrupt routine
(3) restores R0 and R1, possibly destroyed by the C routine,
(4) returns via "RTI".

Here is the macro "front end" program:

```
        .GLOBL isr,ISRMAC
        .PSECT  c$text
ISRMAC:
        MOV     R1, -(SP)
        MOV     R0, -(SP)

        JSR     PC,isr              ;Call C counterpart

        MOV     (SP)+, R0
        MOV     (SP)+, R1
        RTI                         ;Return From Interrupt
        .END
```

Since a DL-11 has 2 consecutive vectors, the 1st for input and the 2nd for output, its vector address is generally given as the <u>input</u> vector address, while the output vector is 4 bytes beyond that. Thus I will define DLOVEC, the output vector as follows:

```
#define  DLVECADDR    060
#define  DLOVEC       ((unsigned *) (DLVECADDR+04))
```

That is, DLOVEC is the 2nd vector treated as a pointer to an unsigned variable, which may be treated as an array of (2) unsigneds. Thus the vector gets its new contents via:

```
DLOVEC[0] = <address of ISRMAC>;
DLOVEC[1] = <New PS value.>;
```

However, the program should also save the old contents of the vector and restore them when it is done so that when it is finished, the operating system will be able to print things the way it normally does. So the main program to test the printer driver now looks like:

```
...
extern   ISRMAC();
unsigned OldVec[2];
...

DisabInt(OSTAT);  /* While vectors are being changed, an */
                  /* interrupt has unpredictable effect. */
OldVec[0] = DLOVEC[0];
OldVec[1] = DLOVEC[1];
DLOVEC[0] = ISRMAC;       /* 1 */
DLOVEC[1] = Priority(7); /* 2 */

...[exercize driver]

DLOVEC[0] = OldVec[0];
DLOVEC[1] = OldVec[1];
...
```

Note on /* 1 */: Getting at the true starting addresses of subroutines is naturally system dependent and may lead to rather misleading code. To do it, I may declare ISRMAC as an extern function (i.e. global). If I declare some identifier as an extern function, then a reference to it not followed by a parenthesized argument list (not even an empty argument list, like "()") will act in a very system-dependent way as the address of the starting address of the function with that name (if one can be found). I could print ISRMAC, and get the same number that I find for ISRMAC on a linker load map. A non system-dependent purpose for this is to allow the passing of functions to other functions so that, for instance, a graphics function to plot points could have the mathematical function which it is to plot as an argument. Function names end up being global labels in the assembler translation of the C program, and are interchangeable with labels of programs originally written in assembler.

Note on /* 2 */: Now, consider the second word of the vector, the Processor Status. Normally with RT-11 and no memory management, only the 3 bits starting at 5 which represent the priority should be non-zero. So I can use the following to generate a PS for a given priority:

```
#define Priority(n) ((n) << 5)
```

In particular I want the PS to make the CPU run at priority 7.

Finally, here are the complete driver and test program:

[I WILL JUST ATTACH THEM FOR NOW]

It occupies 440 words of which 256 are part of or below the (normal sized) stack.

Note on C features used: On the "for" statement, see Kernighan and Ritchie, p 16 and section 3.5. "Call by value", which is the reason for the peculiar way of passing "done", has several references in the index, as does the "static storage class", which is the basis of the "sharing" of out and pDone between InitOut() and isr(). A somewhat similar use of static variables is in section 5.7 (a date conversion routine). K&R´s index, by the way, is very thorough, which is one reason it is quite a nice reference.

The way this program cycles waiting for "done" fails to illustrate the point of interrupts, which is to allow other work to get done while the CPU is waiting on a device to do something. One way to get some overlap between computation and output is to only wait for the printer if one wants to print something. An output routine might be devised which just gets things going and returns immediately if the printer is free, but a if printer job is being finished, it has to wait on it. Much more sophisticated things could be done; in particular, with multiprogramming, when one job is waiting for output to finish, another can be be running. But here is a modest way of benefitting from interrupts:

```
...
static char   buf[BUFSIZE];
static BOOL   done = YES; /* 1 */

OupLine(line)
char    *line;
{
      copy(buf, line); /* 2 */
      while(!done)
         ;             /* 3 */
      InitOut(buf, &done); /*  Start this job; can´t start    */
                           /* another til this one´s finished. */
}
```

Some final notes on the above program:

/* 1 */: This is YES the first time a request is made and thereafter depends on whether the job requested on the last call has finished or not. (BOOL is #defined to be int, just to give an air of respectability to using integers an booleans.)

/* 2 */: This allows the caller to not worry about using its own buffer (pointed to by line). Note again the necessary use of static.

/* 3 */: Wait until the output from the last request is finished.

If perchance every printed line is followed by enough computation to allow it to finish printing, then the OupLine will always find done true at the start, so the amount of time spent doing output will just be a little over the amount of time spent in the ISR, which is negligible.

Some of my example programs, as well as some of my understanding of bare machine I/O derives from a new book: PDP-11 Assembler Language Programming and Machine Organization (Michael Singer, (c) 1980, John Wiley and Sons), Chapter 4, "Peripheral Devices". I like it the best of any book on PDP-11 assembly language I've seen.

```
.!      DLDR.C:

/*      DLDR.C  =  DL-11 Output driver consisting of:
 *      (1) InitOut(): Point ISR at a line of text and say go.
 *      (2) isr(): Jumps in whenever interrupts on and printer ready.
 */

#include        <c:std.h>
#include        <dldr.h>

/*   Shared by InitOut() and isr(): */
static  char    *out;
static  BOOL    *pDone;

/*      Output initiation routine:
 *         Tell ISR where line to print is, and where user's
 *      done signal is; tell printer to interrupt when ready.
 */
InitOut(NewOut, pNewDone)
char    *NewOut;
BOOL    *pNewDone;
{
        if(isNULL(*NewOut))     /*  Printing empty string? Just */
                *pNewDone = YES;/* tell caller he's done.       */
        else
        {       DisabInt(OSTAT);        /* Unfinished job killed */
                out     = NewOut;
                pDone   = pNewDone;
                *pDone = NO;
                EnabInt(OSTAT);

        }
}

/*
 * Interrupt Service Routine; Called by ISRMAC() upon interrupt:
 */

isr()
{
        if(isNULL(*out))
        {       DisabInt(OSTAT);
                *pDone = YES;
        }
        else
                OBUF = *out++;
}
```

```
!       TDLDR.C
/*      T D L D R . C  =  _main.
 *         Test DLDR.C:  DL driver written in C.
 */

#include        <c:std.h>
#include        <DLDR.h>

_main()
{
        BOOL            done = NO;
        static char     *out[] =
                {"outl\r\n", "out2\r\n", "out3\r\n", NULL} ;
        register int    i;
        unsigned        OldVec[2];
        extern          ISRMAC();       /* Macro front end of Interrupt */
                                        /* Service Routine.             */

        /* Save old vector and replace with ours. */
        DisabInt(OSTAT);
        OldVec[0] = DLOVEC[0];
        OldVec[1] = DLOVEC[1];
        DLOVEC[0]  = ISRMAC;
        DLOVEC[1]  = Priority(7);

        for(i=0 ; notNULL(out[i]) ; ++i)
        {       InitOut(out[i], &done) ;
                while(!done)
                        ; /* Wait till output completes */
        }

        /* restore vector */
        DLOVEC[0] = OldVec[0];
        DLOVEC[1] = OldVec[1];
}
```

```
!       DLDR.H:
/*      D L D R . H  =  constants & macros
 *         for DLDR, or DL-11 driver.  Note that this file determines
 *      the specific DL-11 for which the driver will work.
 */

/*==================================================================*/
/* TOOLS FOR BIT MANIP. AND ABSOLUTE ADDRESS REFERENCING:          */
/*==================================================================*/
#define WordAt(loc)                 (* (unsigned *) (loc))
#define ByteAt(loc)                 (* (char *)  (loc))

#define Bit(n)                      (1<<(n))
#define IsOn(n, x)                  ((x) & Bit(n))
        /*  Bit n of x is on.  Equal to Bit(n), which is true, if so. */
        /* It doesn't equal the "canonical true value" YES, or 1. */
#define IsOff(n, x)                 (!IsOn(n, x))
        /*  Is bit n of x off? */
#define TurnOn(bitnum, x)           ((x) |= Bit(bitnum))
#define TurnOff(bitnum, x)          ((x) &= ~Bit(bitnum))

/* MISC. */
#define notNULL(whatnot)            (whatnot)
#define isNULL(whatnot)             (!(whatnot))

/*==================================================================*/
/* DEFINES WHICH DL-11 THE PROGRAM IS FOR. (NEEDN'T BE CONSOLE) */
/*==================================================================*/
#define DLREGADDR        0177560
#define DLVECADDR        060

/*==================================================================*/
/* TOOLS FOR BASIC I/O; GETTING AT DEVICE REGISTERS, ETC.:         */
/*==================================================================*/
#define OSTAT            WordAt(DLREGADDR+04)
#define OBUF             ByteAt(DLREGADDR+06)

#define DLOVEC   ((unsigned *) (DLVECADDR+04))

#define EnabInt(statreg)            TurnOn(6, statreg)
#define DisabInt(statreg)           TurnOff(6, statreg)

#define Priority(n)                 ((n) << 5)
```

ISRMAC.MAC:

```
;=====================================================================
;       This routine is entered on interrupt since its address
;       will be placed in the appropriate vector.  It in turn calls
;       the C function which does most of the work.
;          The job of this routine is to preserve R0 and R1 (any C
;       function is guaranteed to preserve the rest), call the
;       C funtion isr(), and return with a RTI (ReTurn from Interrupt).
;=====================================================================
        .GLOBL  isr,ISRMAC
        .PSECT  c$text
ISRMAC:
        MOV     R1, -(SP)
        MOV     R0, -(SP)

        JSR     PC,isr          ;Call C counterpart

        MOV     (SP)+, R0
        MOV     (SP)+, R1
        RTI                     ;Return From Interrupt
        .END
```

TDLDR.LNK:

```
LINK/EXEC:TDLDR/MAP:TDLDR C:CHDR,TDLDR,ISRMAC,DLDR,C:CLIB
```

TDLDR.MAP:

```
RT-11 LINK  V05.04E     Load Map        Wed 20-Aug-80 00:42:30
TDLDR .SAV        Title:  START   Ident:

Section  Addr    Size    Global  Value   Global  Value   Global  Value

. ABS.   000000  001000   (RW,I,GBL,ABS,OVR)
C$TEXT   001000  000462   (RW,I,LCL,REL,CON)
                         C$STAC  001000  .MAIN   001020  ISRMAC  001166
                         INITOU  001204  ISR     001266  EXIT    001336
                         ONEXIT  001412  C$SAV   001426  C$RET   001444
                         C$RETS  001454
C$DATA   001462  000114   (RW,I,LCL,REL,CON)

Transfer address = 001000, High limit = 001576 =   447. words
```

Editor's note: The 'pipe' in the UNIX* operating system is a convenient way to pass output from one program to another. Programs which transform their input into some other form are often called filters. Filters and pipes can be used to express rather complex transformations of data. A simple example which I have found useful is:

ls | pr -4 -l1 -t

ls is a system (shell) command which produces an alphabetic list of the files in the directory. This output is sent directly to the program invoked by the pr shell command which produces formated output. The arguments to pr specify: (1) 4 column output, (2) a page length of 1, and (3) no header. The result is a list of files spaced across the terminal screen rather than zipping down the left margin, the early ones disappearing off the screen before I can see their names.

Documentation for RT-11 pipe program 'pipe.sav'

author - David L. O'Connor
370 Eden St.
Buffalo, N.Y. 14220
716-828-0898

The purpose of 'pipe.sav' is to provide limited support for the UNIX pipe notation on RT-11 version 3b and later operating systems. The program prompts the user for a command line with '>' and then translates the command line into the RT-11 indirect file 'pipe.com' which implements the intent of the command line. Pipe.sav executes pipe.com on exit.

Detailed description of the legal syntax follows:

1. The " | " string is used as a delimiter between pipe segments. Eg:

   input | program -flag | output

2. The first segment must contain one RT-11 filespec for the input file, or if none is specified, input defaults to the terminal.

3. The last segment must contain one RT-11 filespec for the output file, or if none is specified, output defaults to the terminal.

*Trademark of Bell Laboratories.

4. The other segments are referred to as internal segments, and contain the name of an executable program, followed by optional flags, auxillary input files, and arguments. There are two types of internal segments; one for RT-11 programs which accept standard CSI input, and one for WHITESMITHS 'c' programs which accept an 'argv' and 'argc' command line with < and > redirection of input and output.

5. CSI internal segments begin with '* ', then the executable program name, followed by up to 5 optional auxillary input files, followed by 1 optional CSI option string. Eg:

   ```
   ! * pip file1 file2 file3 /U !
   ```

6. 'c' internal segments begin with the executable program name, followed by any combination of ASCII strings. Eg:

   ```
   ! find -n match !
   ```

---

Example of the indirect file created by a sample command line.

```
! RT-11 pipe version 2 ; author - David L. O'Connor
!
! the next line is the pipe command line.
! *.txt ! * sy:pip /U ! sort -n -d ! unique ! compar dict ! errors
!
set error error
run sy:pip
pipe.tm1=*.txt/U
↑C
run sort
<pipe.tm1 >pipe.tm2 -n -d
↑C
run unique
<pipe.tm2 >pipe.tm1
↑C
run compar
<pipe.tm1 >errors dict
↑C
set error none
delete/noquery pipe.tm1,pipe.tm2
set error error
reset
```

Restrictions & comments

1. 'c' programs using 'main' must return(YES) or exit(YES) to function with pipe.

2. Command line must not exceed 80 characters.

3. [n] notation not allowed after RT-11 filespecs eg: test.txt[100].

4. Although pipe allows lower case input, and lower case file names execute properly, RT-11 system utilities need upper case options eg: | * sy:pip /U | .

5. I never tested pipe with the F/B monitor.

DECUS

DIGITAL EQUIPMENT COMPUTER USERS SOCIETY
ONE IRON WAY, MR2-3/E55
MARLBORO, MASSACHUSETTS 01752

BULK RATE
U.S. POSTAGE
PAID
PERMIT NO. 129
NORTHBORO, MA
01532

**MOVING OR REPLACING A DELEGATE?**

Please notify us immediately to guarantee continuing receipt of DECUS literature. Allow up to six weeks for change to take effect.

( ) Change of Address
( ) Delegate Replacement

DECUS Membership No.: ____________________

Name: ____________________

Company: ____________________

Address: ____________________

____________________

State/Country: ____________________

Zip/Postal Code: ____________________

Mail to: DECUS - ATT: Membership
One Iron Way, MR2-3
Marlboro, Massachusetts 01752 USA

Affix mailing label here. If label is not available, print old address here. Include name of installation, company, university, etc.